

**LAPORAN TUGAS BESAR 1**  
**“PENCARIAN SOLUSI DIAGONAL *MAGIC CUBE* DENGAN *LOCAL SEARCH*”**  
**IF3070 DASAR INTELIGENSI ARTIFISIAL**  
Dosen Mata Kuliah: Dr. Nur Ulfa Maulidevi, S.T, M.Sc.



**Disusun Oleh:**

Kelompok 43

Eldwin Pradipta	/ 18222042
Mattheuw Suciadi Wijaya	/ 18222048
Theo Livius Natanael	/ 18222076
Muhammad Ridho Rabbani	/ 18222098

**PROGRAM STUDI SISTEM DAN TEKNOLOGI INFORMASI**  
**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**  
**INSTITUT TEKNOLOGI BANDUNG**  
**BANDUNG**

## DAFTAR ISI

<b>HALAMAN JUDUL</b>	<b>1</b>
<b>DAFTAR ISI</b>	<b>2</b>
<b>DESKRIPSI PERSOALAN</b>	<b>4</b>
<b>PEMBAHASAN</b>	<b>5</b>
A. Pemilihan Objective Function	5
Keterangan :	6
B. Implementasi Algoritma Local Search	7
1. Hill-Climbing Search	7
1) Steepest Ascent Hill-Climbing	7
2) Hill-Climbing with Sideways Move	9
3) Stochastic Hill-Climbing Search	12
4) Random Restart Hill-Climbing Search	14
2. Simulated Annealing	17
3. Genetic Algorithm	20
C. Hasil Eksperimen & Analisis	25
1. Hasil Eksperimen	25
1. Steepest Ascent Hill Climbing Search	25
● Percobaan Ke - 1	25
● Percobaan Ke - 2	27
● Percobaan Ke - 3	29
2. Hill-Climbing Search with Sideways Move	31
● Percobaan Ke - 1	31
● Percobaan Ke - 2	33
● Percobaan Ke - 3	35
3. Stochastic Hill-Climbing Search	37
● Percobaan Ke - 1	37
● Percobaan Ke - 2	39
● Percobaan Ke - 3	41

4. Random Restart Hill-Climbing Search	44
5. Simulated Annealing	50
● Percobaan ke-1	50
● Percobaan ke-2	52
● Percobaan ke-3	54
6. Genetic Algorithm	57
● Percobaan Ke - 1	57
● Percobaan Ke - 2	59
● Percobaan Ke - 3	60
● Percobaan Ke - 4	61
● Percobaan Ke - 5	63
2. Analisis	64
<b>KESIMPULAN &amp; SARAN</b>	<b>71</b>
A. Kesimpulan	71
B. Saran	71
<b>PEMBAGIAN TUGAS KELOMPOK</b>	<b>72</b>
<b>REFERENSI</b>	<b>75</b>

## DESKRIPSI PERSOALAN

Terdapat sebuah *diagonal magic cube* yang tersusun dari angka 1 hingga  $n^3$  tanpa pengulangan n dengan panjang sisi kubus tersebut. *Diagonal magic cube* tersebut berukuran  $5 \times 5 \times 5$ . Angka-angka yang tersusun pada *diagonal magic cube* tersebut memiliki aturan sebagai berikut.

1. Terdapat satu angka yang merupakan *magic number* dari kubus tersebut ( $M$ ).
2. Jumlah angka-angka pada setiap baris sama dengan *magic number* ( $S_{row}$ ).
3. Jumlah angka-angka setiap kolom sama dengan *magic number* ( $S_{col}$ ).
4. Jumlah angka-angka setiap tiang sama dengan *magic number* ( $S_{tiang}$ ).
5. Jumlah angka-angka untuk seluruh diagonal ruang pada kubus sama dengan *magic number* ( $S_{d.ruang}$ ).
6. Jumlah angka-angka untuk seluruh diagonal pada suatu potongan bidang dari kubus sama dengan *magic number* ( $S_{d.bidang}$ ).

Rumus dari untuk menghitung *magic number* ( $M$ ) itu sendiri yang berukuran  $n \times n \times n$ , yakni sebagai berikut  $M = \frac{n(n^3 + 1)}{2}$ . Berdasarkan permasalahan yang harus diselesaikan dimana, nilai  $n = 5$  maka hasil perhitungan *magic number* dari *diagonal magic cube* yang berukuran  $5 \times 5 \times 5$  ialah sebagai berikut.

$$M = \frac{5(5^3 + 1)}{2} = \frac{5(125 + 1)}{2} = \frac{5(126)}{2} = 315$$

Berdasarkan hasil perhitungan *magic number* di atas, dapat dilihat bahwa nilai *magic number* untuk *diagonal cube* berukuran  $5 \times 5 \times 5$  adalah 315. *Initial state* dari suatu kubus untuk *diagonal magic cube* dengan ukuran  $5 \times 5 \times 5$  adalah susunan angka 1 hingga  $5^3$ . Pada tiap iterasi algoritma *local search* yang dilakukan, solusi objektif yang diusulkan haruslah dapat menukar posisi 2 angka yang terletak pada kubus tersebut.

## PEMBAHASAN

### A. Pemilihan *Objective Function*

Pada kasus *diagonal magic cube* yang ada, kami mengusulkan salah satu rumus yang dapat menyelesaikan permasalahan yang ada, yakni dengan konsep pendekatan deviasi total. Tujuan dari penggunaan deviasi total sendiri ialah untuk menghitung total keseluruhan selisih dari setiap baris, kolom, tiang, diagonal ruang, dan diagonal bidang dari *magic number* ( $M$ ) yang dimiliki oleh *diagonal magic cube* berukuran  $5 \times 5 \times 5$  tersebut ( $M = 315$ ).

$$M = \frac{n(n^3 + 1)}{2} = \frac{5(5^3 + 1)}{2} = \frac{5(125 + 1)}{2} = \frac{5(126)}{2} = 315$$

Rumus deviasi total juga akan berperan sebagai *heuristic* dalam mencari solusi yang optimum dimana dengan mengetahui jumlah selisih setiap baris, kolom, tiang, dan diagonal yang ada pada *magic cube* tersebut dimana hasil total deviasi yang ada akan menunjukkan nilai selisih secara keseluruhan dari hasil deviasi yang ada sehingga didapatkan solusi yang bersifat *global optimum* juga. Semakin kecil total deviasi yang dihasilkan, maka akan semakin sempurna solusi yang akan dihasilkan oleh *diagonal magic cube* tersebut. Bila suatu kubus adalah *diagonal magic cube*, maka total deviasi dari semua baris, kolom, tiang, diagonal ruang, dan diagonal bidang kubus terhadap *magic number* harus sama dengan 0. Sehingga, rumus *diagonal magic cube* dapat ditulis seperti berikut :

$$\Sigma(Xi - M) + \Sigma(Yi - M) + \Sigma(Zi - M) + \Sigma(Ri - M) + \Sigma(Di - M) = 0$$

Keterangan :

- X** adalah jumlah angka pada baris.
- Y** adalah jumlah angka pada kolom.
- Z** adalah jumlah angka pada tiang.
- R** adalah jumlah angka pada diagonal bidang.
- D** adalah jumlah angka pada diagonal ruang.
- M** adalah *magic number*.

Berdasarkan rumus diatas, semakin mendekati 0 nilai fungsinya semakin mendekati solusi *diagonal magic cube*. Dengan mengkuadratkan rumus tersebut, kami dapat meningkatkan kesensitifan terhadap kesalahan. Semakin besar deviasi semakin besar penalti yang dapat mendorong perbaikan lebih cepat dan efektif pada kesalahan-kesalahan tersebut.

Dengan ini maka kita dapatkan *objective function* berbentuk *Sum of Squared Differences* sebagai berikut:

$$f(x) = \Sigma(Xi - M)^2 + \Sigma(Yi - M)^2 + \Sigma(Zi - M)^2 + \Sigma(Ri - M)^2 + \Sigma(Di - M)^2$$

Dengan menggunakan *Sum of Squared Differences* (SSD) sebagai *objective function* kita mendapatkan beberapa kelebihan sebagai berikut:

1. **Penalti yang lebih besar untuk deviasi yang lebih besar:** seperti yang telah disebutkan sebelumnya, dengan SSD penalti yang diberikan pada deviasi akan jauh lebih besar (terutama pada kesalahan besar). Akibatnya, algoritma akan didorong untuk mengoreksi deviasi terbesar terlebih dahulu, yang dapat mempercepat mencapai solusi.
2. **Fungsi yang Halus dan Diferensiabel:** sifat SSD yang halus dan diferensiabel memudahkan algoritma berbasis gradien dalam menjelajahi ruang solusi tanpa terjebak pada titik-titik non-diferensiabel yang dapat menghambat proses optimisasi. Selain itu, kehalusan fungsi dapat membantu metode *Stochastic* seperti *simulated annealing* dan *genetic algorithm* dengan menyediakan lanskap optimisasi yang lebih teratur, sehingga memfasilitasi eksplorasi yang lebih efisien dan terarah.

Berdasarkan riset yang kami temukan, menurut Schroepel 2003, Boyer 2003:

- Tengah dari *Diagonal Magic Cube* memiliki nilai konsisten 63
- Metode yang digunakan Trump dan Boyer melibatkan *auxiliary cube* berorde 3 yang bersifat simetris sentral. Dalam kubus bantu ini, 13 garis yang mencakup nilai tengah memenuhi identitas  $x + y + 63 = 189$ .

Sehingga *Objective Function* akhir kami menjadi:

$$f(x) = \Sigma(Xi - M)^2 + \Sigma(Yi - M)^2 + \Sigma(Zi - M)^2 + \Sigma(Ri - M)^2 + \Sigma(Di - M)^2 + \Sigma(X'i - 189)^2 + \Sigma(Y'i - 189)^2 + \Sigma(Z'i - 189)^2 + \Sigma(D'i - 189)^2 + (CV - 63)^2$$

Keterangan :

**X'** adalah jumlah baris dari *auxiliary cube*.

**Y'** adalah jumlah kolom dari *auxiliary cube*.

**Z'** adalah jumlah tiang dari *auxiliary cube*.

**R'** adalah jumlah diagonal ruang dari *auxiliary cube*.

**CV** adalah central value atau nilai tengah kubus.

## B. Implementasi Algoritma Local Search

Berikut merupakan hasil implementasi algoritma local search berdasarkan pemilihan *objective function* yang akan diterapkan menggunakan bahasa pemrograman *Go*.

### 1. Hill-Climbing Search

#### 1) Steepest Ascent Hill-Climbing

```
func SteepestAscentHillClimb(cube [5][5][5]int) types.AlgorithmResult {  
  
    // initialize results  
    results := types.AlgorithmResult{  
        Algorithm: "Steepest Ascent Hill Climb",  
        InitialCube: cube,  
        InitialOF: objectiveFunction.OF(cube),  
        States: make([]types.IterationState, 0),  
    }  
  
    // record initial state  
    results.States = append(results.States, types.IterationState{  
        Iteration: 0,  
        Cube: cube,  
        OF: objectiveFunction.OF(cube),  
        Action: "Initial",  
    })  
  
    // initialize variables  
    var newcube [5][5][5]int  
    exit := false  
    starttime := time.Now()  
  
    // main loop  
    for !exit {  
        // find best successor  
        newcube = cubeFuncs.FindBestSuccessor(cube)  
  
        // exit if new successor is not better than current  
        if objectiveFunction.OF(newcube) >= objectiveFunction.OF(cube)  
        {  
            exit = true  
        }  
  
        // move to new successor  
        cube = newcube  
  
        // record state  
        results.States = append(results.States, types.IterationState{  
            Iteration: len(results.States),  
            Cube: cube,  
            OF: objectiveFunction.OF(cube),  
            Action: "Move",  
        })  
    }  
  
    // record final state  
    results.States = append(results.States, types.IterationState{  
        Iteration: len(results.States),  
        Cube: cube,  
    })  
}
```

```

        OF:          objectiveFunction.OF(cube),
        Action:      "Final state",
    })

    results.FinalCube = cube
    results.FinalOF = objectiveFunction.OF(cube)
    results.Duration = time.Since(starttime)

    return results
}

```

Penerapan algoritma *Steepest Ascent Hill-Climbing Search* dilakukan dengan menggunakan fungsi *class SteepestAscentHillClimb()*. Inisialisasi dilakukan dengan membuat *results* menciptakan variabel *results* sebagai objek dari **types.AlgorithmResult**. Objek tersebut akan menyimpan informasi tentang hasil akhir algoritma, baik berupa nama dari algoritma, *state* awal dari *cube* tersebut, nilai awal dari *Objective Function*, serta urutan perubahan kondisi *cube* pada setiap iterasi tersebut. Hasil dari *state* awal *cube* akan disimpan ke dalam *results* dan akan menyimpan nilai *objective function* yang diperoleh dari fungsi **objectiveFunction.OF(cube)** dengan *action* yang dipanggil berupa “*Initial*”. Kemudian, **exit** akan diinisialisasikan menjadi bernilai “*false*” dengan tujuan untuk menghentikan proses awal iterasi. Variabel **starttime** berperan dalam menyimpan waktu mulai dari setiap eksekusi yang akan dilakukan.

Selanjutnya, algoritma *local search* akan melakukan proses *loop*, dimana algoritma akan melakukan proses *looping* hingga variabel **exit** terinisialisasi menjadi “*true*” yang berarti algoritma akan terus melakukan pengulangan hingga menemukan *state successor* yang bernilai paling optimal. Langkah pertama dalam proses pengulangan tersebut ialah untuk menemukan *state successor* paling optimal dimana fungsi **cubeFuncs.FindBestSuccessor(cube)** berperan dalam mencari *successor* terbaik pada kondisi awal, yakni kondisi yang memiliki *objective function* paling terbaik dari semua probabilitas *neighbor* yang ada. Apabila ditemukan *objective function* dari *successor neighbor* terbaik (**newcube**) bernilai tidak lebih kecil dari *objective function* dari *state* awal (**cube**), maka algoritma *local search* akan melakukan pemberhentian terhadap pencarian *successor* dengan menginisialisasi variabel **exit** menjadi bernilai “*true*”. Setelah *neighbor* dengan value *successor* terbaik ditemukan, maka algoritma *local search* akan mengubah kondisi *state* awal *cube* menjadi *state neighbor cube* yang baru (**cube = newcube**). Algoritma *local search* akan mencatat perubahan kondisi *cube* dengan menyimpan *value cube* terbaru, serta nilai *objective function*-nya.

Setelah proses pencarian *successor* berhenti, algoritma *local search* akan mencatat *state* akhir dari *cube*, beserta dengan *action* “*Final state*” ke dalam variabel **results** yang kemudian akan dikembalikan sebagai nilai hasil akhir dari *cube* tersebut.

## 2) *Hill-Climbing with Sideways Move*

```
func HillClimbWithSidewaysMoves (cube [5][5][5]int, maxSidewaysMoves int)
types.AlgorithmResult {

    // initialize results
    results := types.AlgorithmResult{
        Algorithm:    "Hill Climb with Sideways Moves",
        InitialCube:  cube,
        InitialOF:    objectiveFunction.OF(cube),
        CustomVar:    maxSidewaysMoves,
        States:       make([]types.IterationState, 0),
    }

    // record initial state
    results.States = append(results.States, types.IterationState{
        Iteration: 0,
        Cube:      cube,
        OF:        objectiveFunction.OF(cube),
        Action:    "Initial",
    })

    // initialize variables
    var newcube [5][5][5]int
    sidewaysCount := 0
    exit := false
    starttime := time.Now()

    // main loop
    for !exit {

        // find best successor
        newcube = cubeFuncs.FindBestSuccessor(cube)

        // check if new successor is better than current
        if objectiveFunction.OF(newcube) > objectiveFunction.OF(cube) {

            exit = true // exit if not

        } else if objectiveFunction.OF(newcube) ==
objectiveFunction.OF(cube) && sidewaysCount < maxSidewaysMoves { // sideways
move
            sidewaysCount++
            cube = newcube

            // record state
            results.States = append(results.States,
types.IterationState{
                Iteration: len(results.States),
                Cube:      cube,
                OF:        objectiveFunction.OF(cube),
                Action:    "Sideways Move",
            })
        }
    }
}
```

```

        } else if objectiveFunction.OF(newcube) <
objectiveFunction.OF(cube) { // move to new successor
    sidewaysCount = 0
    cube = newcube

    // record state
    results.States = append(results.States,
types.IterationState{
    Iteration: len(results.States),
    Cube:      cube,
    OF:        objectiveFunction.OF(cube),
    Action:    "Move",
})
}

} else {
    exit = true
}
}

// record final state
results.States = append(results.States, types.IterationState{
    Iteration: len(results.States),
    Cube:      cube,
    OF:        objectiveFunction.OF(cube),
    Action:    "Final state",
})
results.FinalCube = cube
results.FinalOF = objectiveFunction.OF(cube)
results.Duration = time.Since(starttime)

return results
}
}

```

Penerapan algoritma ***Hill-Climbing Search with Sideways Move*** dilakukan dengan menerapkan kelas fungsi **HillClimbWithSidewaysMoves ()**. Pada dasarnya, algoritma *Hill-Climbing with Sideways Move* merupakan modifikasi dari algoritma *Hill-Climbing* yang memungkinkan adanya gerakan *sideways* (berpindah ke *state neighbor* yang memiliki nilai *objective function* yang sama atau lebih baik dari *state awal*) dengan tujuan untuk menghindari kondisi *plateau (local maximum)* pada proses pencarian *successor*.

Pada awal algoritma dilakukan inisialisasi dengan membuat objek **results** dari variabel **types.AlgorithmResult** untuk menyimpan hasil akhir dari *state cube* tersebut, *state awal cube* (**cube**), nilai *objective function* awal (**objectiveFunction.OF (cube)**), serta jumlah maksimum gerakan *sideways* (**maxSidewaysMoves**), serta juga akan menyimpan urutan terhadap perubahan kondisi *cube* untuk setiap iterasi yang akan dilakukan (**make ([] types.IterationState, 0)**). Pada *state awal*, algoritma *local search* akan mencatat kondisi awal dari *cube* ke dalam

`results`, termasuk nilai *objective function* dari *cube* awal, dengan menyimpan nilai *action* yang ditandai dengan *value* “Initial”. Kemudian, baru akan dilakukan inisialisasi terhadap jenis variabel lain, seperti variabel `sidewaysCount` yang akan diinisialisasikan dengan nilai 0 dengan tujuan untuk mengkalkulasi jumlah gerakan *sideways* yang dapat terjadi, sedangkan variabel `exit` (*boolean*) akan diinisialisasi dengan nilai “*false*” dengan tujuan untuk mengontrol kondisi *loop* dari awal hingga mencapai proses terminasi. Variabel `starttime` berperan untuk menyimpan waktu dari awal algoritma *local search* melakukan pencarian *successor* hingga proses terminasi atau pemberhentian dilakukan.

Selanjutnya, algoritma *local search* akan mulai melakukan proses *looping* untuk mencari *successor neighbor* terbaik. Program akan mencari *neighbor* yang memiliki *value* paling optimal dari *state* awal *cube* saat ini menggunakan *function* `cubeFuncs.FindBestSuccessor(cube)`. Jika selama proses pencarian ditemukan *neighbor* dengan *value* yang lebih baik atau lebih besar dibandingkan dengan *value* *state* awal *cube*, maka algoritma akan berpindah dari *state* awal *cube* ke *state neighbor* yang memiliki *value* baru tersebut (`cube = newcube`), dimana gerakan tersebut ditandai sebagai *action* “`Move`”.

Jika nilai *objective function* dari *neighbor* bernilai sama dengan *state* awal *cube*, maka algoritma tetap akan melakukan gerakan *sideways* menuju *neighbor* tersebut, apabila jumlah *sideways count* belum mencapai maksimum (`maxSidewaysMoves`). Apabila algoritma *local search* melakukan *sideways move*, maka `sidewaysCount` akan bertambah satu dan algoritma akan mencatat *state* terbaru tersebut dengan *action* “`Sideways Move`”. Namun, jika selama proses pencarian *successor* didapatkan *neighbor* yang tidak memiliki *value* lebih baik dari *state* awal *cube*, maka `sidewaysCount` akan tetap mengembalikan nilai 0 pada saat melakukan proses terminasi tersebut. Algoritma kemudian akan memindahkan *state neighbor* tersebut dan menyimpan *state* terbaru dengan *action* “`Move`”.

Jika tidak ada kondisi *neighbor* yang lebih baik ataupun sudah tidak ada gerakan *sideways move* yang dapat dilakukan karena telah mencapai batas maksimum, maka algoritma *local search* akan mengakhiri proses *loop* atau pencarian *successor* tersebut dengan menginisialisasi variabel `exit` menjadi “*true*”. Setelah proses pencarian telah selesai, program akan menyimpan kondisi final dari *cube*, beserta dengan *objective function*-nya, dan memmenginisialisasi *action* “*Final state*” ke dalam variabel `results`. Algoritma *local search* juga akan menghitung durasi dari waktu awal proses pencarian hingga proses

terminasi dan menyimpan proses pencarian solusi tersebut ke dalam **results** yang kemudian akan dikembalikan sebagai hasil akhir dari *cube* tersebut.

### 3) Stochastic Hill-Climbing Search

```
func StochasticHillClimb(cube [5][5][5]int, amount int)
types.AlgorithmResult {

    // initialize results
    results := types.AlgorithmResult{
        Algorithm: "Stochastic Hill Climb",
        InitialCube: cube,
        InitialOF: objectiveFunction.OF(cube),
        CustomVar: amount,
        States: make([]types.IterationState, 0),
    }

    // record initial state
    results.States = append(results.States, types.IterationState{
        Iteration: 0,
        Cube: cube,
        OF: objectiveFunction.OF(cube),
        Action: "Initial",
    })

    // initialize variables
    var newcube [5][5][5]int
    nomoves := 0
    maxnomoves := 2000
    lastIter := 20000
    starttime := time.Now()

    // main loop
    for i := 1; i < amount+1; i++ { // repeat n times
        newcube = cubeFuncs.FindSuccessor(cube)
        if objectiveFunction.OF(newcube) < objectiveFunction.OF(cube) {
            cube = newcube

            // record state
            results.States = append(results.States,
types.IterationState{
                Iteration: i,
                Cube: cube,
                OF: objectiveFunction.OF(cube),
                Action: "Move",
            })
        }

        nomoves = 0
    } else {
        nomoves++
        if nomoves == maxnomoves {
            break
        }
    }
    lastIter = i
}
```

```

    // record final state
    results.States = append(results.States, types.IterationState{
        Iteration: lastIter,
        Cube:      cube,
        OF:        objectiveFunction.OF(cube),
        Action:    "Final state",
    })

    results.FinalCube = cube
    results.FinalOF = objectiveFunction.OF(cube)
    results.Duration = time.Since(starttime)

    return results
}

```

Penerapan Algoritma *Stochastic Hill-Climbing Search* dilakukan dengan menerapkan kelas fungsi **StochasticHillClimb()**. Algoritma *Stochastic Hill-Climbing Search* bersifat stokastik yang berarti algoritma tidak akan selalu mencari solusi terbaik pada setiap pengecekan *successor*, melainkan memilih *successor* secara *random* berdasarkan *state neighbor* tersebut.

Inisialisasi dilakukan dengan membuat variabel **results** dari **types.AlgorithmResult** dengan tujuan untuk menyimpan nilai hasil dari eksekusi yang terdiri dari kondisi awal *cube* (**cube**), nilai *objective function* awal (**objectiveFunction.OF(cube)**), jumlah iterasi (**amount**), serta menyimpan urutan perubahan kondisi *cube* tersebut. *State* awal *cube* akan tersimpan pada variabel **results** bersamaan dengan *objective function*-nya yang ditandai dengan *action* “Initial”. Variabel **nomoves** akan diinisialisasi sebagai variabel yang menghitung jumlah pengecekan *successor* secara berturut-turut dimana tidak ada perbaikan yang ditemukan, sedangkan variabel **maxnomoves** akan diinisialisasikan dengan nilai 10.000 sebagai batas proses pencarian *successor* oleh algoritma *local search* apabila proses pencarian dilakukan terlalu lama.

Selanjutnya, algoritma *local search* akan melakukan proses *looping* sebanyak jumlah iterasi yang telah diinisialisasi sebelumnya pada variabel **amount**, dimana pada setiap iterasi yang akan dilakukan, algoritma *local search* akan membangkitkan *successor neighbor* secara acak menggunakan *function* **cubeFuncs.FindSuccessor(cube)**. Apabila *successor* bernilai lebih baik daripada *state* awal *cube*, maka algoritma *local search* akan mengubah *state* awal *cube* menjadi *state neighbor* tersebut (**cube = newcube**), lalu menyimpan perubahan tersebut ke dalam variabel **results** dengan *action* “Move” dan melakukan

inisialisasi pada variabel **nomoves** menjadi bernilai 0. Apabila nilai *successor* yang ditemukan tidak lebih optimal dibandingkan *state cube* pada saat itu, maka algoritma *local search* akan menambah nilai pada **nomoves** untuk menghitung jumlah iterasi.

Ketika jumlah iterasi telah mencapai batas maksimum (**maxnomoves**), maka algoritma akan melakukan pemberhentian terhadap proses pencarian *successor* secara langsung dengan tujuan untuk mencegah durasi pengulangan yang tidak efisien. Setelah proses iterasi atau pengecekan *successor* selesai dilakukan, maka algoritma *local search* akan menyimpan *state akhir cube*, nilai *objective function*, serta action “*Final state*” ke dalam variabel **results**.

#### 4) Random Restart Hill-Climbing Search

```
func RandomRestartHillClimb(cube [5][5][5]int, amount int)
types.AlgorithmResult {
    // initialize results
    results := types.AlgorithmResult{
        Algorithm: "Random Restart Hill Climb",
        InitialCube: cube,
        InitialOF: objectiveFunction.OF(cube),
        CustomVar: amount,
        States: make([]types.IterationState, 0),
        CustomArr: make([]int, amount),
    }

    // record initial state
    results.States = append(results.States, types.IterationState{
        Iteration: 0,
        Cube: cube,
        OF: objectiveFunction.OF(cube),
        Action: "Initial",
    })

    // initialize variables
    var newcube [5][5][5]int
    starttime := time.Now()
    bestcube := cube
    bestcubeOF := objectiveFunction.OF(cube)

    // random restart loop
    for i := 0; i < amount; i++ {
        // initialize variables
        exit := false

        if i > 0 {
            cube = cubeFuncs.RandomizeCube(cube)
            // Record restart state
            results.States = append(results.States,
types.IterationState{
            Iteration: len(results.States),
            Cube: cube,
        })
    }
}
```

```

        OF:      objectiveFunction.OF(cube),
        Action:  "restart",
    })
}

// main loop (modified steepest ascent hill climb)
for !exit {

    // find best successor
    newcube = cubeFuncs.FindBestSuccessor(cube)

    // exit if new successor is not better than current
    if objectiveFunction.OF(newcube) >=
objectiveFunction.OF(cube) {
        exit = true

        // record iteration count for each restart
        results.CustomArr[i] = len(results.States)
    }

    cube = newcube // move to new successor

    // record state
    results.States = append(results.States,
types.IterationState{
    Iteration: len(results.States),
    Cube:      cube,
    OF:        objectiveFunction.OF(cube),
    Action:    "Move",
})
}

if objectiveFunction.OF(cube) < bestcubeOF { // finds best
cube
    bestcube = cube
    bestcubeOF = objectiveFunction.OF(cube)
}

// record final restart state
results.States = append(results.States, types.IterationState{
    Iteration: len(results.States),
    Cube:      cube,
    OF:        objectiveFunction.OF(cube),
    Action:    "Final state",
})
}

// record final state
results.FinalCube = bestcube // returns best cubeq
results.FinalOF = bestcubeOF
results.Duration = time.Since(starttime)

return results
}

```

Penerapan algoritma ***Random Restart Hill Climbing Search*** dilakukan dengan menerapkan fungsi kelas **RandomRestartHillClimb()**. Pada dasarnya, algoritma ini

bertujuan untuk menghindari jebakan *local maximum* pada saat proses iterasi pencarian *successor* terbaik, dimana algoritma *local search* akan melakukan *restart* atau mengulangi proses pengecekan *successor* dari awal ketika algoritma local search terjebak pada kondisi *local maximum* tersebut.

Inisialisasi dilakukan dengan membuat variabel **results** dari **types.AlgorithmResult** dengan tujuan untuk menyimpan nilai hasil dari eksekusi yang terdiri dari kondisi awal *cube* (**cube**), nilai *objective function* awal (**objectiveFunction.OF(cube)**), jumlah iterasi (**amount**), serta menyimpan urutan perubahan kondisi *cube* tersebut. *State* awal *cube* akan tersimpan pada variabel **results** bersamaan dengan *objective function*-nya yang ditandai dengan *action* “Initial”. Variabel **bestucube** dan **bestcubeOf** akan berperan dalam menyimpan *state neighbor* dengan *value* terbaik yang akan ditemukan selama proses iterasi pengecekan *successor* berlangsung. Selain itu, **starttime** juga akan berperan dalam menyimpan durasi pada waktu mulai algoritma *local search* melakukan iterasi.

Pada saat proses iterasi pengecekan *successor* dilakukan, algoritma *local search* akan melakukan proses *random restart* sebanyak **amount** sesuai dengan batasan yang telah diinisialisasikan sebelumnya. Algoritma *local search* akan membangkitkan *successor* secara *random* melalui *function* **cubeFuncs.RandomizeCube(cube)** yang kemudian akan membuat algoritma melakukan proses pengecekan secara ulang dari *successor* awal ketika proses iterasi dilakukan. Lalu, algoritma *local search* juga akan menyimpan *state cube* yang ada setiap kali proses *restart* tersebut dilakukan yang ditandai dengan *action* “restart”. Pada setiap *restart* yang dilakukan, algoritma akan menjalankan proses pengecekan *successor* secara *Hill Climbing*, dimana algoritma *local search* bertujuan melakukan pemeriksaan pada setiap *neighbor* yang ada untuk menemukan nilai *objective function* yang lebih baik dari *state* awal pada *cube* tersebut. Sama seperti dengan konsep *Steepest Ascent Hill Climbing*, algoritma *local search* akan mencari *successor* yang paling optimal menggunakan *function* **cubeFuncs.FindBestSuccessor(cube)**. Jika algoritma local search telah menemukan *successor* dengan *value objective function* yang lebih baik dari *objective function* di *state* awal *cube*, maka algoritma akan mengganti kondisi *state* awal *cube* dengan *value successor* yang baru (**cube = newcube**) serta menyimpan perubahan kondisi *value cube* ke dalam variabel **results**. Apabila selama pengecekan *successor*, terdapat *value neighbor* yang tidak lebih baik dari *value cube* di awal, maka algoritma akan melakukan pemberhentian iterasi untuk pengecekan *successor* terbaru (**exit = true**).

Selanjutnya, algoritma akan melakukan perbandingan terhadap setiap *successor* yang telah dibangkitkan, dimana apabila kondisi *cube* yang ditemukan pada saat melakukan proses *random restart* memiliki *value* yang lebih baik dibandingkan dengan *value* yang dimiliki oleh variabel **bestcube**, maka algoritma akan memperbaiki kondisi *state cube* pada variabel **bestcube** dan **bestcubeOf** dengan *value objective function* yang telah ditemukan sebelumnya.

## 2. Simulated Annealing

```
func SimulatedAnnealing(cube [5][5][5]int) types.AlgorithmResult {
    // Schedule function for temperature
    Schedule := func(T float64, iteration int) float64 {
        if iteration < 1000 {
            return T * 0.995 // Faster cooling at start
        } else if iteration < 5000 {
            return T * 0.999
        } else if iteration < 10000 {
            return T * 0.9995
        } else {
            return T * 0.999935
        }
    }

    // initialize results
    results := types.AlgorithmResult{
        Algorithm:    "Simulated Annealing",
        InitialCube:  cube,
        InitialOF:    objectiveFunction.OF(cube),
        States:       make([]types.IterationState, 0),
    }

    T := rand.Float64() * 3000000000.0 // initial temperature

    // record initial state
    results.States = append(results.States, types.IterationState{
        Iteration:    0,
        Cube:         cube,
        OF:           objectiveFunction.OF(cube),
        Action:       "Initial",
        Temperature: T,
    })

    // initialize variables
    var newcube [5][5][5]int
    exit := false
    i := 1
    starttime := time.Now()

    for !exit {
        T = Schedule(T, i) // Update temperature
        i++
        if T < 0.0005 { // Exit if temperature is too low

```

```

        exit = true
    }

    newcube = cubeFuncs.FindSuccessor(cube)
    deltaE := objectiveFunction.OF(newcube) -
objectiveFunction.OF(cube)

    prob := math.Exp(-float64(deltaE) / T)

    // Check if new cube has lower objective function value
    if deltaE < 0 {
        cube = newcube

        // avoid prob > 1 (avoid infinite numbers)
        if prob > 1 {
            prob = 1
        }

        // record state
        results.States = append(results.States,
types.IterationState{
            Iteration: i,
            Cube: cube,
            OF: objectiveFunction.OF(cube),
            Action: "Move",
            Temperature: T,
            Prob: float32(prob),
        })
        results.CustomVar++

    } else {

        // Check if new cube has higher objective function value
        if rand.Float64() < prob {
            cube = newcube

            // record state
            results.States = append(results.States,
types.IterationState{
                Iteration: i,
                Cube: cube,
                OF: objectiveFunction.OF(cube),
                Action: "Backward Move",
                Temperature: T,
                Prob: float32(prob),
            })
        }

    }

    // record final state
    results.States = append(results.States, types.IterationState{
        Iteration: i,
        Cube: cube,
        OF: objectiveFunction.OF(cube),
        Action: "Final State",
        Temperature: T,
    })
}

```

```

    results.FinalCube = cube
    results.FinalOF = objectiveFunction.OF(cube)
    results.Duration = time.Since(starttime)

    return results
}

```

Algoritma **Simulated Annealing** diterapkan dengan fungsi `SimulatedAnnealing`. Algoritma dibuat agar program dapat mencapai *global maximum* dan tidak terjebak di *local maximum* dengan membuat algoritma dapat mengambil *successor* dengan *heuristic value* yang lebih buruk daripada *current state* sebagai *next state* selama *probability*  $e^{\Delta E/T}$  lebih besar dari *random float number*.

Algoritma dimulai dengan membuat *schedule*, *schedule* ini mengatur kecepatan temperatur menurun. Ketika iterasi telah melewati batas - batas tertentu yang telah ditetapkan kecepatan temperatur menurun akan dikurangi. Hal ini akan mempengaruhi waktu berhenti algoritma dan penilaian probabilitas.

Kemudian dilakukan inisialisasi variabel `results`. Variabel ini merupakan struktur `AlgorithmResult` yang berfungsi untuk menyimpan hasil akhir dari eksekusi program yang terdiri dari beberapa variabel . Pada fase inisialisasi, variabel yang menyimpan nilai inisial, seperti `Algorithm`, `InitialCube`, `InitialOF`, `States` diberi nilai. `Algorithm` berupa nama simulasi yang dilakukan. `InitialCube` bernilai kubus input dari fungsi. `InitialOF` berupa nilai *objective function* dari `InitialCube`. `States` diinisialisasi dengan slice kosong yang berisikan tipe struktur `IterationState`.

Setelah menginisialisasi `results`, algoritma juga menginisialisasi variabel temperatur (`T`). Setelah itu algoritma merecord *state* awal dari fungsi. Inisialisasi terakhir dilakukan dengan menginisialisasi beberapa variabel, yaitu `newcube`, `exit`, `i`, `starttime`. `newcube` pada inisialisasi baru dideklarasikan sebagai kubus (matriks *integer*). `exit` berupa boolean yang bernilai `false` untuk menentukan kapan fungsi akan berhenti. `i` berupa integer bernilai satu yang menjadi angka dari jumlah *loop* yang dilakukan fungsi. `starttime` adalah waktu mulai fungsi yang bernilai waktu sekarang (didapat dari fungsi `time.Now()`).

Fungsi memulai *loop* yang akan terus berlanjut selama `exit` masih bernilai *false*. `exit` akan bernilai *true* jika `T` bernilai dibawah dari nilai yang ditetapkan. Variabel `T` selalu *di-update* tiap memulai loop dengan fungsi `Schedule`. Setelah mengecek nilai dari `exit` algoritma *loop* ini kemudian mendeklarasi variabel `newcube` sebagai *successor*, `deltaE` yang merupakan nilai dari *objective function* `newcube` dikurangi *objective function* `cube`, dan `prob` yang merupakan probabilitas ( $e^{\Delta E/T}$ ).

Jika `deltaE` lebih kecil dari 0, berarti nilai *objective function* `newcube` mendekati nilai 0 yaitu solusinya. Maka, `cube` diubah menjadi `newcube` dan jika nilai `prob` diatas 1 maka diubah nilainya menjadi 1. Setelah itu, *state* iterasi ini ditambahkan kedalam variabel `States` sebagai *record* yang berada dalam `results`.

Jika `deltaE` lebih besar dari 0, berarti nilai *objective function* `newcube` lebih jauh dari nilai *objective function* `cube` yang berarti `newcube` lebih jauh dari solusi daripada `cube`. Algoritma akan mengecek apakah probabilitas successornya melebihi *random float*. Jika iya, maka `cube` diubah menjadi `newcube`, lalu *state* iterasi ini ditambahkan kedalam variabel `States` sebagai *record* yang berada dalam `results`. Jika tidak, maka algoritma akan langsung lanjut ke iterasi berikutnya.

Jika *loop* telah selesai maka dilakukan *record state* yang terakhir kali dan variabel `FinalCube`, `FinalOF`, `Duration` yang berada dalam `results` diisi dengan nilai akhir yang didapat. `FinalCube` diisi dengan `cube` terakhir. `FinalOF` diisi dengan nilai *objective function* dari `FinalCube`. `Duration` diisi dengan lama waktu setelah `starttime` yang didapat dengan fungsi `time.Since`. Akhirnya, fungsi `SimulatedAnnealing` mengembalikan `results` sebagai output.

### 3. Genetic Algorithm

```
func GeneticAlgorithm(initialCube [5][5][5]int, populationSize int,
maxGenerations int) types.AlgorithmResult {
    // Initialize result
    results := types.AlgorithmResult{
        Algorithm:    "Genetic Algorithm",
        InitialCube: initialCube,
        InitialOF:   objectiveFunction.OF(initialCube),
        States:       make([]types.IterationState, 0),
        CustomVar:    populationSize,
        CustomArr:    make([]int, 0),
    }

    // Record initial state
```

```

results.States = append(results.States, types.IterationState{
    Iteration: 0,
    Cube:      initialCube,
    OF:        objectiveFunction.OF(initialCube),
    Action:    "Initial",
    Population: populationSize,
})
results.CustomArr = append(results.CustomArr, 0)

// Initialize population
population := make([][5][5]int, populationSize)
population[0] = initialCube
for i := 1; i < populationSize; i++ {
    population[i] = cubeFuncs.FindSuccessor(initialCube)
}

// Track best solution
bestCube := initialCube
bestFitness := objectiveFunction.OF(initialCube)

// Initialize time
starttime := time.Now()

// Main loop
for generation := 0; generation < maxGenerations; generation++ {
    // Calculate fitness for all cubes and average
    fitness := make([]int, populationSize)
    totalFitness := 0

    for i := 0; i < populationSize; i++ {
        fitness[i] = objectiveFunction.OF(population[i])
        totalFitness += fitness[i]

        if fitness[i] < bestFitness {
            bestFitness = fitness[i]
            bestCube = population[i]
        }
    }

    // Record average fitness for this generation
    avgFitness := totalFitness / populationSize
    results.CustomArr = append(results.CustomArr, avgFitness)

    // Record best cube state before next generation
    results.States = append(results.States, types.IterationState{
        Iteration: generation + 1,
        Cube:      bestCube,
        OF:        bestFitness,
        Action:    "Progress",
        AvgOF:    float64(avgFitness),
        Population: populationSize,
    })
}

// Create new population
newPopulation := make([][5][5]int, populationSize)
newPopulation[0] = bestCube // Elitism

// Generate new individuals
for i := 1; i < populationSize; i += 2 {
    // Select parents
}

```

```

parent1 := tournamentSelect(population, fitness)
parent2 := tournamentSelect(population, fitness)

// Create children
child1, child2 := crossover(parent1, parent2)

// Mutate children (50% chance)
if rand.Float64() < 0.5 {
    child1 = mutate(child1)
}
if rand.Float64() < 0.5 {
    child2 = mutate(child2)
}

// Add to new population
newPopulation[i] = child1
if i+1 < populationSize {
    newPopulation[i+1] = child2
}
}

population = newPopulation
}

// Record final state
results.States[len(results.States)-1].Action = "Final state"

results.FinalCube = bestCube
results.FinalOF = bestFitness
results.Duration = time.Since(starttime)
results.CustomVar = populationSize // Store population size

return results
}

// Select best cube from random tournament
func tournamentSelect(population [[[5][5][5][5]]int, fitness []int)
[5][5][5]int {
    best := rand.Intn(len(population))
    for i := 0; i < 3; i++ { // Tournament size of 3
        next := rand.Intn(len(population))
        if fitness[next] < fitness[best] {
            best = next
        }
    }
    return population[best]
}

// Swap some random positions between parents
func crossover(parent1, parent2 [5][5][5]int) ([5][5][5]int, [5][5][5]int)
{
    child1 := parent1
    child2 := parent2

    numSwaps := rand.Intn(10) + 5
    for i := 0; i < numSwaps; i++ {
        x := rand.Intn(5)
        y := rand.Intn(5)
        z := rand.Intn(5)
        child1[x][y][z], child2[x][y][z] = child2[x][y][z],
        child2[x][y][z], child1[x][y][z] = child1[x][y][z],
    }
}

```

```

        child1[x][y][z]
    }

    return child1, child2
}

// Swap some random positions in the cube
func mutate(cube [5][5][5]int) [5][5][5]int {
    numSwaps := rand.Intn(3) + 1
    for i := 0; i < numSwaps; i++ {
        x1, y1, z1 := rand.Intn(5), rand.Intn(5), rand.Intn(5)
        x2, y2, z2 := rand.Intn(5), rand.Intn(5), rand.Intn(5)
        cube[x1][y1][z1], cube[x2][y2][z2] = cube[x2][y2][z2],
        cube[x1][y1][z1]
    }
    return cube
}

```

Algoritma Genetik atau *Genetic Algorithm* diterapkan dengan fungsi **GeneticAlgorithm()** di atas. Fungsi **GeneticAlgorithm()** menerima input **initialCube** [5][5][5]int, **populationSize** int, **maxGenerations** int. Fungsi ini melakukan *local search* dengan cara reproduksi dengan implementasi *elitism* dan mutasi. Reproduksi dilakukan dengan seleksi *random* menggunakan fungsi **tournamentSelect()** lalu melakukan crossover dengan fungsi **crossover()**. Mutasi dilakukan menggunakan fungsi **mutate()**.

Inisialisasi pertama dilakukan pada variabel *results* untuk menyimpan hasil eksekusi fungsi yang terdiri dari **initialCube**, **initialOF**, **States**, **CustomVar**, dan **customArr**. **initialCube** diinisialisasi dengan hasil *cube* yang random, dengan **initialOF** adalah Objective Function dari **initialCube** tersebut. **States** berisi data setiap *state* seperti **Iteration**, **Cube**, **OF**, **Action**, **AvgOF**, dan **Population**. Setelah ini, *State* inisial disimpan ke array **States**.

Populasi diinisialisasi dengan membuat array dengan ukuran sesuai **populationSize**, dan value pertama pada array tersebut adalah **initialCube**. Populasi kemudian diisi dengan Successor dari **initialCube** sebanyak **populationSize - 1**.

Main Loop dilakukan setelah variabel **bestCube** diinisialisasi dengan **initialCube**, begitu juga dengan **bestFitness**. Loop dilakukan sebanyak **maxGenerations**. Langkah pertama, array fitness diinisialisasi dengan array kosong sebesar **populationSize**, dan variabel **totalFitness** adalah 0. Kemudian cari **bestFitness** dan **bestCube** menggunakan loop yang dilakukan **populationSize** kali. Setelah itu, hitung fitness rata-rata dengan cara membagi **totalFitness** dengan

`populationSize` dan simpan ke variabel `avgFitness`, `bestCube` disimpan state-nya pada `States` untuk generasi berikutnya. Selanjutnya, tahap reproduksi dengan cara looping sebanyak `populationSize` kali dengan interval 2 karena jumlah parent yang dipilih adalah 2. Kedua parent dipilih menggunakan fungsi `tournamentSelect()`. Setelah kedua parent dipilih, reproduksi dengan fungsi `crossover()`, lalu mutasi kedua child dengan probabilitas 50%. Akhirnya, kedua child dimasukkan ke populasi dengan cara menggantikan parents-nya. Setelah Main Loop berakhir, final state akan disimpan dalam results beserta `Duration` dan `populationSize` akhir.

Fungsi `tournamentSelect()` mengembalikan individu dengan *fitness* terbaik dari populasi. Pertama-tama, variabel `best` diinisialisasi dengan integer random. Lalu, lakukan loop sebanyak 3 kali untuk memilih individu random dari populasi, jika *fitness* dari individu kurang dari *fitness* `best`, maka assign `next` kepada `best`. Setelah loop selesai, kembalikan individu dengan *best fitness*.

Fungsi `crossover()` mengembalikan 2 individu hasil reproduksi 2 parent. Kedua child diinisialisasi dengan masing-masing parent, lalu lakukan loop sebanyak `numSwaps` kali, dengan `numSwaps` adalah random integer (di bawah 10) + 5. Pada loop, tukar *cell random* dari kedua `child` tersebut sebanyak angka *random*. Setelah loop selesai, kembalikan kedua `child`.

Fungsi `mutate()` mengembalikan *cube* yang sudah dimutasi. Pertama lakukan loop sebanyak `numSwaps` kali, dengan `numSwaps` adalah angka random di bawah 3. Pada loop, tukar 2 *cell random* pada *cube*. Setelah loop selesai, kembalikan *cube*.

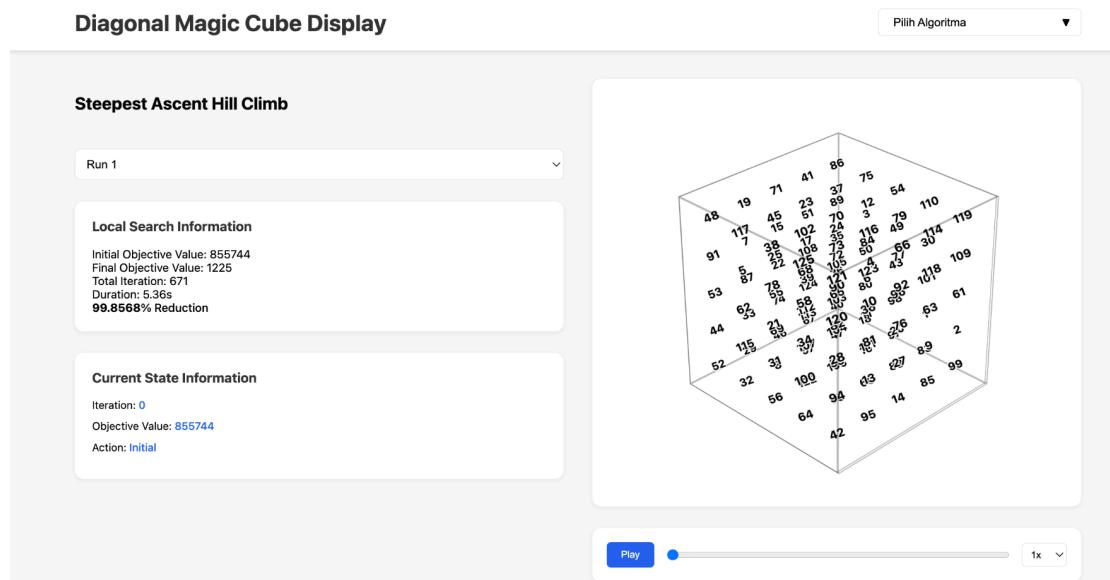
## C. Hasil Eksperimen & Analisis

### 1. Hasil Eksperimen

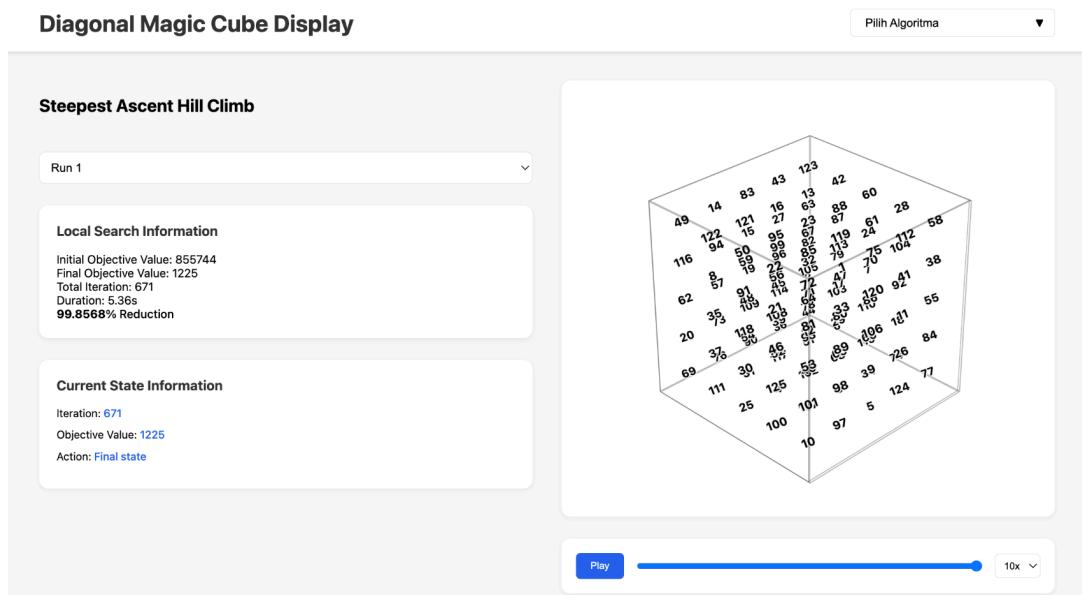
Berikut merupakan hasil eksperimen dari masing-masing tipe implementasi algoritma *local search* untuk menentukan solusi perbandingan nilai *objective function* yang paling optimal dan efisien.

#### 1. Steepest Ascent Hill Climbing Search

- Percobaan Ke - 1

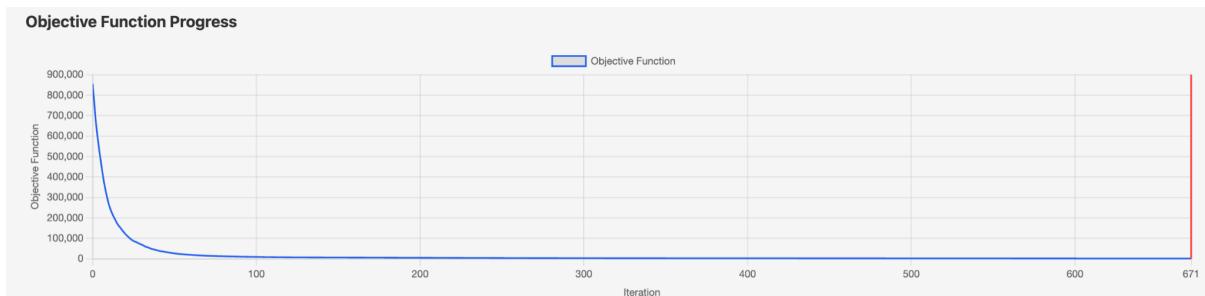


Gambar 1. State Awal Cube Uji Coba Ke - 1 Stochastic Hill Climbing



Gambar 2. Final State Uji Coba Ke - 1 Stochastic Hill Climbing

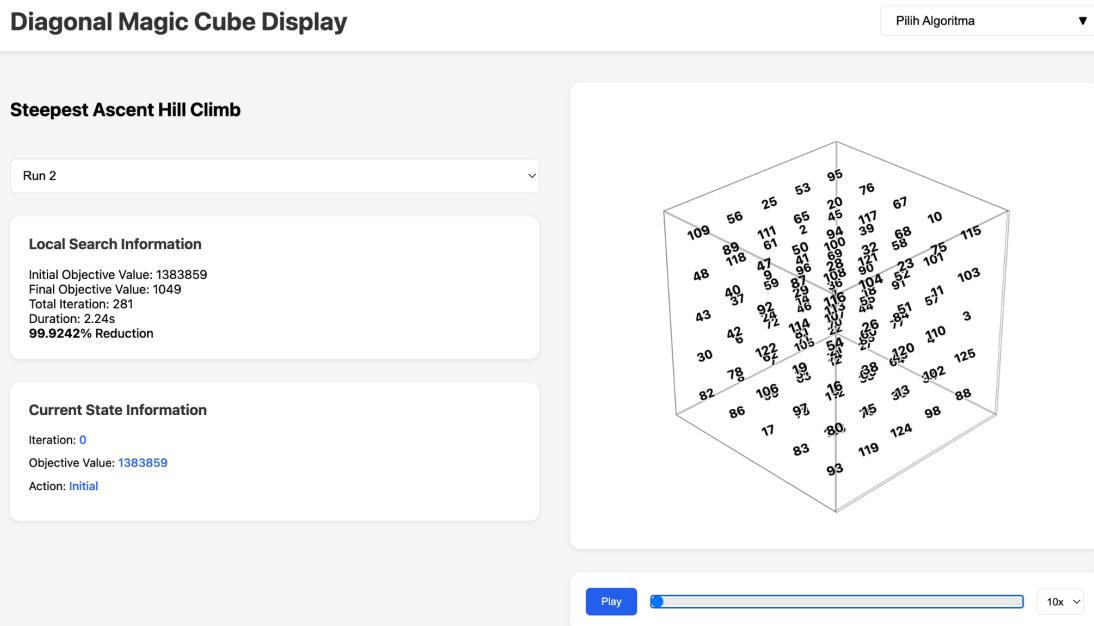
Berdasarkan hasil uji coba pertama terhadap implementasi algoritma **Steepest Ascent Hill-Climbing Search** yang telah dilakukan sesuai dengan gambar di atas, dapat dilihat bahwa terdapat perubahan signifikan yang terjadi pada *state* awal dan *state* akhir *diagonal magic cube* tersebut. Durasi yang digunakan untuk melakukan proses pencarian tersebut ialah berkisar **5.36** detik sesuai dengan yang ditampilkan pada Gambar 1 pada bagian *Local Search Information*. Seperti yang dapat dilihat, *state* awal (iterasi ke-0) pada *diagonal magic cube* tersebut menunjukkan nilai *objective function* berupa **855.744** sebagai *value successor* awal. Sedangkan pada Gambar 2, ditunjukkan nilai dari proses iterasi terakhir, yakni iterasi ke-671, nilai *objective function* yang didapatkan berupa **1.225** yang sekaligus menunjukkan bahwa *value successor* pada *neighbor* tersebut adalah yang paling optimal dibandingkan dengan *successor* lainnya sehingga solusi pada hasil uji coba pertama dapat disimpulkan akan mengambil nilai *final state* yakni berupa 1.225 sebagai nilai *objective function* yang paling optimal tersebut. Berikut merupakan hasil plot nilai *objective function* terhadap banyak iterasi yang telah dilakukan oleh algoritma *local search* selama proses pengecekan *successor*.



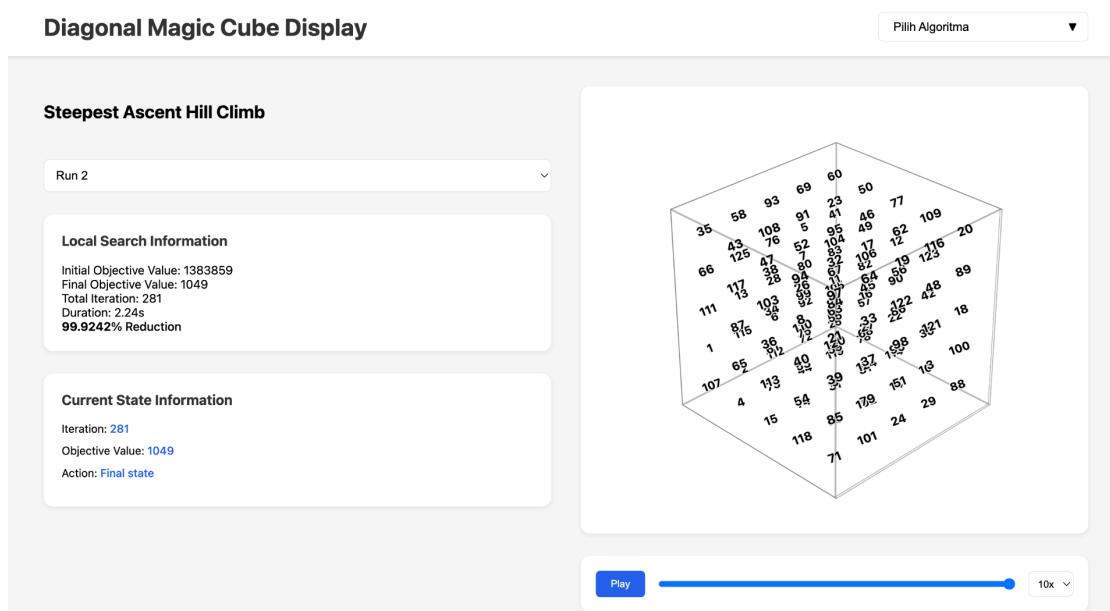
Gambar 3. Plot *Objective Function Steepest Ascent Hill-Climbing* Uji Coba Ke - 1

Dapat dilihat pada grafik tersebut, bahwa hasil perbandingan nilai *objective function* yang didapatkan pada setiap *successor* saat iterasi pengecekan dilakukan oleh algoritma memiliki perbedaan yang cukup signifikan. Dapat dilihat bahwa semakin banyak iterasi yang dilakukan oleh algoritma *local search*, maka nilai *objective function* yang dihasilkan pun akan semakin kecil dibandingkan dengan initial state awal pada *diagonal magic cube* tersebut. Algoritma *Steepest Ascent Hill-Climbing* bertujuan untuk mencari *successor* dengan value yang terbaik dari *neighbor-neighbor* lainnya, sehingga dari hasil uji coba ke-1 yang telah dilakukan dapat dilihat bahwa iterasi terakhir yang dilakukan oleh program yakni iterasi ke-671 mengembalikan nilai *objective function* sebesar **1.225** (*local optimum*).

## ● Percobaan Ke - 2



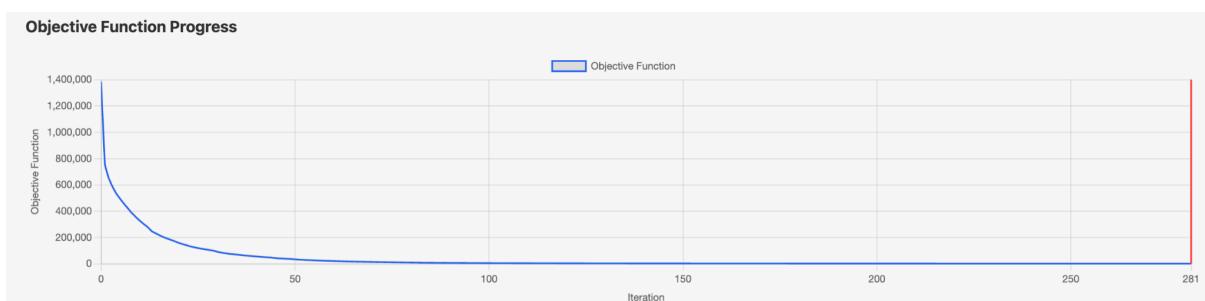
Gambar 4. *State Awal dengan Steepest Ascent Hill-Climbing Pada Uji Coba Ke-2*



Gambar 5. *Final State dengan Steepest Ascent Hill-Climbing Pada Uji Coba Ke-2*

Berdasarkan hasil uji coba kedua terhadap implementasi algoritma **Steepest Ascent Hill-Climbing Search** yang telah dilakukan sesuai dengan gambar di atas, dapat dilihat bahwa terdapat perubahan signifikan yang terjadi pada *state* awal dan *state* akhir *diagonal magic cube* tersebut. Durasi yang digunakan untuk melakukan proses pencarian tersebut ialah berkisar **2.24 detik** sesuai dengan yang ditampilkan pada Gambar 4, dimana durasi yang dimiliki oleh percobaan ke-2 jauh lebih singkat dan efisien dibandingkan uji coba pertama.

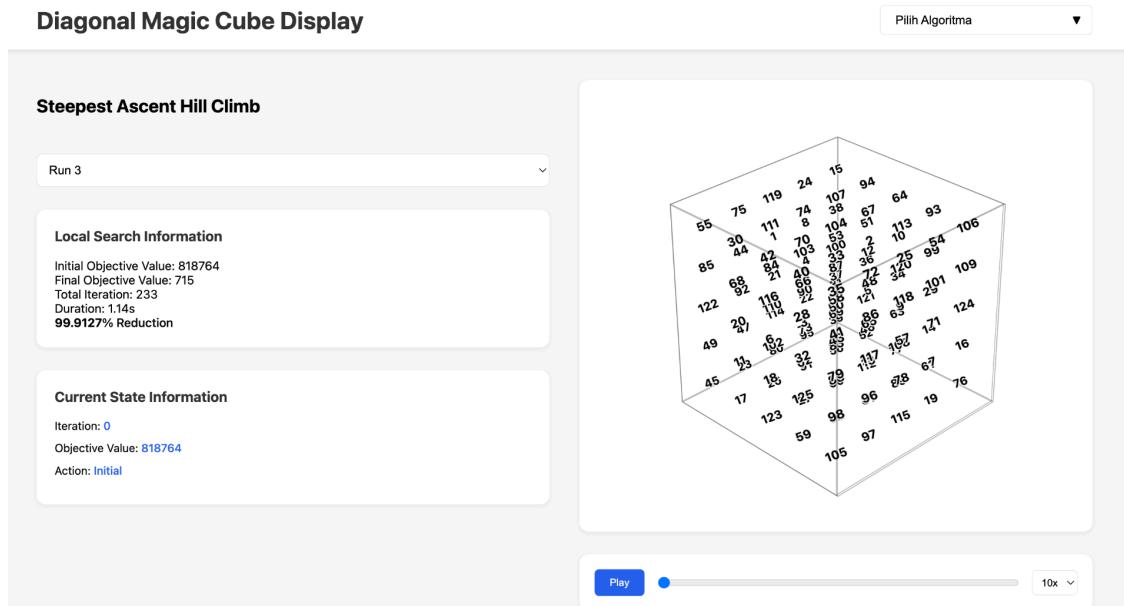
Seperti yang dapat dilihat, *state* awal (iterasi ke-0) pada *diagonal magic cube* tersebut menunjukkan nilai *objective function* berupa **1.383.859** sebagai *value successor* awal. Sedangkan pada Gambar 5, ditunjukkan nilai dari proses iterasi terakhir, yakni iterasi ke-281, nilai *objective function* yang didapatkan berupa **1.049** yang sekaligus menunjukkan bahwa *value successor* pada *neighbor* tersebut adalah yang paling optimal dibandingkan dengan *successor* lainnya sehingga solusi pada hasil uji coba kedua dapat disimpulkan akan mengambil nilai *final state* yakni berupa 1.049 sebagai nilai *objective function* yang paling optimal tersebut. Berikut merupakan hasil plot nilai *objective function* terhadap banyak iterasi yang telah dilakukan oleh algoritma *local search* selama proses pengecekan *successor*.



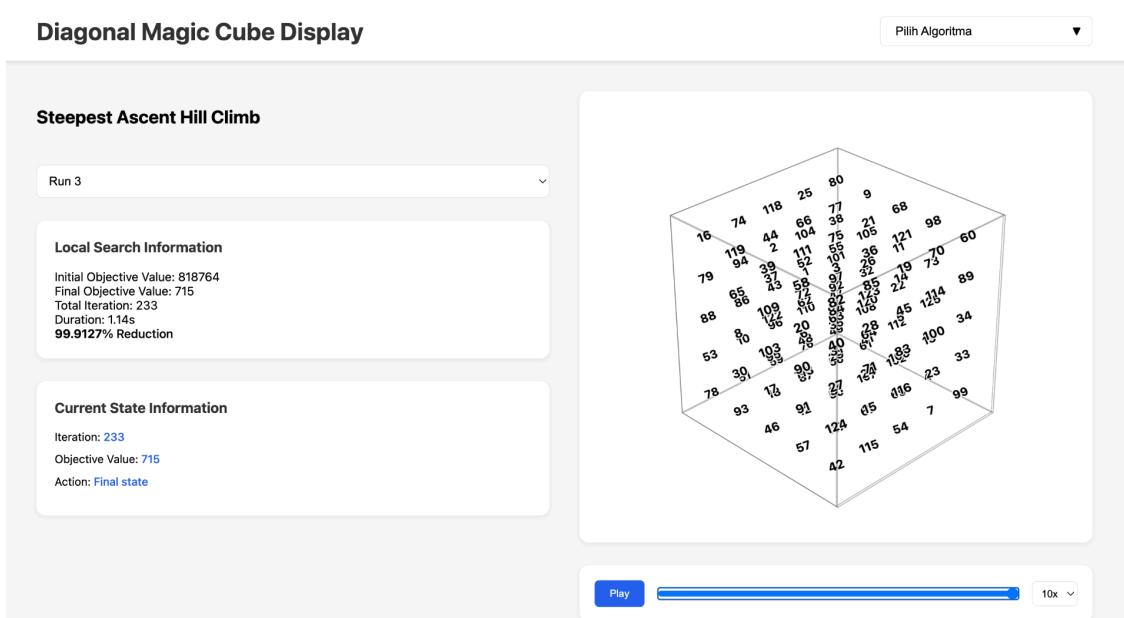
Gambar 6. Plot *Objective Function Steepest Ascent Hill-Climbing* Uji Coba Ke - 2

Dapat dilihat pada grafik tersebut, bahwa hasil perbandingan nilai *objective function* yang didapatkan pada setiap *successor* saat iterasi pengecekan dilakukan oleh algoritma *local search* tidak jauh berbeda dengan uji coba pertama yang telah dilakukan sebelumnya, namun pada grafik yang dihasilkan dari hasil plottingan tersebut tetap memiliki perbedaan yang cukup signifikan. Dapat dilihat bahwa semakin banyak iterasi yang dilakukan oleh algoritma *local search*, maka nilai *objective function* yang dihasilkan pun akan semakin kecil dibandingkan dengan initial state awal pada *diagonal magic cube* tersebut. Algoritma *Steepest Ascent Hill-Climbing* bertujuan untuk mencari *successor* dengan value yang terbaik dari *neighbor-neighbor* lainnya, sehingga dari hasil uji coba ke-2 yang telah dilakukan dapat dilihat bahwa iterasi terakhir yang dilakukan oleh program yakni iterasi ke-281 mengembalikan nilai *objective function* sebesar **1.049** (*local optimum*).

### • Percobaan Ke - 3



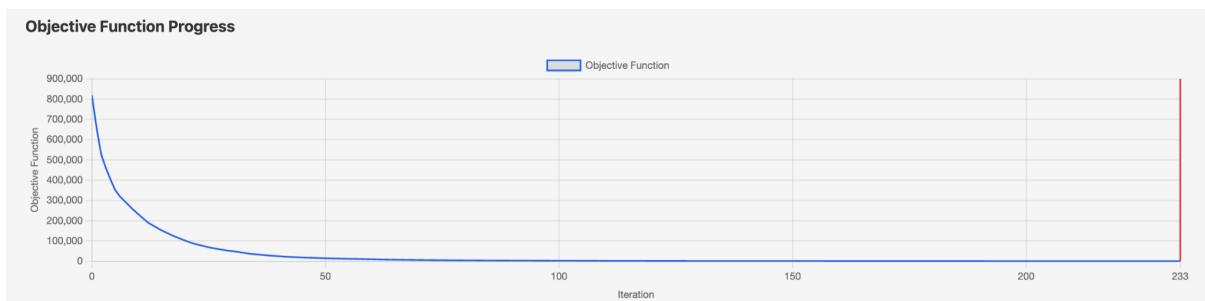
Gambar 7. State Awal Uji Coba Ke - 3 dengan *Steepest Ascent Hill-Climbing*



Gambar 8. Final State Uji Coba Ke - 3 dengan Steepest Ascent Hill-Climbing

Berdasarkan hasil uji coba ketiga terhadap implementasi algoritma *Steepest Ascent Hill-Climbing Search* yang telah dilakukan sesuai dengan gambar di atas, dapat dilihat bahwa terdapat perubahan signifikan yang terjadi pada *state* awal dan *state* akhir *diagonal magic cube* tersebut. Durasi yang digunakan untuk melakukan proses pencarian tersebut ialah berkisar **1.14 second** sesuai dengan yang ditampilkan pada Gambar 7, dimana durasi yang dimiliki oleh percobaan ke-3 berbeda jauh dengan percobaan pertama dan apabila

dibandingkan dengan percobaan kedua durasi yang dimiliki lebih cepat dan efisien. Seperti yang dapat dilihat, *state* awal (iterasi ke-0) pada *diagonal magic cube* tersebut menunjukkan nilai *objective function* berupa **818.764** sebagai *value successor* awal. Sedangkan pada Gambar 8, ditunjukkan nilai dari proses iterasi terakhir, yakni iterasi ke-233, nilai *objective function* yang didapatkan berupa **715** yang sekaligus menunjukkan bahwa *value successor* pada *neighbor* tersebut adalah yang paling optimal dibandingkan dengan *successor* lainnya sehingga solusi pada hasil uji coba ketiga dapat disimpulkan akan mengambil nilai *final state* yakni berupa 715 sebagai nilai *objective function* yang paling optimal tersebut. Berikut merupakan hasil plot nilai *objective function* terhadap banyak iterasi yang telah dilakukan oleh algoritma *local search* selama proses pengecekan *successor*.



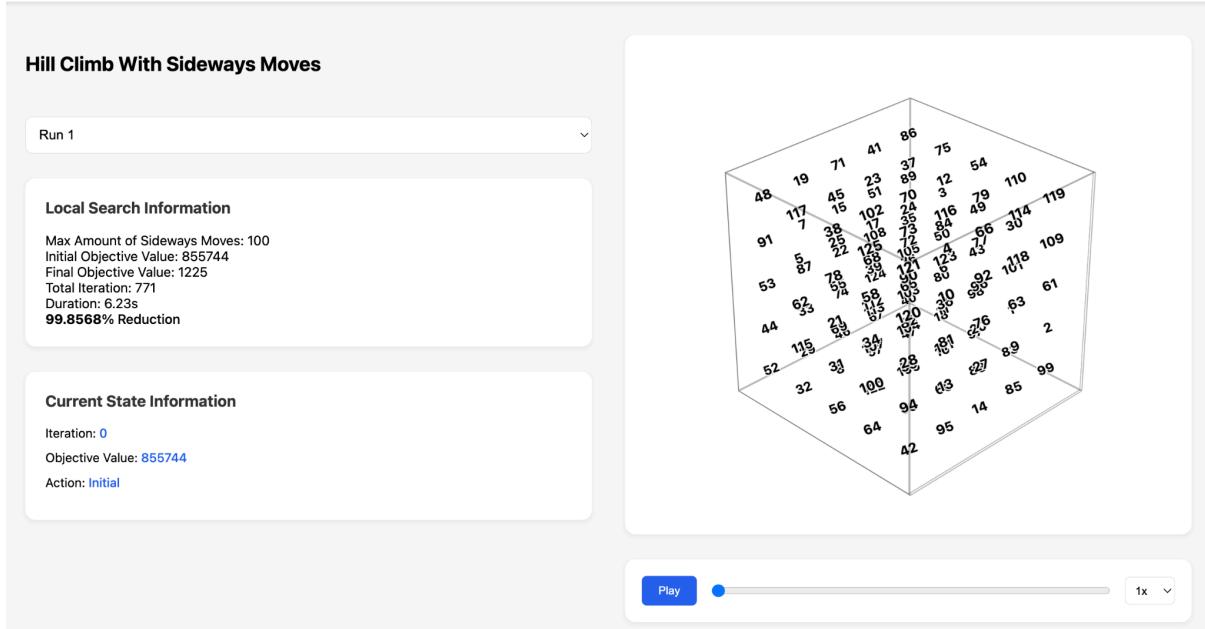
Gambar 9. Plot *Objective Function Steepest Ascent Hill-Climbing* Uji Coba Ke - 3

Dapat dilihat pada grafik tersebut, bahwa hasil perbandingan nilai *objective function* yang didapatkan pada setiap *successor* saat iterasi pengecekan dilakukan oleh algoritma *local search* tidak jauh berbeda dengan uji coba pertama maupun uji coba kedua yang telah dilakukan sebelumnya, namun pada grafik yang dihasilkan dari hasil plotingan tersebut tetap memiliki perbedaan yang cukup signifikan. Dapat dilihat bahwa semakin banyak iterasi yang dilakukan oleh algoritma *local search*, maka nilai *objective function* yang dihasilkan pun akan semakin kecil dibandingkan dengan *initial state* awal pada *diagonal magic cube* tersebut. Algoritma *Steepest Ascent Hill-Climbing* bertujuan untuk mencari *successor* dengan *value* yang terbaik dari *neighbor-neighbor* lainnya, sehingga dari hasil uji coba ke-3 yang telah dilakukan dapat dilihat bahwa iterasi terakhir yang dilakukan oleh program yakni iterasi ke-233 mengembalikan nilai *objective function* sebesar **715** (*local optimum*).

## 2. Hill-Climbing Search with Sideways Move

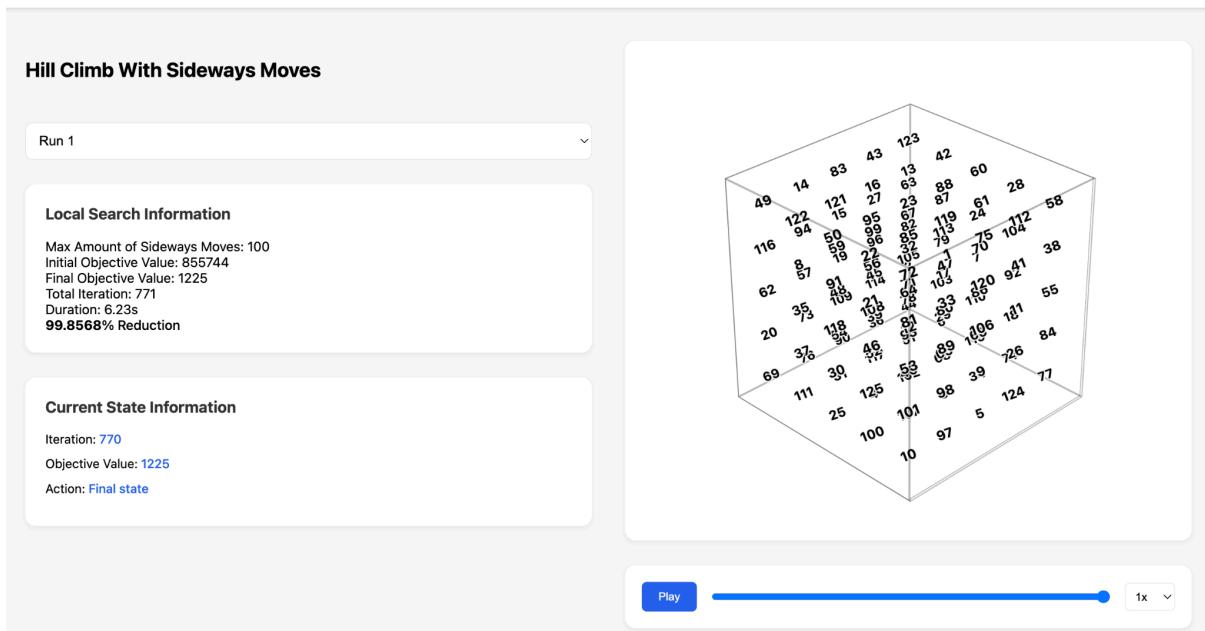
### • Percobaan Ke - 1

#### Diagonal Magic Cube Display



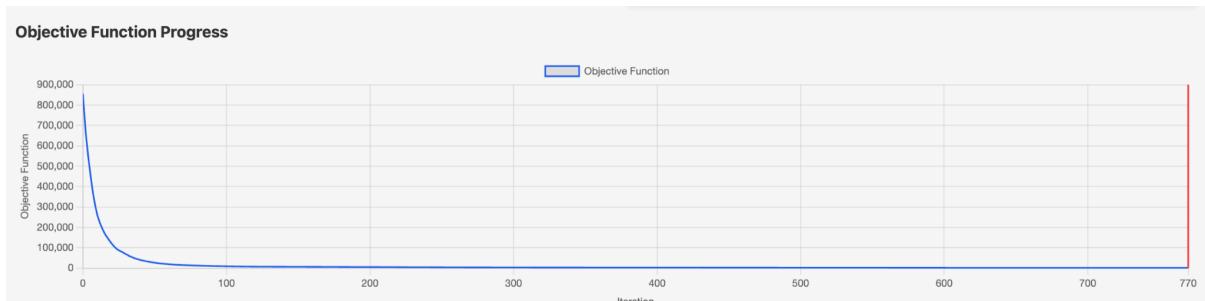
Gambar 10. State Awal Uji Coba Ke - 1 dengan *Sideways Moves*

#### Diagonal Magic Cube Display



Gambar 11. Final State Uji Coba Ke - 1 dengan *Sideways Moves*

Berdasarkan hasil uji coba pertama dengan algoritma **Hill-Climbing With Sideways Moves**, dapat dilihat pada hasil nilai *objective function* yang didapatkan pada initial state awal pada *magic cube* dan state akhir yang didapatkan pada *neighbor* yang lain juga berbeda. Total durasi yang digunakan untuk melakukan pencarian *successor* tersebut ialah sekitar **6.23 detik** dengan batas maksimum *Sideways Moves* yang dapat dilakukan oleh algoritma *local search* selama proses iterasi pengecekan ialah **100 moves**. Apabila dilihat pada Gambar 10 pada bagian *Local Search Information* ataupun *Current State Information*, dapat dinyatakan state awal pada *magic cube* tersebut bernilai **855.744**. Sedangkan apabila dilihat pada Gambar 11, state akhir *value successor* yang didapatkan pada iterasi *final* tersebut, yakni pada iterasi ke-**770** ialah senilai **1.225**. Hal ini justru menunjukkan perbedaan yang signifikan terhadap *value successor* yang didapatkan oleh algoritma *local search* dimana *value* dari *final state* tersebut bernilai lebih baik dibandingkan *value state* awal sehingga dapat disimpulkan bahwa nilai *value successor* pada *final state magic cube* akan diambil sebagai nilai *objective function* yang paling optimal yakni bernilai **1.225**. Berikut merupakan grafik hasil plot terhadap nilai *objective function* terhadap banyak iterasi yang dilakukan oleh algoritma *local search*.



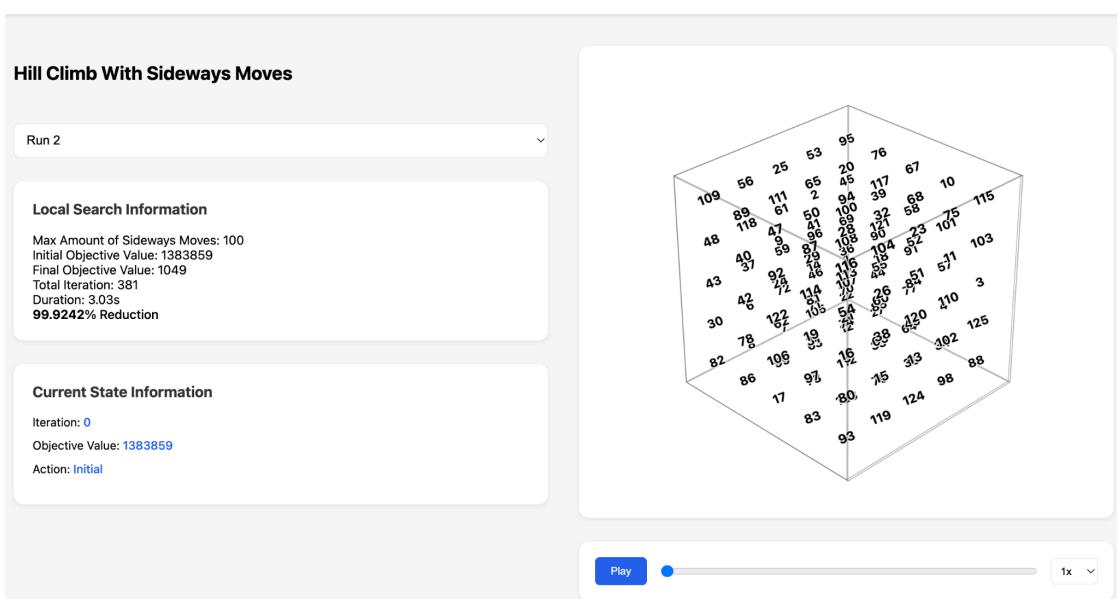
Gambar 12. Plot Objective Function Uji Coba Ke - 1 Sideways Moves

Apabila dilihat pada grafik di atas, dapat dilihat bahwa grafik menunjukkan perubahan secara linier konstan dari awal iterasi dilakukan hingga proses iterasi terakhir dilakukan. Pada initial state awal proses iterasi, *value objective function* yang dimiliki oleh *diagonal magic cube* tersebut bernilai 855.744 sedangkan pada hasil iterasi terakhir, nilai *objective function* dari state yang diperiksa memiliki nilai yang jauh lebih kecil daripada nilai *objective function* di awal state yang dimiliki oleh *cube* tersebut, yakni senilai 1.225. Dengan demikian, dapat ditarik kesimpulan bahwa semakin banyak pengecekan iterasi yang dilakukan oleh algoritma *local search*, nilai *objective function* juga akan semakin optimal dimana tujuan dari penerapan algoritma *Hill-Climbing With Sideways Moves* ialah juga untuk menghindari adanya kondisi *local maximum* sehingga ketika algoritma sedang melakukan

pengecekan pada setiap *successor* yang ada dan memiliki nilai *objective function* yang sama dengan *state* sebelumnya, maka algoritma dapat langsung menghindari kondisi tersebut untuk mencegah hambatan dalam mencari *value successor* yang paling optimal. Maka dari itu, algoritma *Hill-Climbing With Sideways Moves* akan mengembalikan nilai *objective function* yang paling optimal pada iterasi terakhir, yakni iterasi ke-770 dengan value **1.225**.

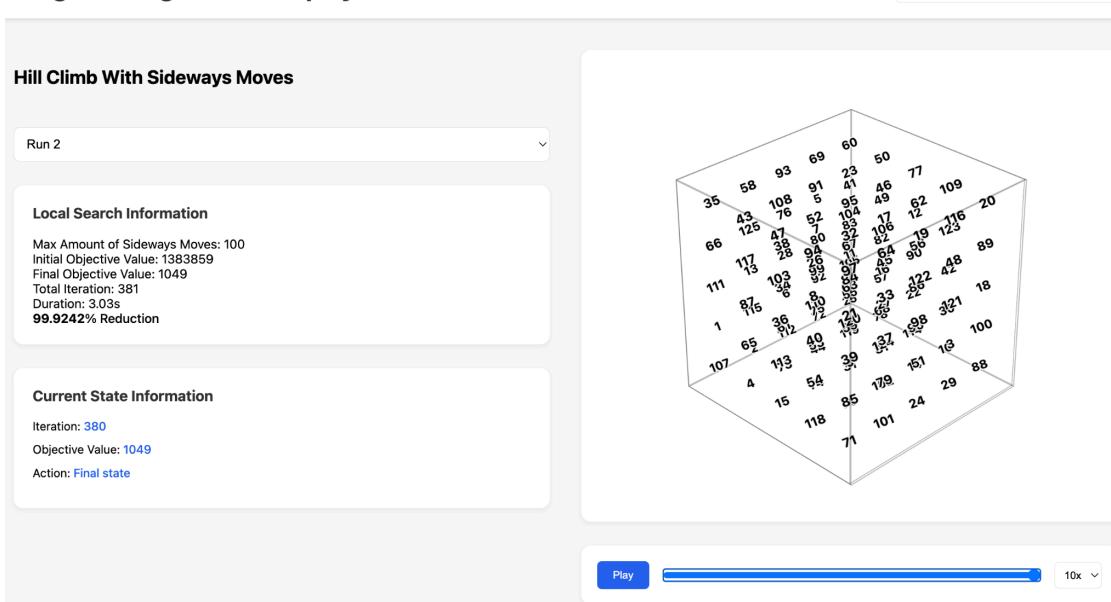
### ● Percobaan Ke - 2

Diagonal Magic Cube Display



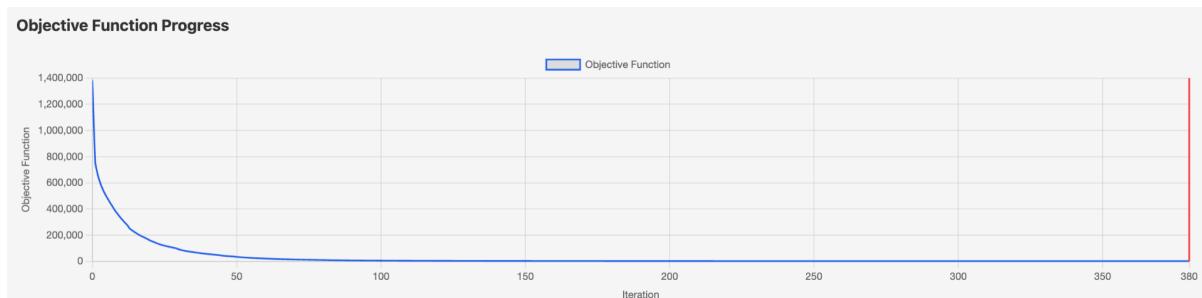
Gambar 13. State Awal Uji Coba Ke - 2 Sideways Moves

Diagonal Magic Cube Display



Gambar 14. Final State Uji Coba Ke - 2 Sideways Moves

Berdasarkan hasil uji coba kedua pada algoritma **Hill-Climbing With Sideways Moves**, dapat dilihat pada hasil nilai *objective function* yang didapatkan pada initial state awal pada *magic cube* dan state akhir yang didapatkan pada *neighbor* yang lain juga berbeda. Total durasi yang digunakan untuk melakukan pencarian *successor* tersebut ialah sekitar **3.03 second** dengan batas maksimum *Sideways Moves* yang dapat dilakukan oleh algoritma *local search* selama proses iterasi pengecekan ialah **100 moves**. Apabila dilihat pada Gambar 13 pada bagian *Local Search Information* ataupun *Current State Information*, dapat dinyatakan state awal pada *magic cube* tersebut bernilai **1.383.859**. Sedangkan apabila dilihat pada Gambar 14, state akhir *value successor* yang didapatkan pada iterasi *final* tersebut, yakni pada iterasi ke-**380** ialah senilai **1.049**. Hal ini justru menunjukkan perbedaan yang signifikan terhadap *value successor* yang didapatkan oleh algoritma *local search* dimana *value* dari *final state* tersebut bernilai lebih baik dibandingkan *value state* awal sehingga dapat disimpulkan bahwa nilai *value successor* pada *final state magic cube* akan diambil sebagai nilai *objective function* yang paling optimal yakni bernilai **1.049**. Berikut merupakan grafik hasil plot terhadap nilai *objective function* terhadap banyak iterasi yang dilakukan oleh algoritma *local search*.

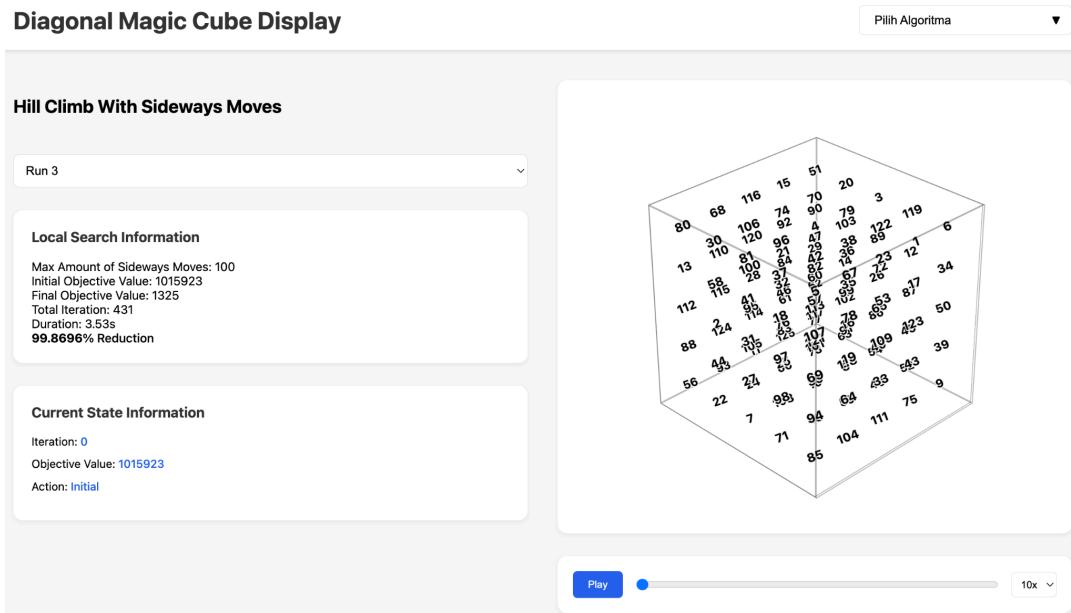


Gambar 15. Plot *Objective Function Sideways Moves* Uji Coba Ke - 2

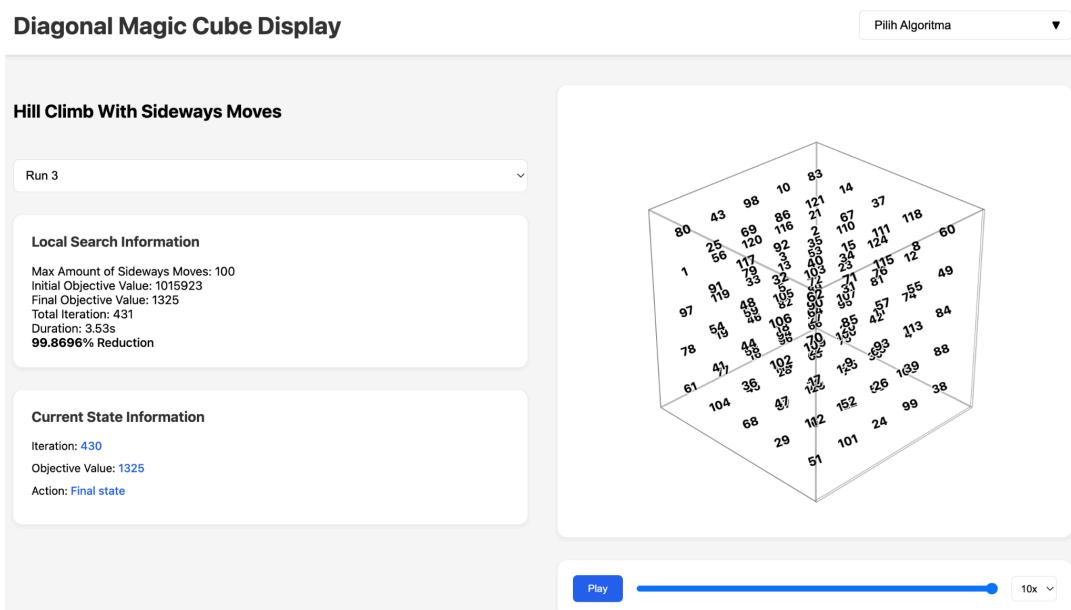
Apabila dilihat pada grafik di atas, dapat dilihat bahwa grafik menunjukkan perubahan secara linier konstan dari awal iterasi dilakukan hingga proses iterasi terakhir dilakukan. Pada initial state awal proses iterasi, *value objective function* yang dimiliki oleh *diagonal magic cube* tersebut bernilai 1.383.859 sedangkan pada hasil iterasi terakhir, nilai *objective function* dari *state* yang diperiksa memiliki nilai yang jauh lebih kecil daripada nilai *objective function* di awal *state* yang dimiliki oleh *cube* tersebut, yakni senilai 1.049. Dengan demikian, dapat ditarik kesimpulan bahwa semakin banyak pengecekan iterasi yang dilakukan oleh algoritma *local search*, nilai *objective function* juga akan semakin optimal

dimana tujuan dari penerapan algoritma *Hill-Climbing With Sideways Moves* ialah juga untuk menghindari adanya kondisi *local maximum* sehingga ketika algoritma sedang melakukan pengecekan pada setiap *successor* yang ada dan memiliki nilai *objective function* yang sama dengan *state* sebelumnya, maka algoritma dapat langsung menghindari kondisi tersebut untuk mencegah hambatan dalam mencari *value successor* yang paling optimal. Maka dari itu, algoritma *Hill-Climbing With Sideways Moves* akan mengembalikan nilai *objective function* yang paling optimal pada iterasi terakhir, yakni iterasi ke-380 dengan value **1.049**.

### ● Percobaan Ke - 3

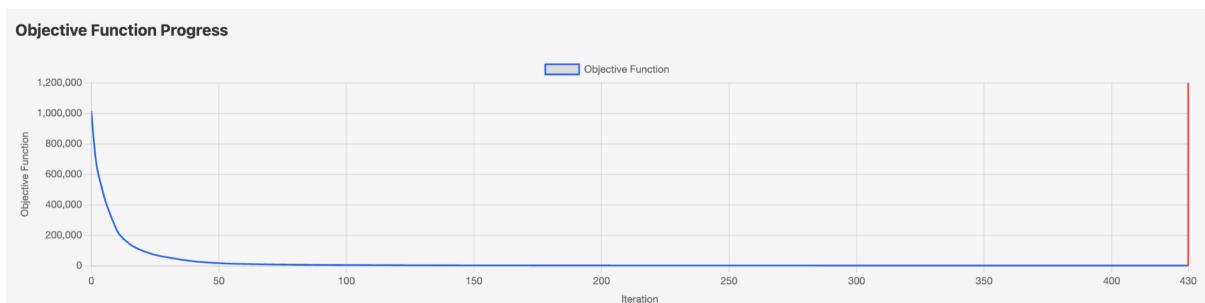


Gambar 16. State Awal Uji Coba Ke - 3 Sideways Moves



Gambar 17. Final State Uji Coba Ke - 2 Sideways Moves

Berdasarkan hasil uji coba kedua pada algoritma **Hill-Climbing With Sideways Moves**, dapat dilihat pada hasil nilai *objective function* yang didapatkan pada initial state awal pada *magic cube* dan state akhir yang didapatkan pada *neighbor* yang lain juga berbeda. Total durasi yang digunakan untuk melakukan pencarian *successor* tersebut ialah sekitar **3.53 detik** dimana bila dibandingkan dengan uji coba pertama pada algoritma *Hill-Climbing With Sideways Moves* yakni berkisar **6.23 detik** yang lebih lambat dibandingkan uji coba ketiga. Sedangkan jika dibandingkan dengan uji coba kedua, yakni berkisar **3.03 detik** yang justru membuat percobaan ketiga pada *Sideways Moves* menjadi lebih tidak efisien dari percobaan sebelumnya. Batas maksimum *Sideways Moves* yang dapat dilakukan oleh algoritma *local search* selama proses iterasi pengecekan ialah **100 moves**. Apabila dilihat pada Gambar 16 pada bagian *Local Search Information* ataupun *Current State Information*, dapat dinyatakan state awal pada *magic cube* tersebut bernilai **1.015.923**. Sedangkan apabila dilihat pada Gambar 17, state akhir *value successor* yang didapatkan pada iterasi *final* tersebut, yakni pada iterasi ke-**430** ialah senilai **1.325**. Hal ini justru menunjukkan perbedaan yang signifikan terhadap *value successor* yang didapatkan oleh algoritma *local search* dimana *value* dari *final state* tersebut bernilai lebih baik dibandingkan *value state* awal sehingga dapat disimpulkan bahwa nilai *value successor* pada *final state magic cube* akan diambil sebagai nilai *objective function* yang paling optimal yakni bernilai **1.325**. Berikut merupakan grafik hasil plot terhadap nilai *objective function* terhadap banyak iterasi yang dilakukan oleh algoritma *local search*.



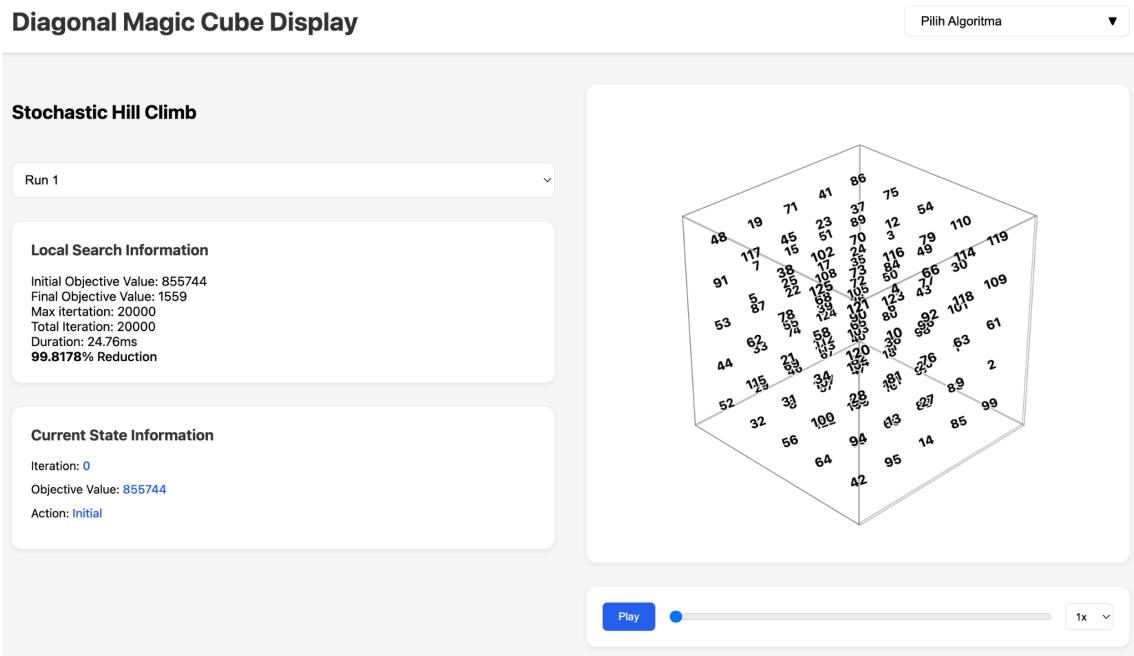
Gambar 18. Plot *Objective Function* Sideways Moves Uji Coba Ke - 3

Apabila dilihat pada grafik di atas, dapat dilihat bahwa grafik menunjukkan perubahan secara linier konstan dari awal iterasi dilakukan hingga proses iterasi terakhir

dilakukan. Pada initial *state* awal proses iterasi, *value objective function* yang dimiliki oleh *diagonal magic cube* tersebut bernilai 1.015.923 sedangkan pada hasil iterasi terakhir, nilai *objective function* dari *state* yang diperiksa memiliki nilai yang jauh lebih kecil daripada nilai *objective function* di awal *state* yang dimiliki oleh *cube* tersebut, yakni senilai 1.325. Dengan demikian, dapat ditarik kesimpulan bahwa semakin banyak pengecekan iterasi yang dilakukan oleh algoritma *local search*, nilai *objective function* juga akan semakin optimal dimana tujuan dari penerapan algoritma *Hill-Climbing With Sideways Moves* ialah juga untuk menghindari adanya kondisi *local maximum* sehingga ketika algoritma sedang melakukan pengecekan pada setiap *successor* yang ada dan memiliki nilai *objective function* yang sama dengan *state* sebelumnya, maka algoritma dapat langsung menghindari kondisi tersebut untuk mencegah hambatan dalam mencari *value successor* yang paling optimal. Maka dari itu, algoritma *Hill-Climbing With Sideways Moves* akan mengembalikan nilai *objective function* yang paling optimal pada iterasi terakhir, yakni iterasi ke-430 dengan value **1.325**.

### 3. *Stochastic Hill-Climbing Search*

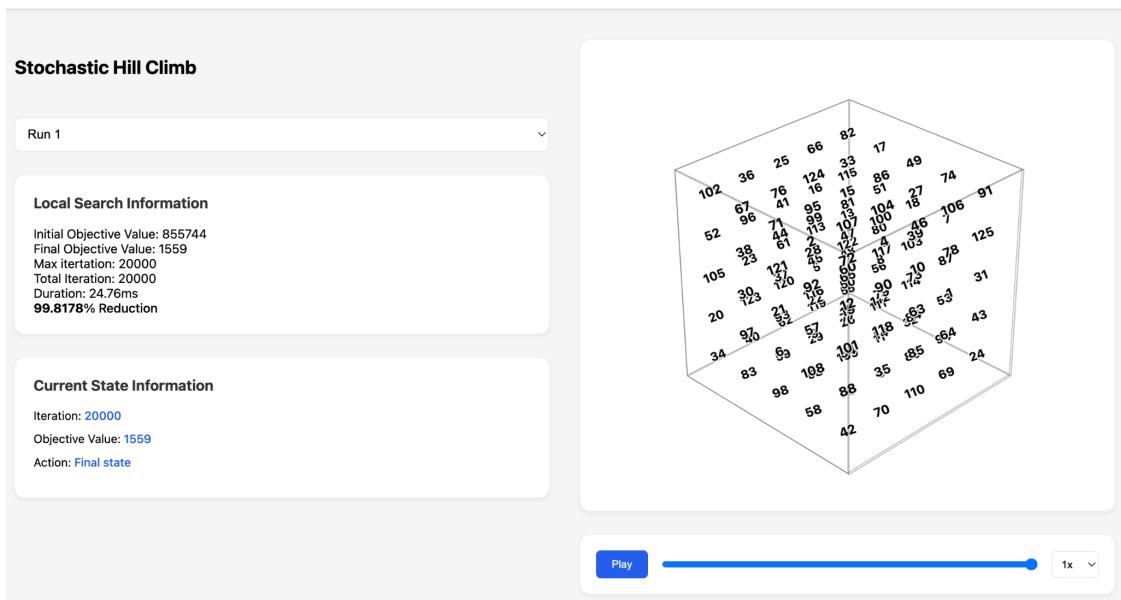
- **Percobaan Ke - 1**



Gambar 19. *State* Awal Uji Coba Ke - 1 *Stochastic Hill-Climbing*

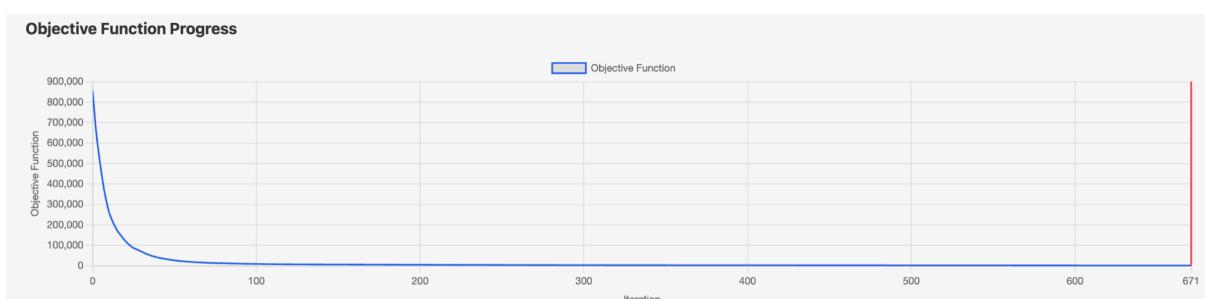
## Diagonal Magic Cube Display

Pilih Algoritma ▾



Gambar 20. *Final State Uji Coba Ke -1 Stochastic Hill-Climbing*

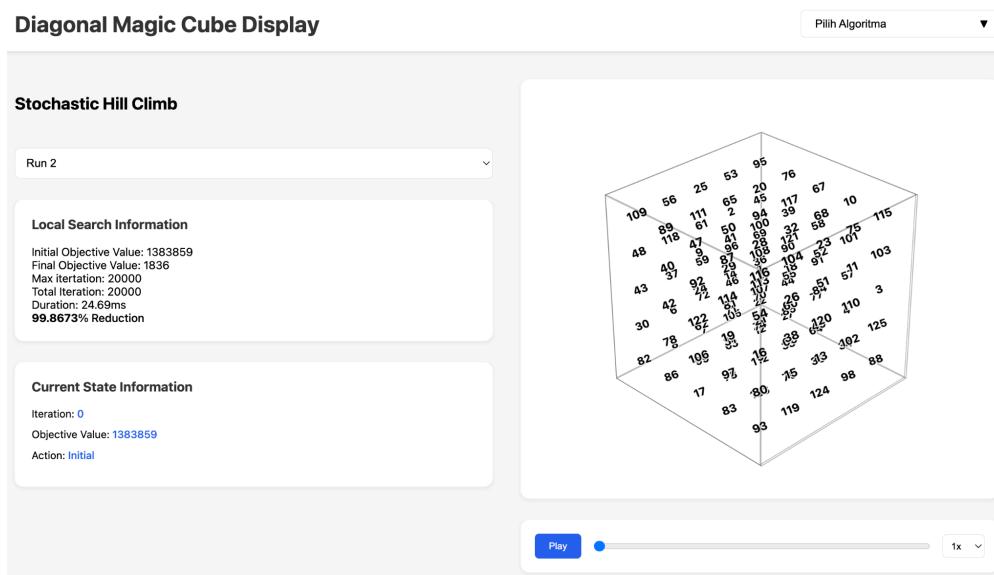
Berdasarkan hasil uji coba pertama terhadap implementasi algoritma ***Stochastic Hill-Climbing Search*** yang telah dilakukan sesuai dengan gambar di atas, dapat dilihat bahwa terdapat perubahan signifikan yang terjadi pada *state* awal dan *state* akhir *diagonal magic cube* tersebut. Durasi yang digunakan untuk melakukan proses pencarian tersebut ialah berkisar **24.76 milliseconds** sesuai dengan yang ditampilkan pada Gambar 19 pada bagian *Local Search Information*. Maksimum iterasi yang dapat dilakukan oleh algoritma *local search* ialah sebanyak **20.000 kali**. Seperti yang dapat dilihat, *state* awal (iterasi ke-0) pada *diagonal magic cube* tersebut menunjukkan nilai *objective function* berupa **855.744** sebagai *value successor* awal. Sedangkan pada Gambar 20, ditunjukkan nilai dari proses iterasi terakhir, yakni iterasi ke-20.000, nilai *objective function* yang didapatkan berupa **1.559** yang sekaligus menunjukkan bahwa *value successor* pada *neighbor* tersebut adalah yang paling optimal dibandingkan dengan *successor* lainnya sehingga solusi pada hasil uji coba pertama dapat disimpulkan akan mengambil nilai *final state* yakni berupa 1.559 sebagai nilai *objective function* yang paling optimal tersebut. Berikut merupakan hasil plot nilai *objective function* terhadap banyak iterasi yang telah dilakukan oleh algoritma *local search* selama proses pengecekan *successor*.



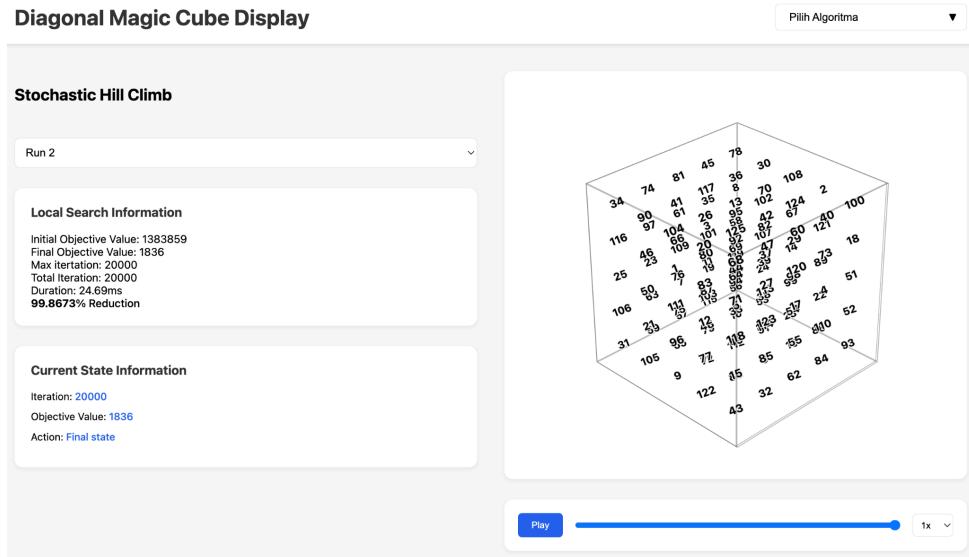
Gambar 21. Plot *Objective Function Stochastic Hill-Climbing* Uji Coba Ke - 1

Dapat dilihat pada grafik tersebut, bahwa hasil perbandingan nilai *objective function* yang didapatkan pada setiap *successor* saat iterasi pengecekan dilakukan oleh algoritma *local search* memiliki perbedaan yang cukup signifikan. Dapat dilihat bahwa semakin banyak iterasi yang dilakukan oleh algoritma *local search*, maka nilai *objective function* yang dihasilkan pun akan semakin kecil dibandingkan dengan initial *state* awal pada *diagonal magic cube* tersebut. Algoritma *Stochastic Hill-Climbing Search* bertujuan untuk mencari *successor* dengan *value* yang bernilai relatif lebih besar dari *state* sebelumnya, dimana apabila algoritma *local search* menemukan *objective function* yang bernilai sama maka algoritma akan tetap melanjutkan iterasi untuk melakukan pencarian *successor* selanjutnya hingga menemukan *value neighbor* yang lebih baik. Sehingga dari hasil uji coba pertama yang telah dilakukan dapat dilihat bahwa iterasi terakhir yang dilakukan oleh program yakni iterasi ke-20.000 mengembalikan nilai *objective function* sebesar **1.559**.

## • Percobaan Ke - 2

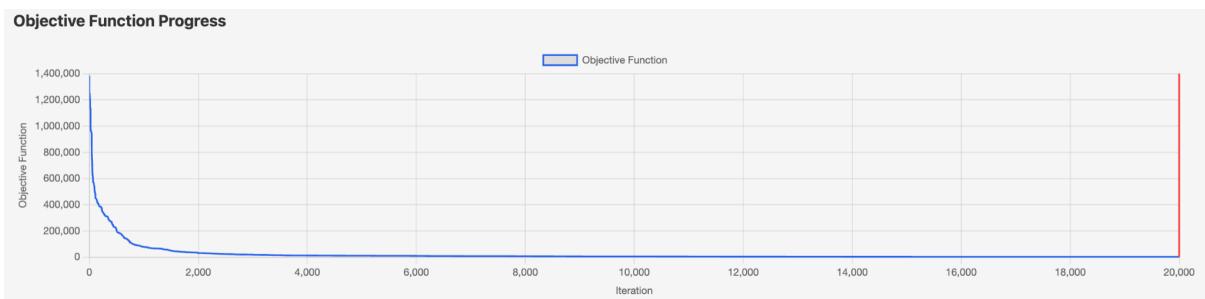


Gambar 22. State Awal Uji Coba Ke - 2 Stochastic Hill Climbing



Gambar 23. Final State Uji Coba Ke - 2 Stochastic Hill-Climbing

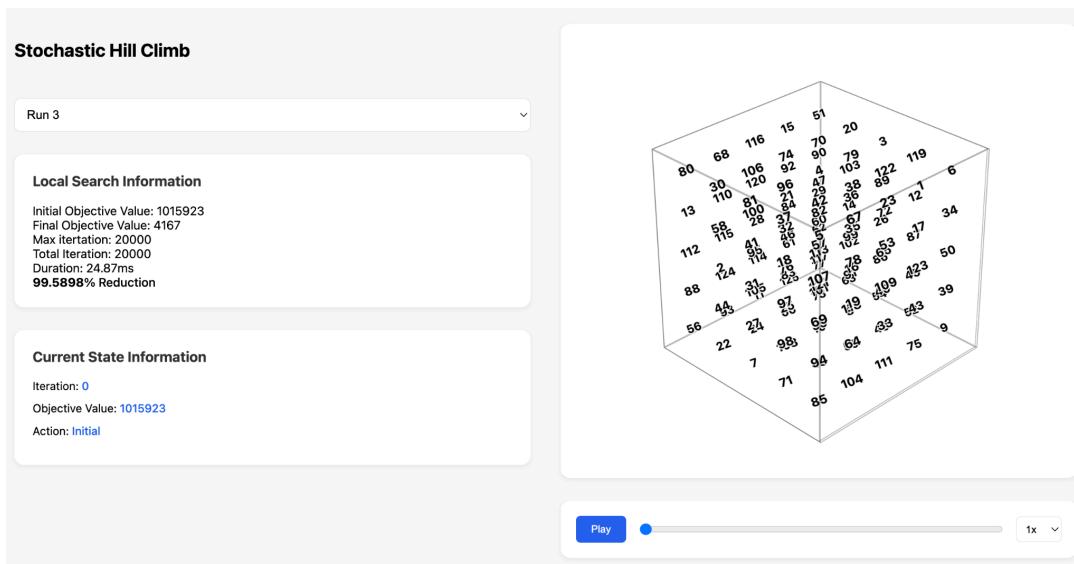
Berdasarkan hasil uji coba pertama terhadap implementasi algoritma **Stochastic Hill-Climbing Search** yang telah dilakukan sesuai dengan gambar di atas, dapat dilihat bahwa terdapat perubahan signifikan yang terjadi pada *state* awal dan *state* akhir *diagonal magic cube* tersebut. Durasi yang digunakan untuk melakukan proses pencarian tersebut ialah berkisar **24.69 milliseconds** sesuai dengan yang ditampilkan pada Gambar 22 pada bagian *Local Search Information*. Namun, bila dibandingkan dengan hasil uji coba pada bagian pertama, durasi waktu yang dibutuhkan justru lebih singkat daripada sebelumnya. Maksimum iterasi yang dapat dilakukan oleh algoritma *local search* ialah sebanyak **20.000 kali**. Seperti yang dapat dilihat, *state* awal (iterasi ke-0) pada *diagonal magic cube* tersebut menunjukkan nilai *objective function* berupa **1.383.859** sebagai *value successor* awal. Sedangkan pada Gambar 23 ditunjukkan nilai dari proses iterasi terakhir, yakni iterasi ke-20.000. Nilai *objective function* yang didapatkan berupa **1.836** yang sekaligus menunjukkan bahwa *value successor* pada *neighbor* tersebut adalah yang paling optimal dibandingkan dengan *successor* lainnya. Dengan demikian, solusi pada hasil uji coba kedua Algoritma *Stochastic Hill-Climbing Search* dapat disimpulkan akan mengambil nilai *final state* yakni berupa 1.836 sebagai nilai *objective function* yang lebih baik selama proses pengecekan *successor* berlangsung. Akan tetapi, apabila dibandingkan dengan hasil uji coba pada bagian pertama, nilai *objective function* yang didapatkan tidak lebih baik dari *final state* yang ada pada bagian uji coba ke-1. Berikut merupakan hasil plot nilai *objective function* terhadap banyak iterasi yang telah dilakukan oleh algoritma *local search* selama proses pengecekan *successor*.



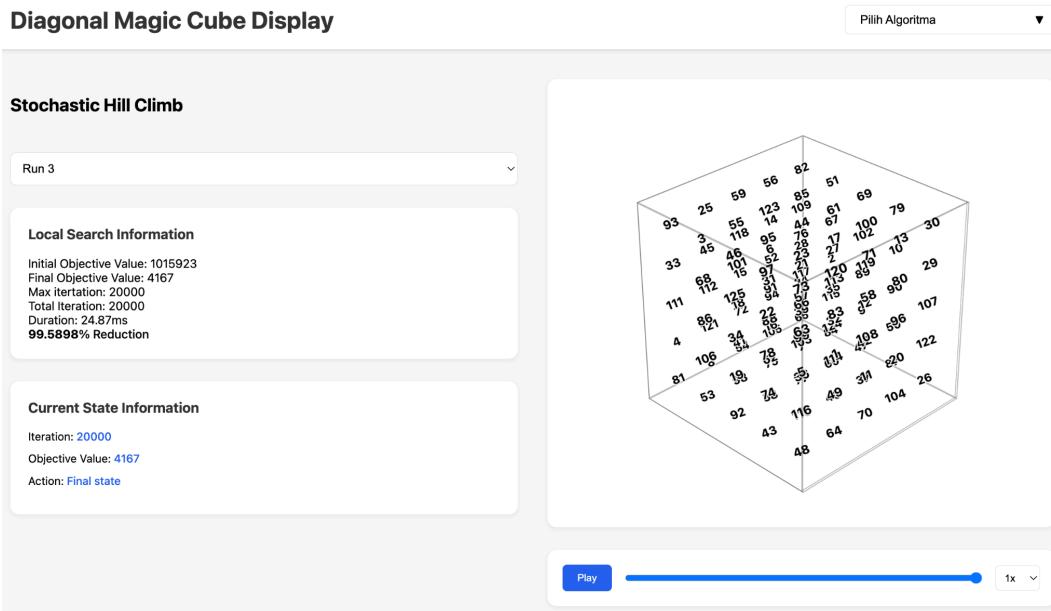
Gambar 24. Plot *Objective Function Stochastic Hill-Climbing* Uji Coba Ke - 2

Dapat dilihat pada grafik tersebut, bahwa hasil perbandingan nilai *objective function* yang didapatkan pada setiap *successor* saat iterasi pengecekan dilakukan oleh algoritma *local search* memiliki perbedaan yang cukup signifikan. Dapat dilihat bahwa semakin banyak iterasi yang dilakukan oleh algoritma *local search*, maka nilai *objective function* yang dihasilkan pun akan semakin kecil dibandingkan dengan initial *state* awal pada *diagonal magic cube* tersebut. Sehingga dari hasil uji coba pertama yang telah dilakukan dapat dilihat bahwa iterasi terakhir yang dilakukan oleh program yakni iterasi ke-20.000 mengembalikan nilai *objective function* sebesar **1.836**.

### ● Percobaan Ke - 3

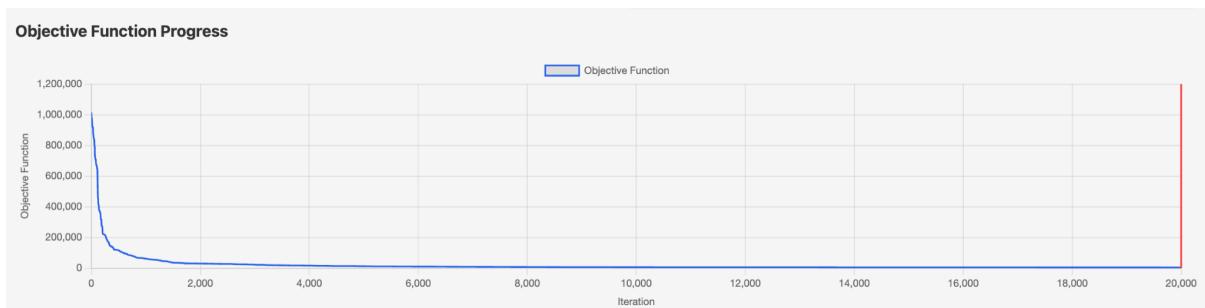


Gambar 25. *State* Awal Uji Coba Ke - 3 *Stochastic Hill-Climbing*



Gambar 26. *Final State Uji Coba Ke - 3 Stochastic Hill-Climbing*

Berdasarkan hasil uji coba pertama terhadap implementasi algoritma ***Stochastic Hill-Climbing Search*** yang telah dilakukan sesuai dengan gambar di atas, dapat dilihat bahwa terdapat perubahan signifikan yang terjadi pada *state* awal dan *state* akhir *diagonal magic cube* tersebut. Durasi yang digunakan untuk melakukan proses pencarian tersebut ialah berkisar **24.87 milliseconds** sesuai dengan yang ditampilkan pada Gambar 25 pada bagian *Local Search Information*. Percobaan ketiga tidak jauh berbeda dengan durasi yang didapatkan pada percobaan pertama maupun percobaan kedua dimana masing-masing percobaan berkisar pada **24 milliseconds**. Maksimum iterasi yang dapat dilakukan oleh algoritma *local search* ialah sebanyak **20.000 kali**. Seperti yang dapat dilihat, *state* awal (iterasi ke-0) pada *diagonal magic cube* tersebut menunjukkan nilai *objective function* berupa **1.015.923** sebagai *value successor* awal. Sedangkan pada Gambar 26, ditunjukkan nilai dari proses iterasi terakhir, yakni iterasi ke-20.000, nilai *objective function* yang didapatkan berupa **4.167** yang sekaligus menunjukkan bahwa *value successor* pada *neighbor* tersebut adalah yang paling optimal dibandingkan dengan *successor* lainnya sehingga solusi pada hasil uji coba pertama dapat disimpulkan akan mengambil nilai *final state* yakni berupa 4.167 sebagai nilai *objective function* yang paling optimal tersebut. Akan tetapi, apabila dibandingkan dengan Berikut merupakan hasil plot nilai *objective function* terhadap banyak iterasi yang telah dilakukan oleh algoritma *local search* selama proses pengecekan *successor*.

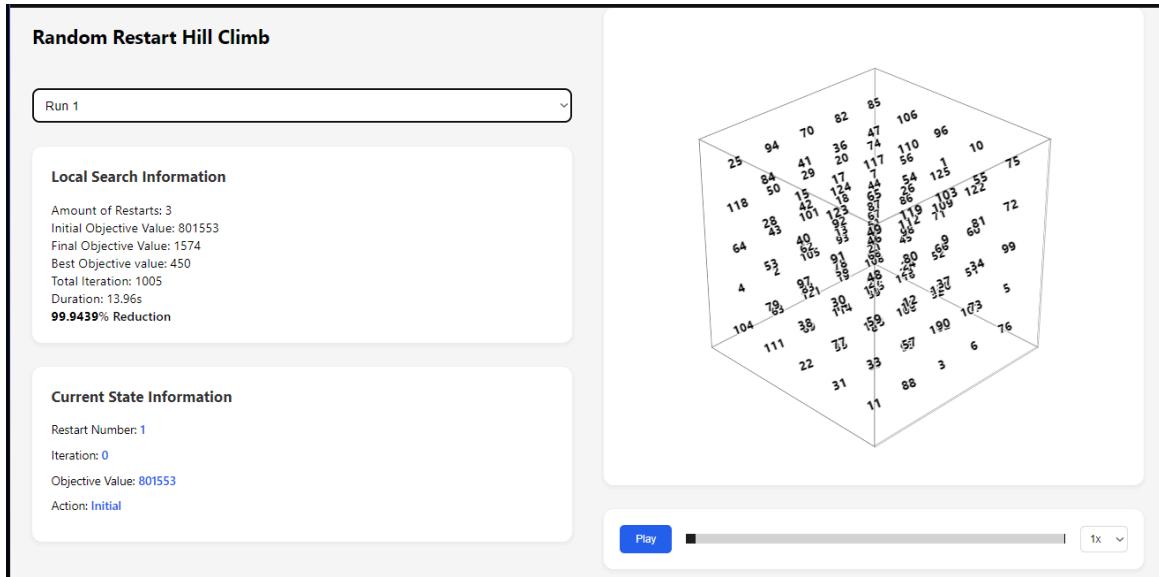


Gambar 27. Plot *Objective Function Stochastic Hill-Climbing* Uji Coba Ke - 3

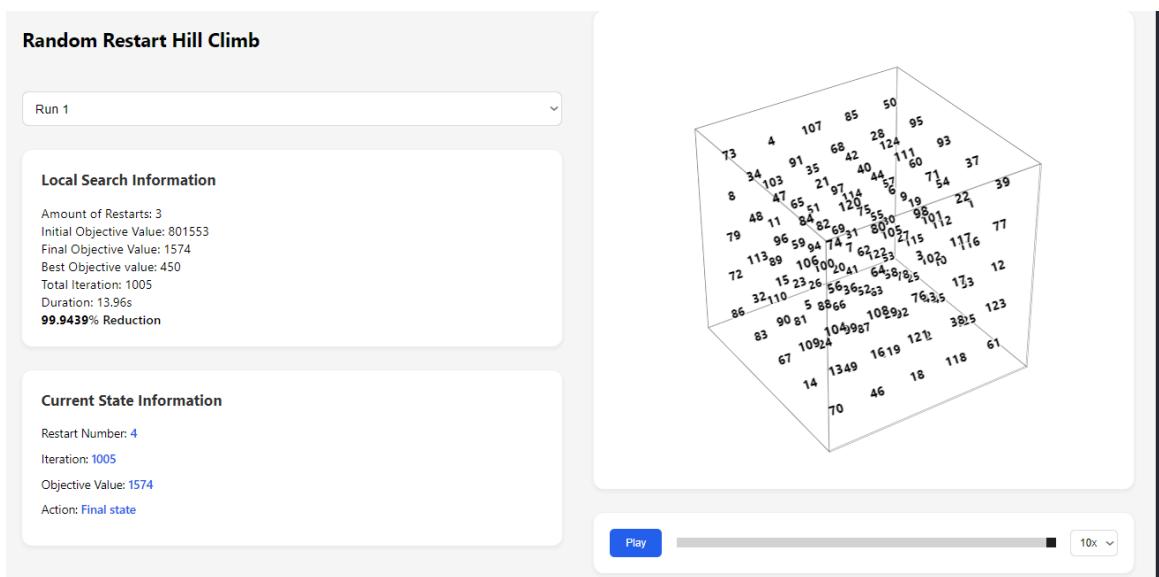
Dapat dilihat pada grafik tersebut, bahwa hasil perbandingan nilai *objective function* yang didapatkan pada setiap *successor* saat iterasi pengecekan dilakukan oleh algoritma *local search* memiliki perbedaan yang cukup signifikan. Dapat dilihat bahwa semakin banyak iterasi yang dilakukan oleh algoritma *local search*, maka nilai *objective function* yang dihasilkan pun akan semakin kecil dibandingkan dengan initial *state* awal pada *diagonal magic cube* tersebut. Algoritma *Stochastic Hill-Climbing Search* bertujuan untuk mencari *successor* dengan *value* yang bernilai lebih besar dari *state* sebelumnya, dimana apabila algoritma *local search* menemukan *objective function* yang bernilai sama maka algoritma akan tetap melanjutkan iterasi untuk melakukan pencarian *successor* selanjutnya hingga menemukan *value neighbor* yang paling optimal. Sehingga dari hasil uji coba ketiga yang telah dilakukan dapat dilihat bahwa iterasi terakhir yang dilakukan oleh program yakni iterasi ke-20.000 mengembalikan nilai *objective function* sebesar **4.167**.

#### 4. Random Restart Hill-Climbing Search

##### • Percobaan Ke - 1



Gambar 28. State Awal Uji Coba Ke-1 Random Restart Hill Climbing



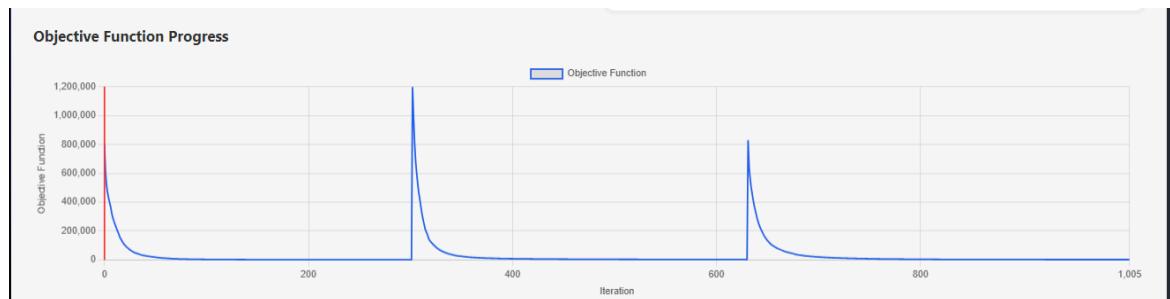
Gambar 29. State Akhir Uji Coba Ke-1 Random Restart Hill Climbing

Berdasarkan hasil uji coba yang pertama, didapati hasil seperti gambar di atas. Awalnya kubus memiliki *objective value* bernilai **801.553**. Setelah algoritma *Random Restart Hill Climbing* dijalankan, *state* kubus semakin mendekati solusi. Dapat dilihat pada gambar *state* akhir dari kubus, bahwa ada perubahan signifikan dari *objective value* kubus. Namun, perlu diperhatikan, bahwa hasil final *objective value* bukanlah *objective value* yang terbaik yang telah ditemukan algoritma. Hal ini terjadi karena *Random Restart Hill-Climbing* melakukan pengulangan beberapa kali sebagai cara algoritma menghindari *local maximum*.

Sehingga, algoritma ini dapat menghasilkan beberapa *end state*. *Objective value* terbaik yang didapat algoritma adalah **450** dengan *final objective valuenya* adalah **1.574**.

Pada loop yang pertama *objective value* kubus awal adalah **801.553** dan berakhir pada nilai **450**. Pada loop kedua, *objective value* kubus dimulai di **1.194.059** dan berakhir di **2029**. Pada loop ketiga, *objective value* kubus dimulai di **825.163** dan berakhir di **1574**.

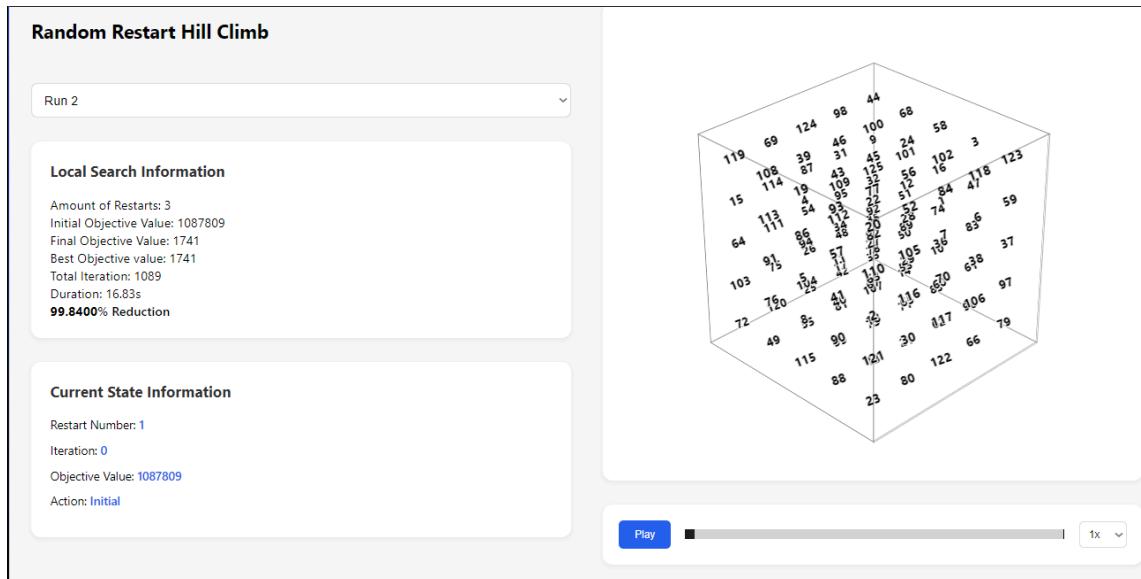
Algoritma melakukan pengulangan sebanyak 3 kali, sehingga algoritma memiliki 3 *end state*. Total iterasi yang dilakukan algoritma untuk melakukan pengulangan sebanyak 3 kali adalah **1005**. Durasi berjalannya algoritma pada uji coba ini adalah **13.96 s**. Berikut adalah plot nilai *objective function* sepanjang iterasi selama algoritma berlangsung.



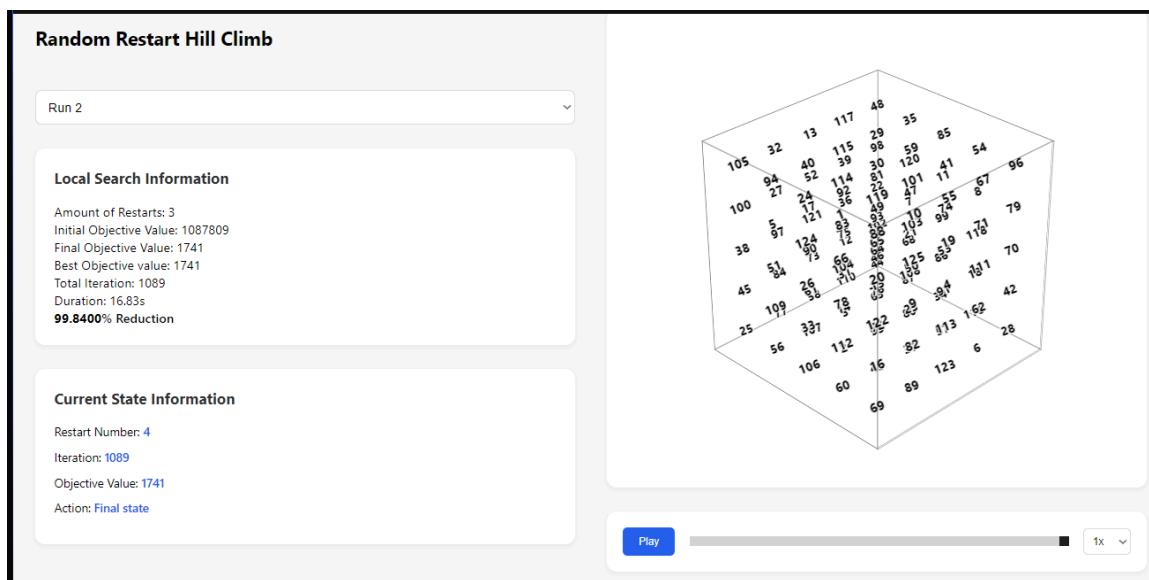
Gambar 30. Plot *Objective Function* Uji Coba ke-1 *Random Restart Hill Climbing*

Berdasarkan hasil algoritma dari *Random Restart Hill Climbing*, dibuat plot yang menampilkan penyebaran dari nilai *Objective Function* sepanjang program berjalan. Dapat dilihat pada grafik, terjadi penurunan *Objective Function* di awal, lalu naik secara signifikan, kemudian menurun lagi. Pola ini terjadi sebanyak 3 kali. Pola ini menunjukkan bahwa algoritma melakukan pengulangan pencarian sebanyak 3 kali dengan *state* awal kubus yang berbeda-beda untuk menemukan solusi yang terbaik.

## ● Percobaan Ke - 2



Gambar 31. State Awal Uji Coba Ke-2 Random Restart Hill Climbing

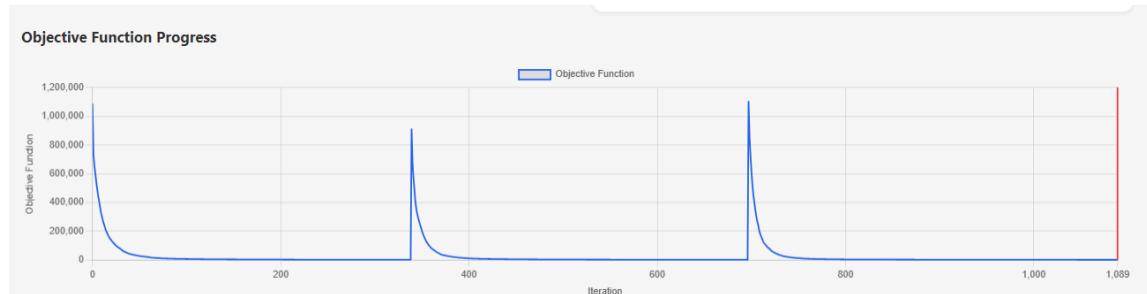


Gambar 32. State Akhir Uji Coba Ke-2 Random Restart Hill Climbing

Berdasarkan hasil uji coba yang kedua, didapati hasil seperti gambar di atas. Awalnya kubus memiliki *objective value* bernilai **1.087.809**. Setelah algoritma *Random Restart Hill Climbing* dijalankan, state kubus semakin mendekati solusi. Dapat dilihat pada gambar *state akhir* dari kubus, bahwa ada perubahan signifikan dari *objective value* kubus. *Objective value* terbaik yang didapat algoritma adalah **1.741** dengan *final objective value*nya adalah sama, yaitu **1.741**.

Pada *loop* yang pertama *objective value* kubus awal adalah **1.087.809** dan berakhir pada nilai **2.176**. Pada *loop* kedua, *objective value* kubus dimulai di **909.338** dan berakhir di **2089**. Pada *loop* ketiga, *objective value* kubus dimulai di **1.100.867** dan berakhir di **1741**.

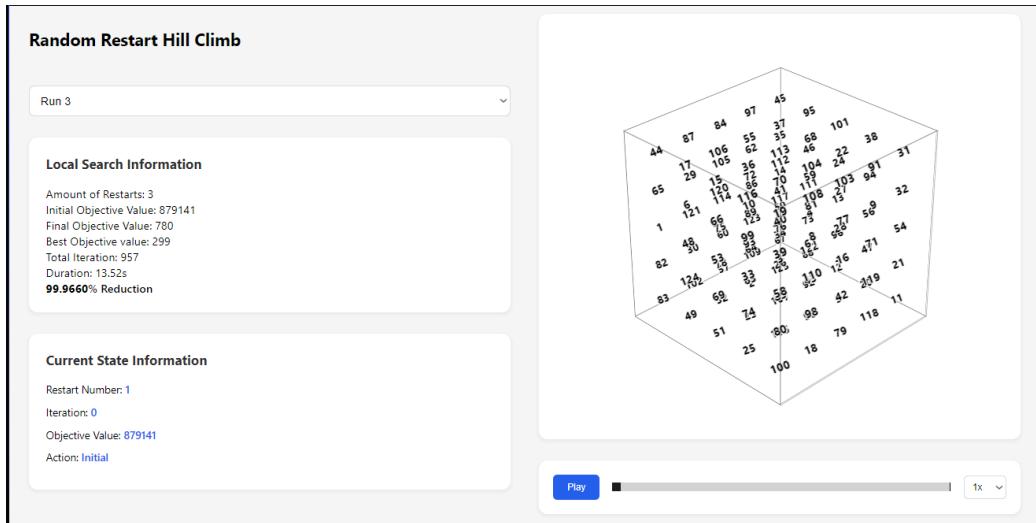
Algoritma melakukan pengulangan sebanyak 3 kali, sehingga algoritma memiliki 3 *end state*. Total iterasi yang dilakukan algoritma untuk melakukan pengulangan sebanyak 3 kali adalah **1089**. Durasi berjalannya algoritma pada uji coba ini adalah **16.83 s**. Berikut adalah plot nilai *objective function* sepanjang iterasi selama algoritma berlangsung.



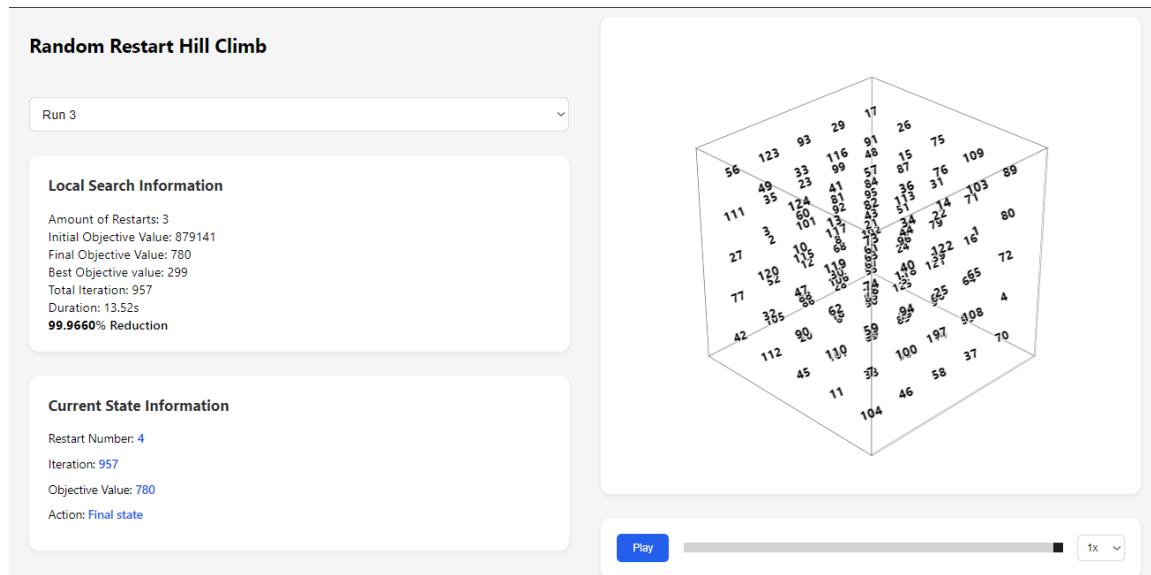
Gambar 33. Plot *Objective Function* Uji Coba ke-2 *Random Restart Hill Climbing*

Berdasarkan hasil algoritma dari *Random Restart Hill Climbing*, dibuat plot yang menampilkan penyebaran dari nilai *Objective Function* sepanjang program berjalan. Dapat dilihat pada grafik, terjadi penurunan *Objective Function* di awal, lalu naik secara signifikan, kemudian menurun lagi. Pola ini terjadi sebanyak 3 kali. Pola ini menunjukkan bahwa algoritma melakukan pengulangan pencarian sebanyak 3 kali dengan *state* awal kubus yang berbeda-beda untuk menemukan solusi yang terbaik.

### ● Percobaan Ke - 3



Gambar 34. State Awal Uji Coba Ke-3 Random Restart Hill Climbing

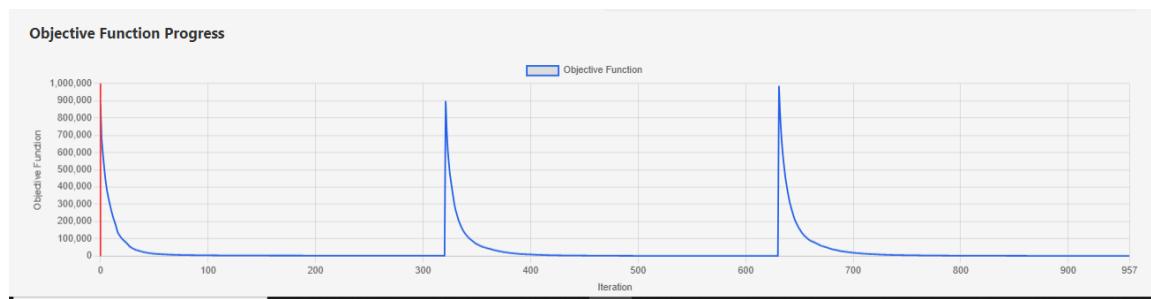


Gambar 35. State Akhir Uji Coba Ke-3 Random Restart Hill Climbing

Berdasarkan hasil uji coba yang ketiga, didapatkan hasil seperti gambar di atas. Awalnya kubus memiliki *objective value* bernilai **879.141**. Setelah algoritma *Random Restart Hill Climbing* dijalankan, state kubus semakin mendekati solusi. Dapat dilihat pada gambar *state akhir* dari kubus, bahwa ada perubahan signifikan dari *objective value* kubus. *Objective value* terbaik yang didapat algoritma adalah **299** dengan *final objective value*nya adalah **957**.

Pada loop yang pertama *objective value* kubus awal adalah **879.141** dan berakhir pada nilai **1.824**. Pada loop kedua, *objective value* kubus dimulai di **894.744** dan berakhir di **299**. Pada loop ketiga, *objective value* kubus dimulai di **982.052** dan berakhir di **780**.

Algoritma melakukan pengulangan sebanyak 3 kali, sehingga algoritma memiliki 3 *end state*. Total iterasi yang dilakukan algoritma untuk melakukan pengulangan sebanyak 3 kali adalah **957**. Durasi berjalannya algoritma pada uji coba ini adalah **13.52 s**. Berikut adalah plot nilai *objective function* sepanjang iterasi selama algoritma berlangsung.

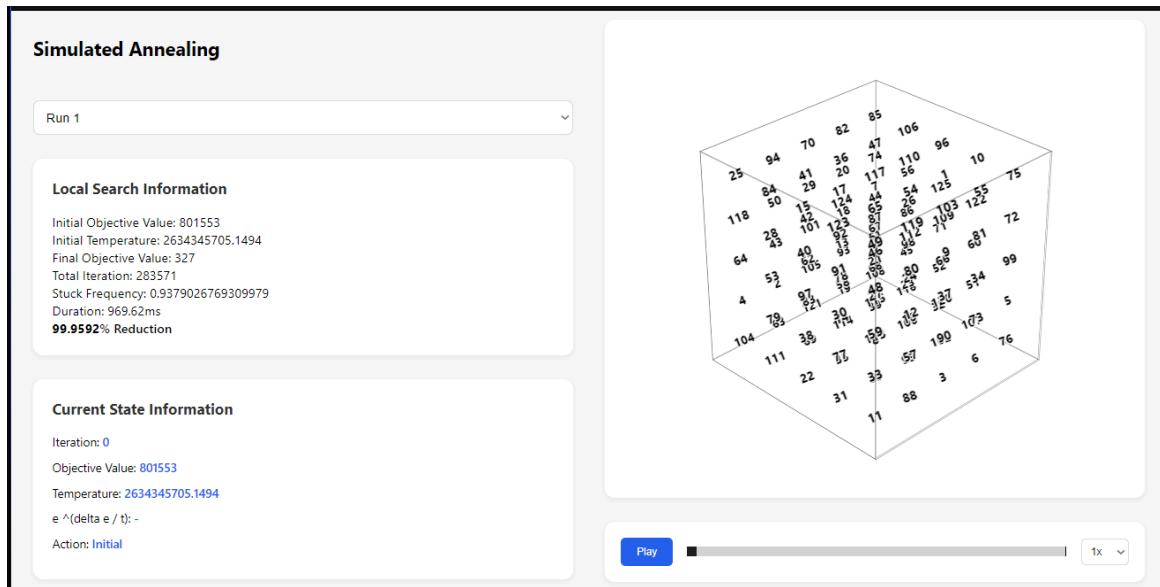


Gambar 36. Plot *Objective Function* Uji Coba ke-3 *Random Restart Hill Climbing*

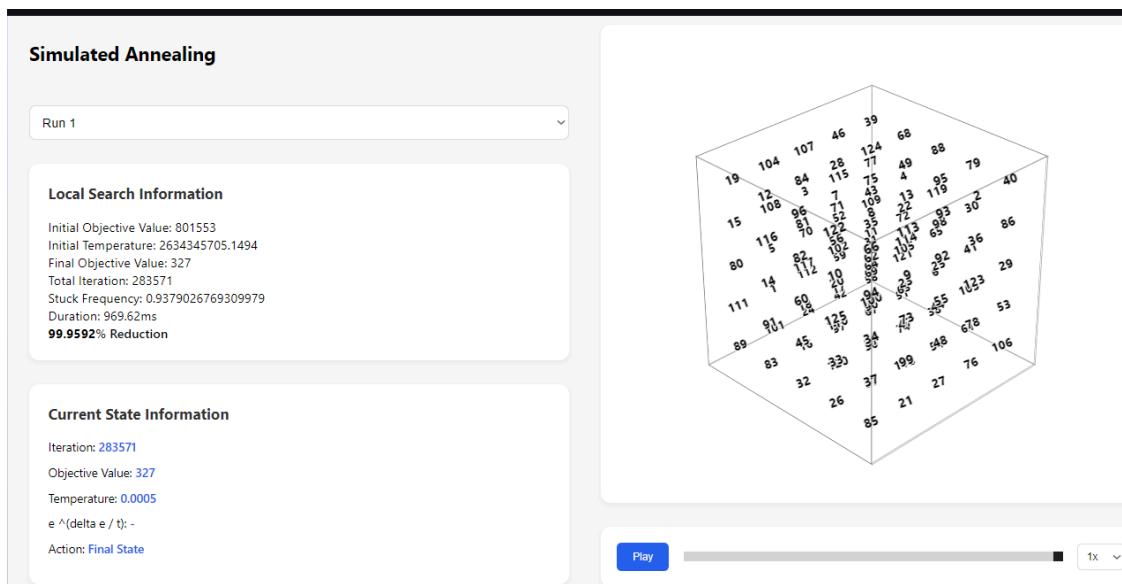
Berdasarkan hasil algoritma dari *Random Restart Hill Climbing*, dibuat plot yang menampilkan penyebaran dari nilai *Objective Function* sepanjang program berjalan. Dapat dilihat pada grafik, terjadi penurunan *Objective Function* di awal, lalu naik secara signifikan, kemudian menurun lagi. Pola ini terjadi sebanyak 3 kali. Pola ini menunjukkan bahwa algoritma melakukan pengulangan pencarian sebanyak 3 kali dengan *state* awal kubus yang berbeda-beda untuk menemukan solusi yang terbaik.

## 5. Simulated Annealing

### • Percobaan ke-1



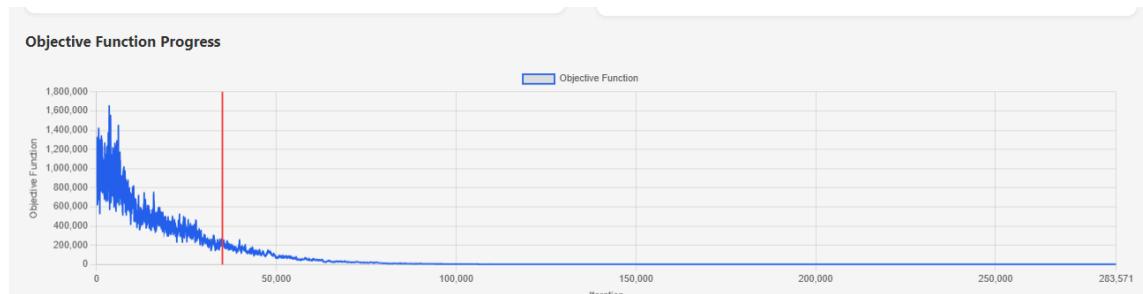
Gambar 37. State Awal Uji Coba ke-1 Simulated Annealing



Gambar 38. State Akhir Uji Coba ke-1 Simulated Annealing

Berdasarkan hasil uji coba pertama algoritma *Simulated Annealing*, didapatkan hasil seperti gambar di atas. Algoritma diawali dengan inisialisasi state. Pada saat itu *objective function* kubus bernilai **801.553**. Temperaturnya bernilai **2.634.345.705,1494**. Setelah algoritma dijalankan, didapati hasil akhir *objective function*-nya adalah **327**. Algoritma berjalan selama **969,62 ms** dengan iterasi sebanyak **283.571**.

Algoritma *Simulated Annealing* dapat menghindari keadaan *stuck* (tidak dapat bergerak mendekati solusi) dengan memperbolehkan pergerakan ke successor dengan konfigurasi yang menjauhi solusi selama probabilitasnya sesuai. Oleh karena itu, selama keberlangsungan algoritma, ada waktu kubus masuk ke dalam *state* dimana kubus terjebak dan perlu mengambil langkah yang lebih buruk. Pada algoritma ini, frekuensi peristiwa itu terjadi sebesar **0,9379026769309979** (menggunakan rumus  $1 - (\text{total gerakan}/\text{total iterasi})$ ). Ini berarti algoritma sering menjumpai keadaan *stuck*.



Gambar 39. Plot *Objective Function* Uji Coba ke-1 *Simulated Annealing*

Plot di atas merupakan gambaran dari persebaran *objective value* kubus di tiap iterasi. Dapat dilihat bentuk grafiknya mengalami naik turun terus menerus. Hal ini menunjukkan, bahwa algoritma *Simulated Annealing* memang dapat bergerak ke *state* dengan *heuristic value* yang lebih jauh dari solusi daripada *current state*. Hal ini dilakukan untuk menghindari program terjebak di *local optimum*.

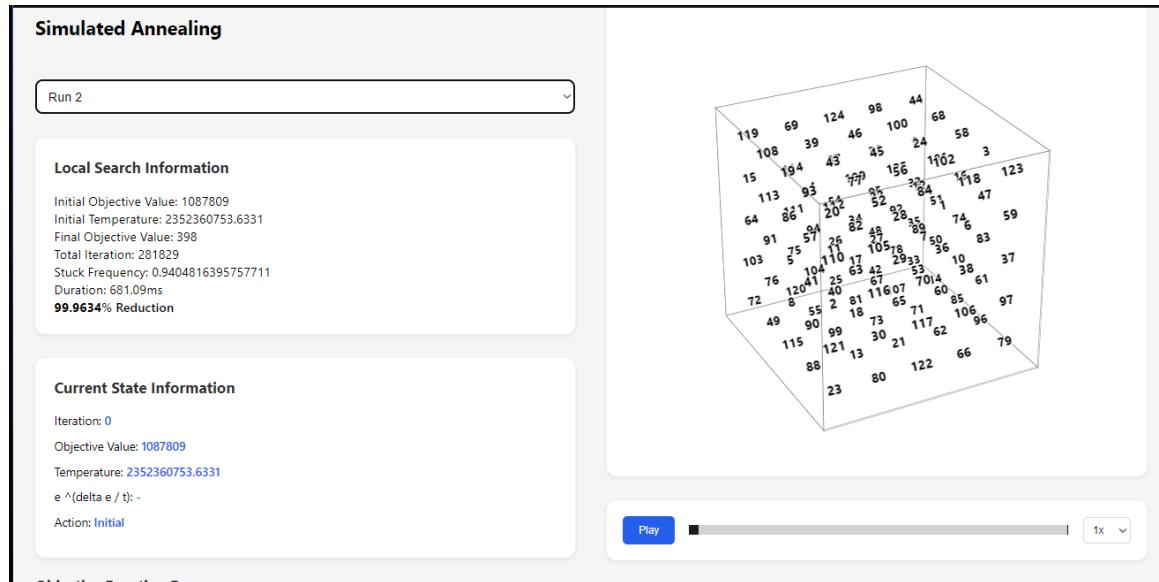
Dapat diperhatikan juga bahwa semakin algoritma berjalan, naik turun grafiknya semakin berkurang dan di akhir menjadi stabil menurun terus. Hal ini terjadi karena setiap iterasi, variabel temperatur terus menurun. Variabel temperatur ini memengaruhi hasil perhitungan *probability*. Sehingga, semakin algoritma berjalan semakin jarang juga program untuk memilih bergerak ke *objective function* yang lebih tinggi dari *current state*.



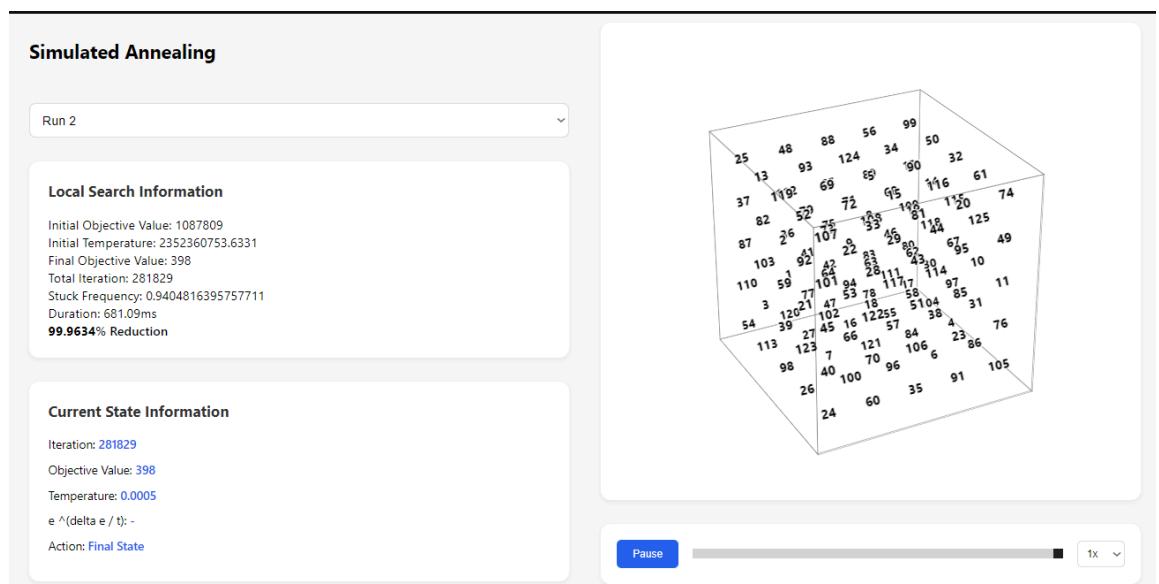
Gambar 40. Plot *Probability* Uji Coba ke-1 *Simulated Annealing*

Plot di atas merupakan gambaran persebaran *probability acceptance* selama algoritma berlangsung. Dapat dilihat semakin algoritma lama berjalan, semakin berkurang banyaknya kemungkinan untuk bergerak ke *objective function* yang lebih besar. Hal ini menunjukkan korelasi antara temperatur dengan *probability*.

### ● Percobaan ke-2



Gambar 41. State Awal Uji Coba ke-2 Simulated Annealing

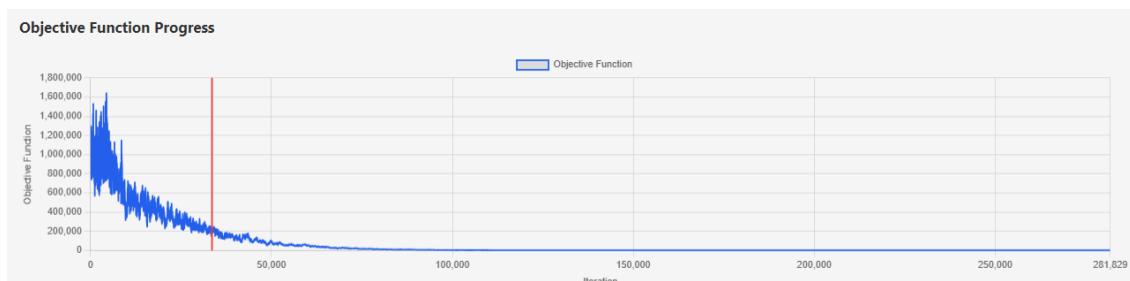


Gambar 42. State Akhir Uji Coba ke-2 Simulated Annealing

Berdasarkan hasil uji coba kedua algoritma *Simulated Annealing*, didapatkan hasil seperti gambar di atas. Algoritma diawali dengan inisialisasi state. Pada saat itu *objective function* kubus bernilai **1.087.809**. Temperaturnya bernilai **2.352.360.753,6331**. Setelah

algoritma dijalankan, didapati hasil akhir *objective function*-nya adalah **398**. Algoritma berjalan selama **681,09 ms** dengan iterasi sebanyak **281.829**.

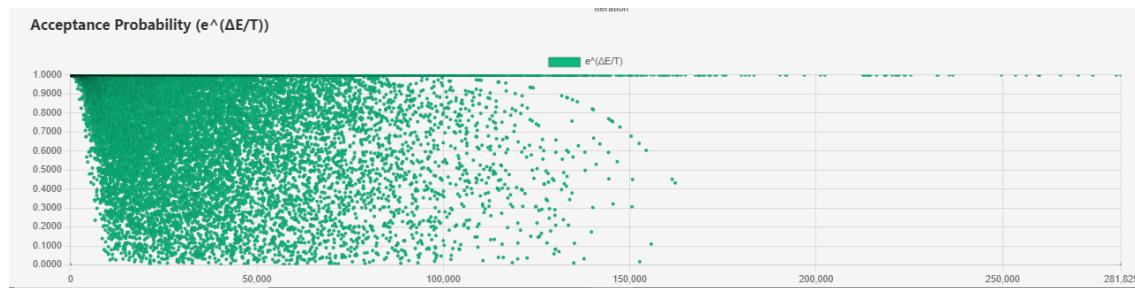
Algoritma *Simulated Annealing* dapat menghindari keadaan *stuck* (tidak dapat bergerak mendekati solusi) dengan memperbolehkan pergerakan ke successor dengan konfigurasi yang menjauhi solusi selama probabilitasnya sesuai. Oleh karena itu, selama keberlangsungan algoritma, ada waktu kubus masuk ke dalam *state* dimana kubus terjebak dan perlu mengambil langkah yang lebih buruk. Pada algoritma ini, frekuensi peristiwa itu terjadi sebesar **0,9404816395757711** (menggunakan rumus  $1 - (\text{total gerakan}/\text{total iterasi})$ ). Ini berarti algoritma sering menjumpai keadaan *stuck*.



Gambar 43. Plot *Objective Function* Uji Coba ke-2 *Simulated Annealing*

Plot di atas merupakan gambaran dari persebaran *objective value* kubus di tiap iterasi. Dapat dilihat bentuk grafiknya mengalami naik turun terus menerus. Hal ini menunjukkan, bahwa algoritma *Simulated Annealing* memang dapat bergerak ke *state* dengan *heuristic value* yang lebih jauh dari solusi daripada *current state*. Hal ini dilakukan untuk menghindari program terjebak di *local optimum*.

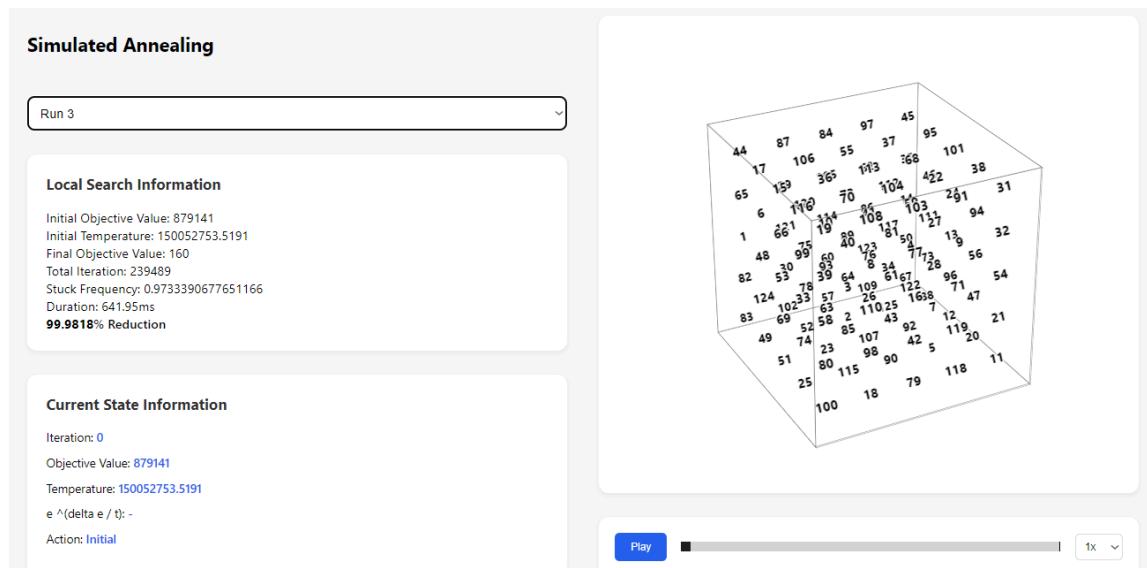
Dapat diperhatikan juga bahwa semakin algoritma berjalan, naik turun grafiknya semakin berkurang dan di akhir menjadi stabil menurun terus. Hal ini terjadi karena setiap iterasi, variabel temperatur terus menurun. Variabel temperatur ini memengaruhi hasil perhitungan *probability*. Sehingga, semakin algoritma berjalan semakin jarang juga program untuk memilih bergerak ke *objective function* yang lebih tinggi dari *current state*.



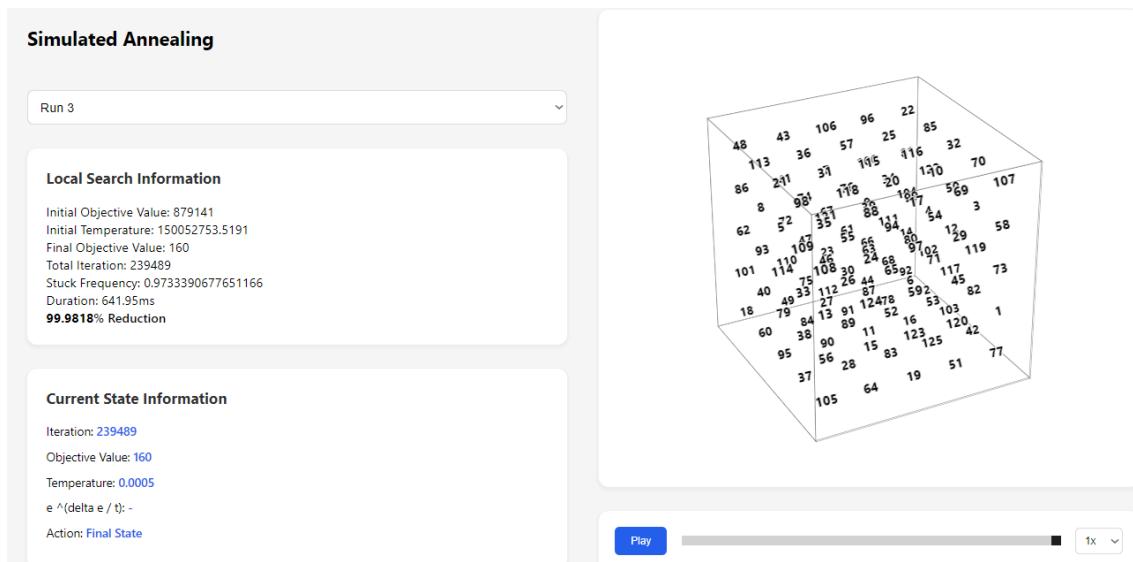
Gambar 44. Plot *Probability* Uji Coba ke-2 *Simulated Annealing*

Plot di atas merupakan gambaran persebaran *probability acceptance* selama algoritma berlangsung. Dapat dilihat semakin algoritma lama berjalan, semakin berkurang banyaknya kemungkinan untuk bergerak ke *objective function* yang lebih besar. Hal ini menunjukkan korelasi antara temperatur dengan *probability*.

- **Percobaan ke-3**



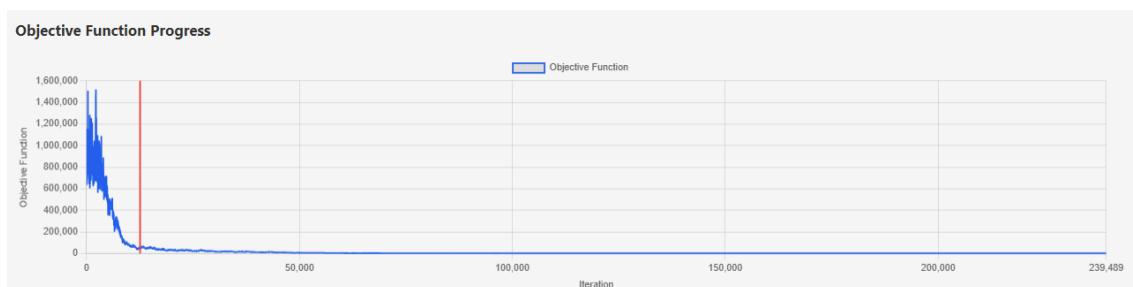
Gambar 45. State Awal Uji Coba ke-3 Simulated Annealing



Gambar 46. State Akhir Uji Coba ke-3 Simulated Annealing

Berdasarkan hasil uji coba ketiga algoritma *Simulated Annealing*, didapatkan hasil seperti gambar di atas. Algoritma diawali dengan inisialisasi state. Pada saat itu *objective function* kubus bernilai **879,141**. Temperaturnya bernilai **150.052.753,5191**. Setelah algoritma dijalankan, didapati hasil akhir *objective function*-nya adalah **160**. Algoritma berjalan selama **641.95 ms** dengan iterasi sebanyak **239.489**.

Algoritma *Simulated Annealing* dapat menghindari keadaan *stuck* (tidak dapat bergerak mendekati solusi) dengan memperbolehkan pergerakan ke successor dengan konfigurasi yang menjauhi solusi selama probabilitasnya sesuai. Oleh karena itu, selama keberlangsungan algoritma, ada waktu kubus masuk ke dalam *state* dimana kubus terjebak dan perlu mengambil langkah yang lebih buruk. Pada algoritma ini, frekuensi peristiwa itu terjadi sebesar **0.9733390677651166** (menggunakan rumus  $1 - (\text{total gerakan}/\text{total iterasi})$ ). Ini berarti algoritma sering menjumpai keadaan *stuck*.

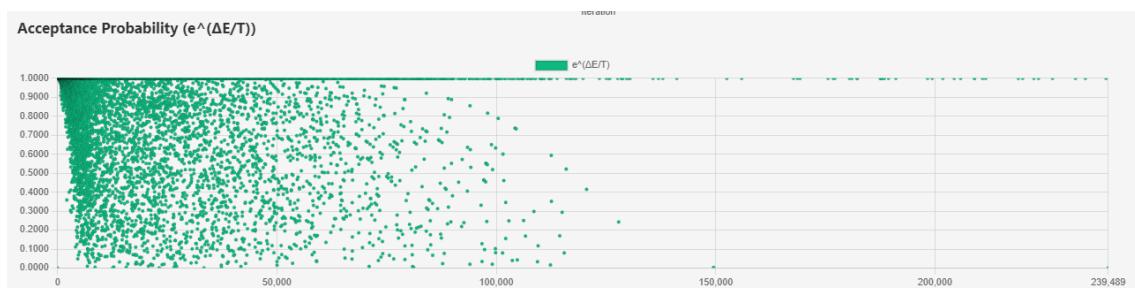


Gambar 47. Plot Objective Function Uji Coba ke-3 Simulated Annealing

Plot di atas merupakan gambaran dari persebaran *objective value* kubus di tiap iterasi. Dapat dilihat bentuk grafiknya mengalami naik turun terus menerus. Hal ini menunjukkan,

bahwa algoritma *Simulated Annealing* memang dapat bergerak ke *state* dengan *heuristic value* yang lebih jauh dari solusi daripada *current state*. Hal ini dilakukan untuk menghindari program terjebak di *local optimum*.

Dapat diperhatikan juga bahwa semakin algoritma berjalan, naik turun grafiknya semakin berkurang dan di akhir menjadi stabil menurun terus. Hal ini terjadi karena setiap iterasi, variabel temperatur terus menurun. Variabel temperatur ini memengaruhi hasil perhitungan *probability*. Sehingga, semakin algoritma berjalan semakin jarang juga program untuk memilih bergerak ke *objective function* yang lebih tinggi dari *current state*.

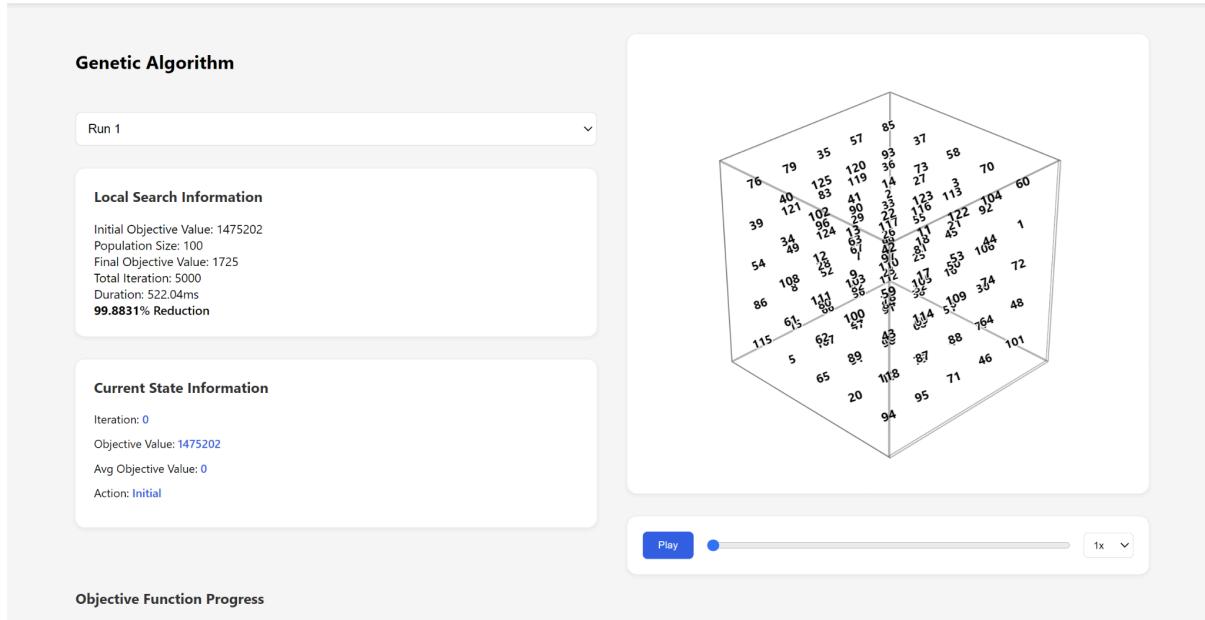


Gambar 48. Plot *Probability* Uji Coba ke-3 *Simulated Annealing*

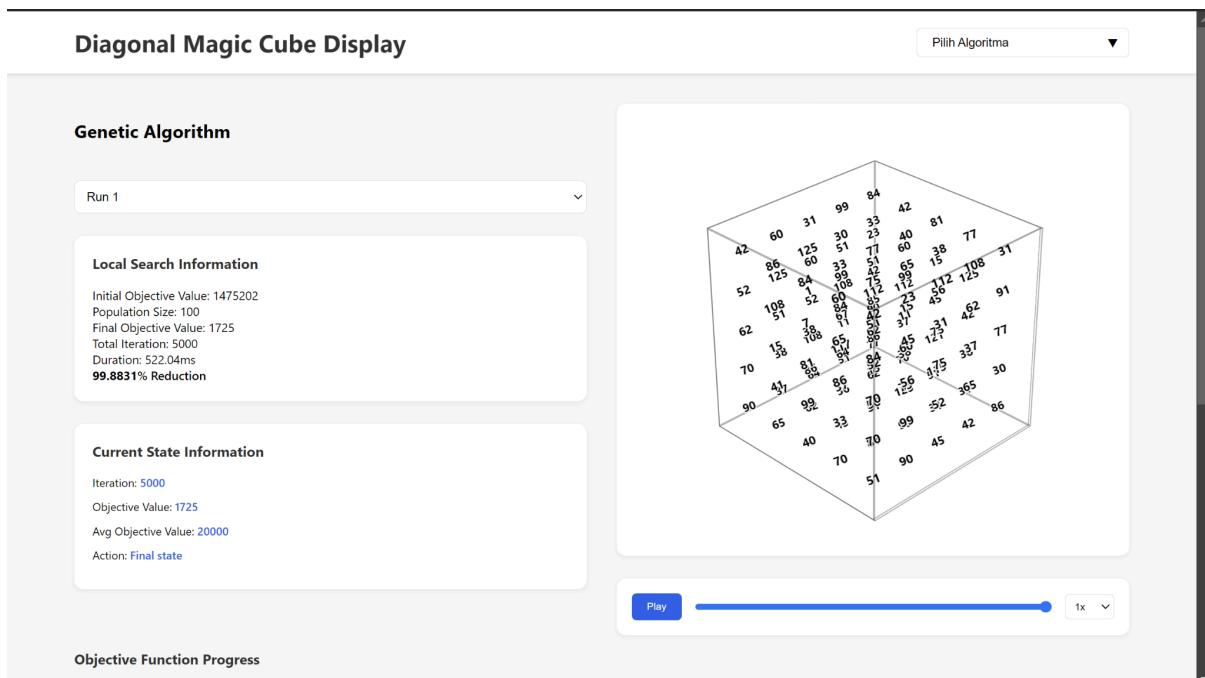
Plot di atas merupakan gambaran persebaran *probability acceptance* selama algoritma berlangsung. Dapat dilihat semakin algoritma lama berjalan, semakin berkurang banyaknya kemungkinan untuk bergerak ke *objective function* yang lebih besar. Hal ini menunjukkan korelasi antara temperatur dengan *probability*.

## 6. Genetic Algorithm

### • Percobaan Ke - 1



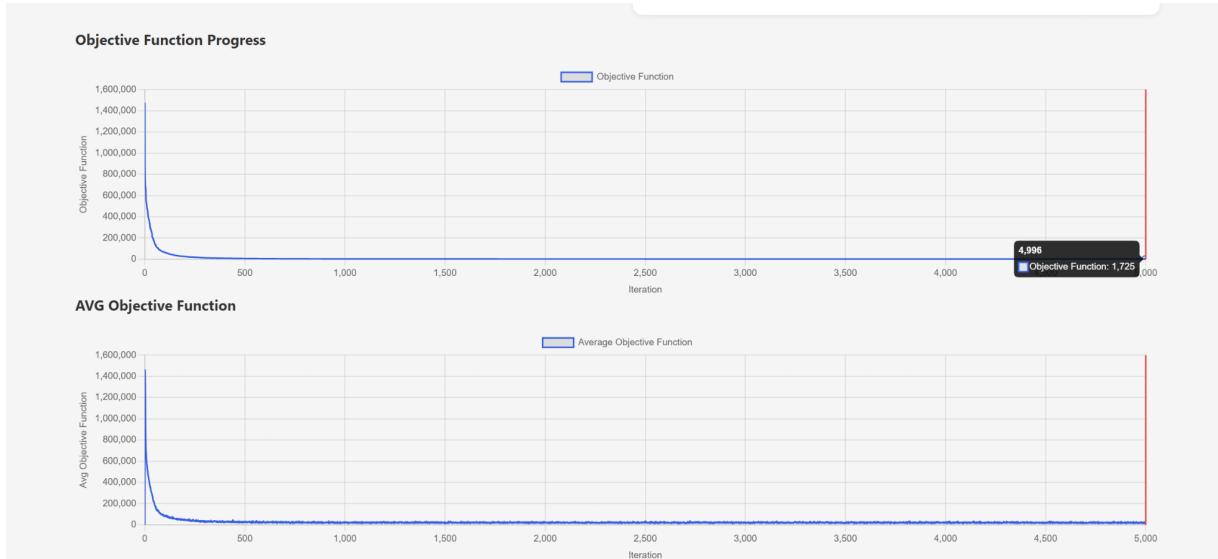
Gambar 44. State Awal Uji Coba Ke - 1 Genetic Algorithm



Gambar 45. State Akhir Uji Coba Ke - 1 Genetic Algorithm

Percobaan pertama *Genetic Algorithm* menghasilkan reduksi >99% dengan *Initial Objective Value* sebesar 1475202 dan *Final Objective Value* sebesar 1725. Percobaan dilakukan dengan jumlah populasi 100, iterasi sejumlah 5000, dan dilakukan durasi

percobaan adalah 522.04ms. Walaupun iterasi yang banyak, durasi ini relatif lebih kecil daripada algoritma lain, hal ini karena *Genetic Algorithm* mengambil sejumlah individu dengan objective value yang optimal, dan menghasilkan keturunan yang lebih optimal, tanpa menambah jumlah populasi. Berikut plot nilai *objective function* sepanjang iterasi selama algoritma berlangsung.

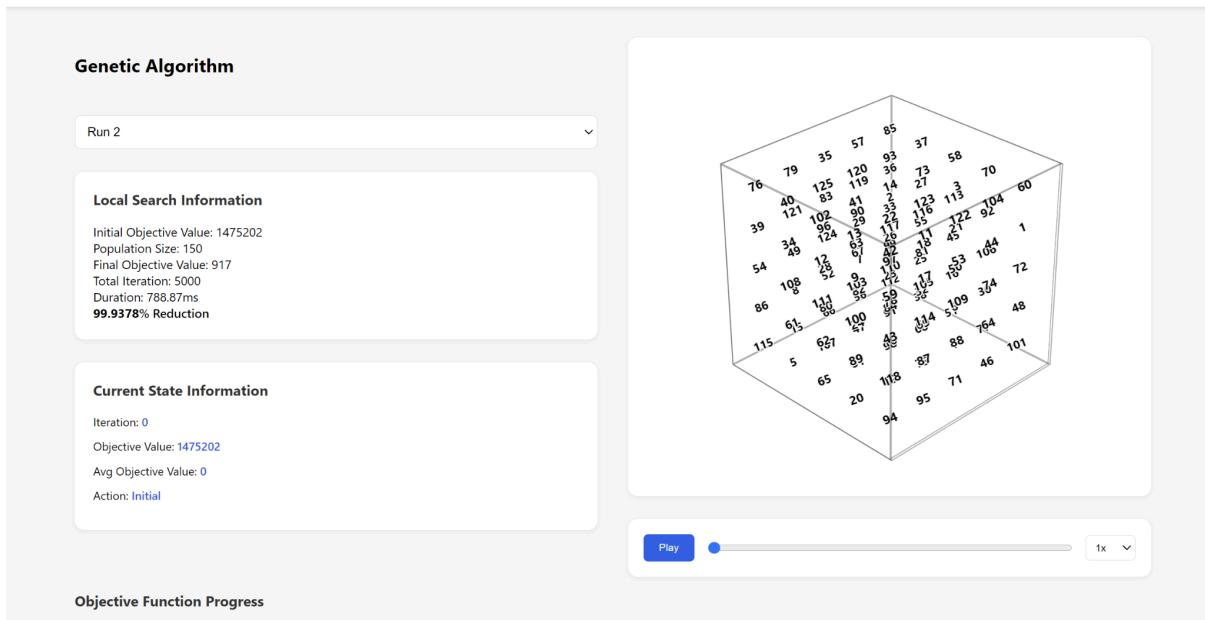


Gambar 46. Plot *Objective Function* Uji Coba ke-1 *Genetic Algorithm*

Dapat dilihat bahwa *Objective Value* menurun drastis pada iterasi 1~200. Namun, setelah iterasi 200 ke atas, *Objective Value* tidak menurun sedrastis itu.

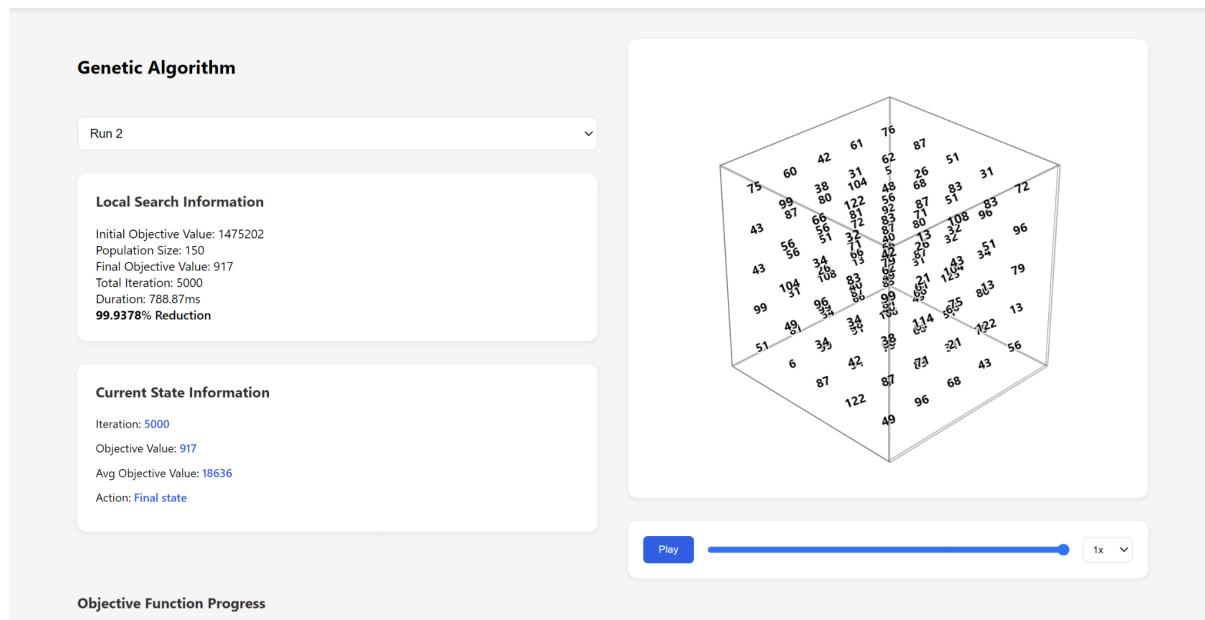
## ● Percobaan Ke - 2

### Diagonal Magic Cube Display



Gambar 47. State Awal Uji Coba Ke - 2 Genetic Algorithm

### Diagonal Magic Cube Display



Gambar 48. State Akhir Uji Coba Ke - 2 Genetic Algorithm

Percobaan kedua *Genetic Algorithm* menghasilkan reduksi >99% dengan *Initial Objective Value* sebesar 1475202 dan *Final Objective Value* sebesar 1031. Percobaan dilakukan dengan jumlah populasi 150, iterasi sejumlah 5000, dan dilakukan durasi

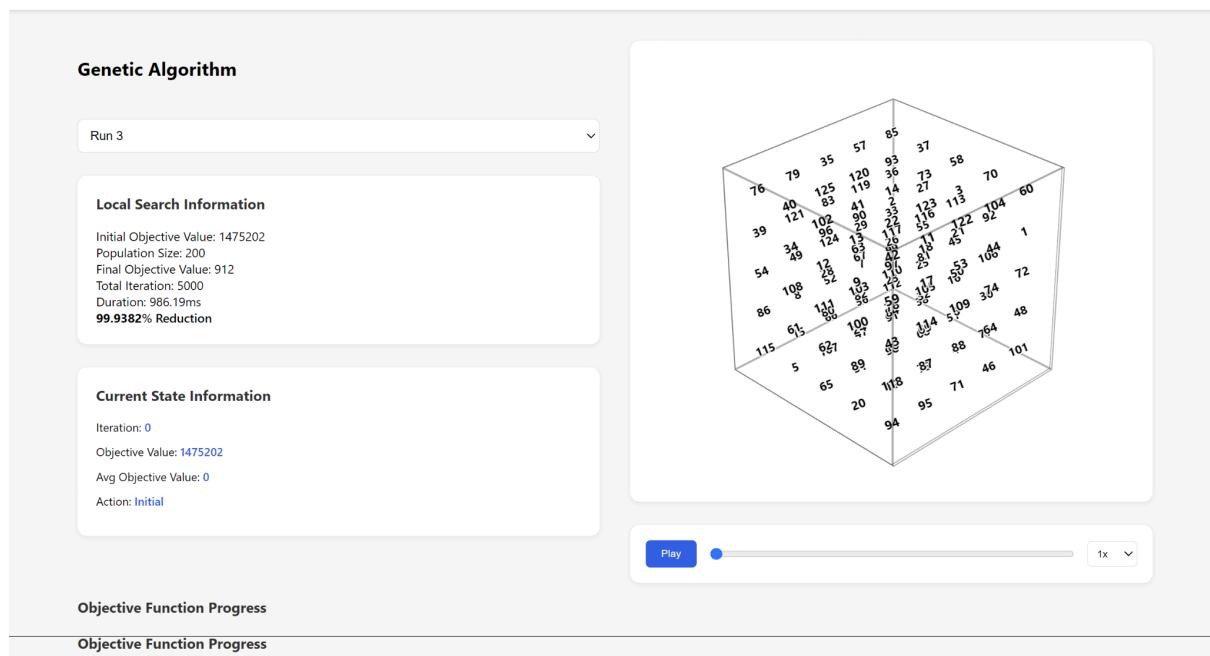
percobaan adalah 1.47s. Berikut plot nilai *objective function* sepanjang iterasi selama algoritma berlangsung.



Gambar 49. Plot *Objective Function* Uji Coba ke-2 *Genetic Algorithm*

### ● Percobaan Ke - 3

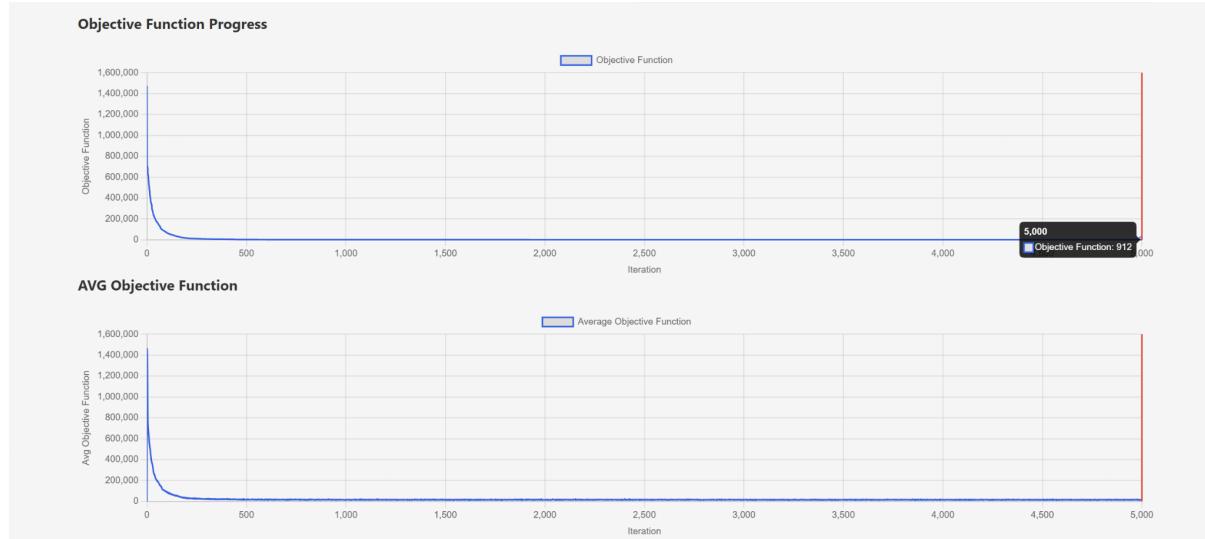
#### Diagonal Magic Cube Display



Gambar 51. State Akhir Uji Coba Ke - 3 *Genetic Algorithm*

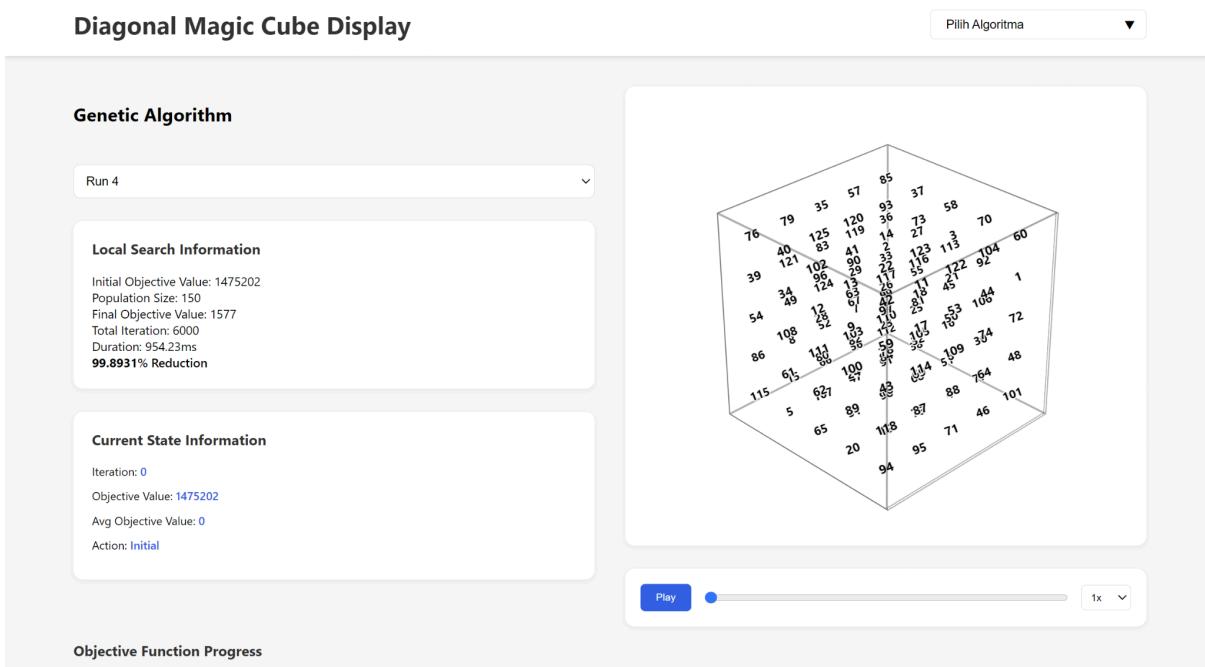
Percobaan ketiga *Genetic Algorithm* menghasilkan reduksi >99% dengan *Initial Objective Value* sebesar 1475202 dan *Final Objective Value* sebesar 912. Percobaan

dilakukan dengan jumlah populasi 200, iterasi sejumlah 5000, dan dilakukan durasi percobaan adalah 986.19ms. Berikut plot nilai *objective function* sepanjang iterasi selama algoritma berlangsung.

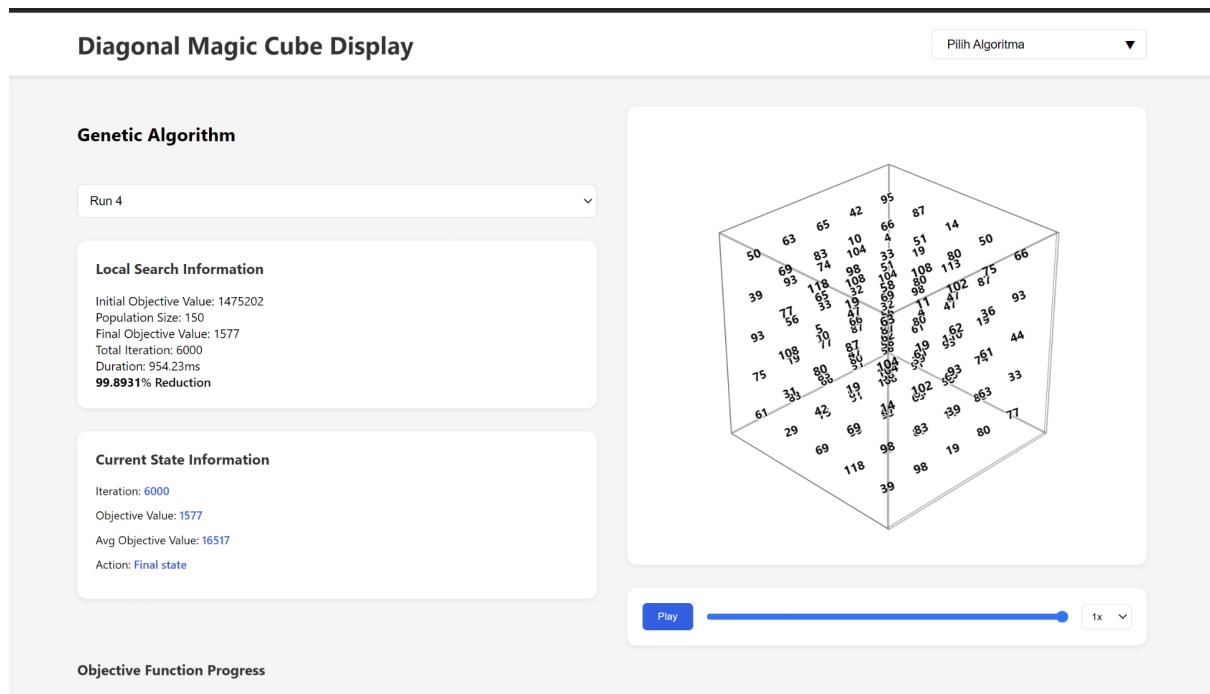


Gambar 52. Plot *Objective Function* Uji Coba ke-3 *Genetic Algorithm*

### ● Percobaan Ke - 4

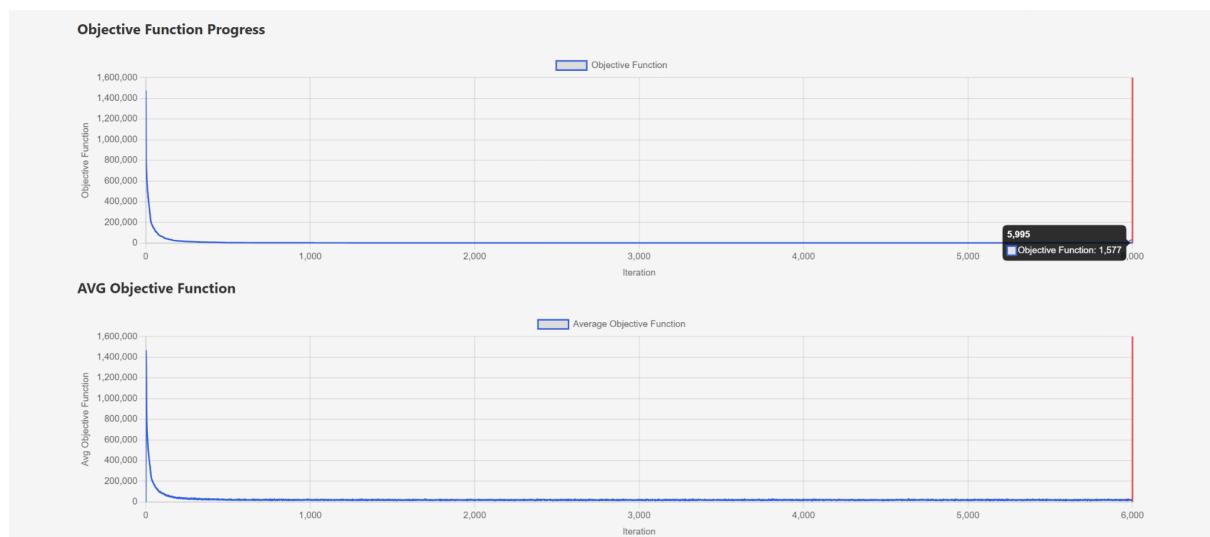


Gambar 53. State Awal Uji Coba Ke - 4 *Genetic Algorithm*



Gambar 54. State Akhir Uji Coba Ke - 4 Genetic Algorithm

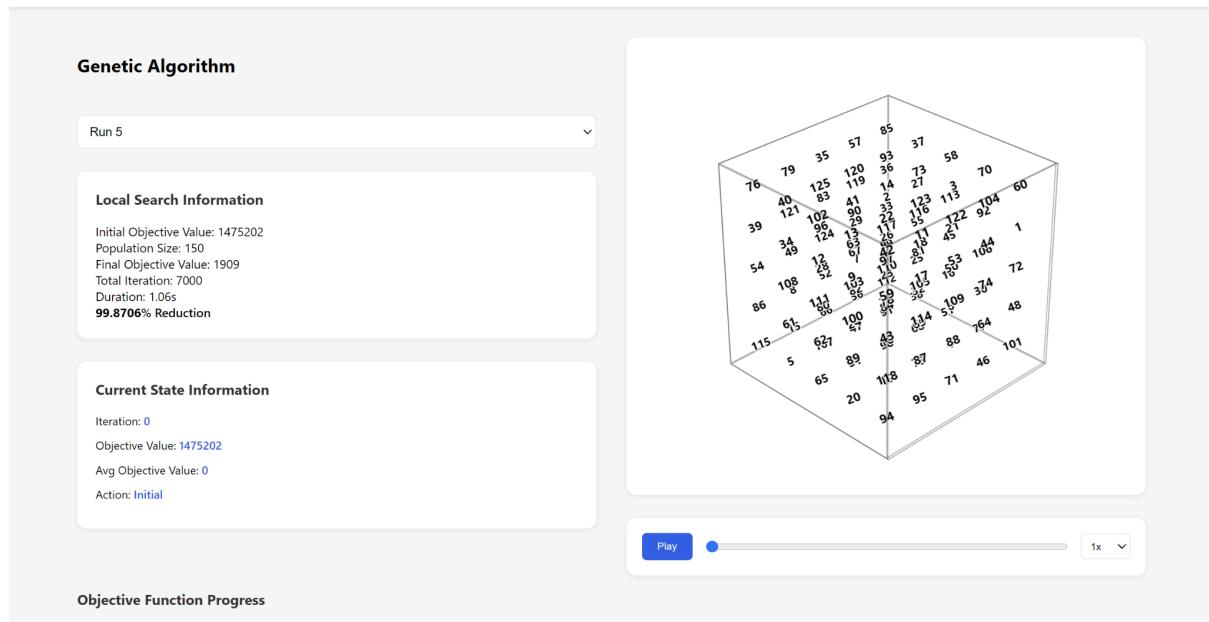
Percobaan keempat *Genetic Algorithm* menghasilkan reduksi >99% dengan *Initial Objective Value* sebesar 1475202 dan *Final Objective Value* sebesar 1577. Percobaan dilakukan dengan jumlah populasi 150, iterasi sejumlah 6000, dan dilakukan durasi percobaan adalah 954.23ms. Berikut plot nilai *objective function* sepanjang iterasi selama algoritma berlangsung.



Gambar 55. Plot *Objective Function* Uji Coba ke-4 *Genetic Algorithm*

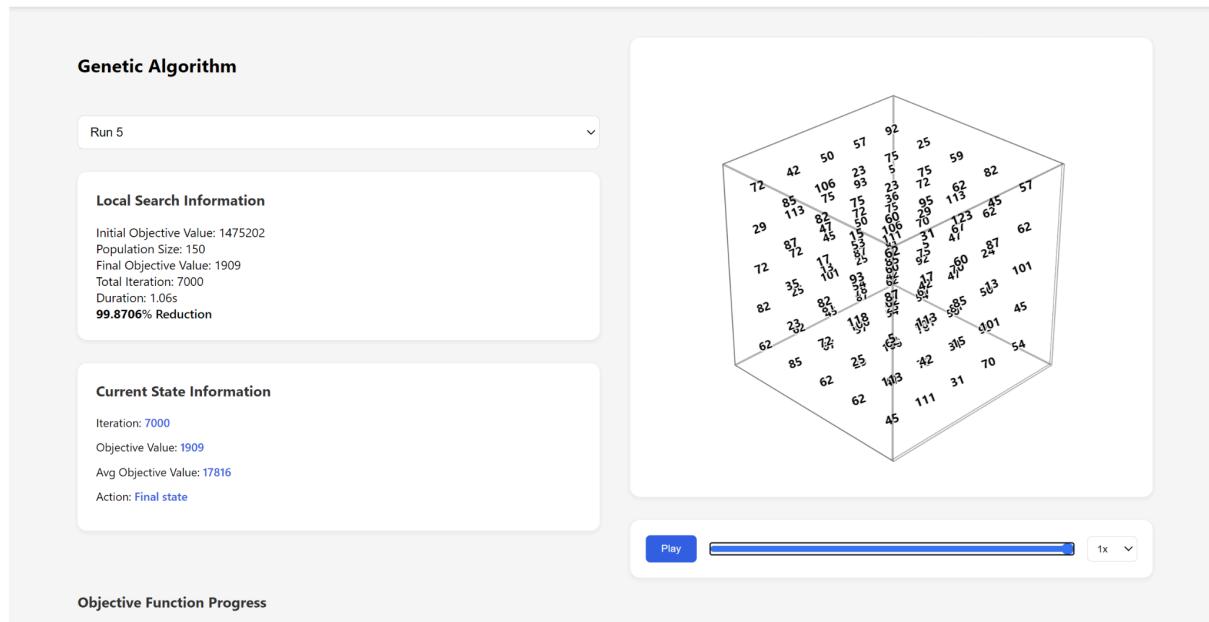
## ● Percobaan Ke - 5

### Diagonal Magic Cube Display



Gambar 56. State Awal Uji Coba Ke - 5 Genetic Algorithm

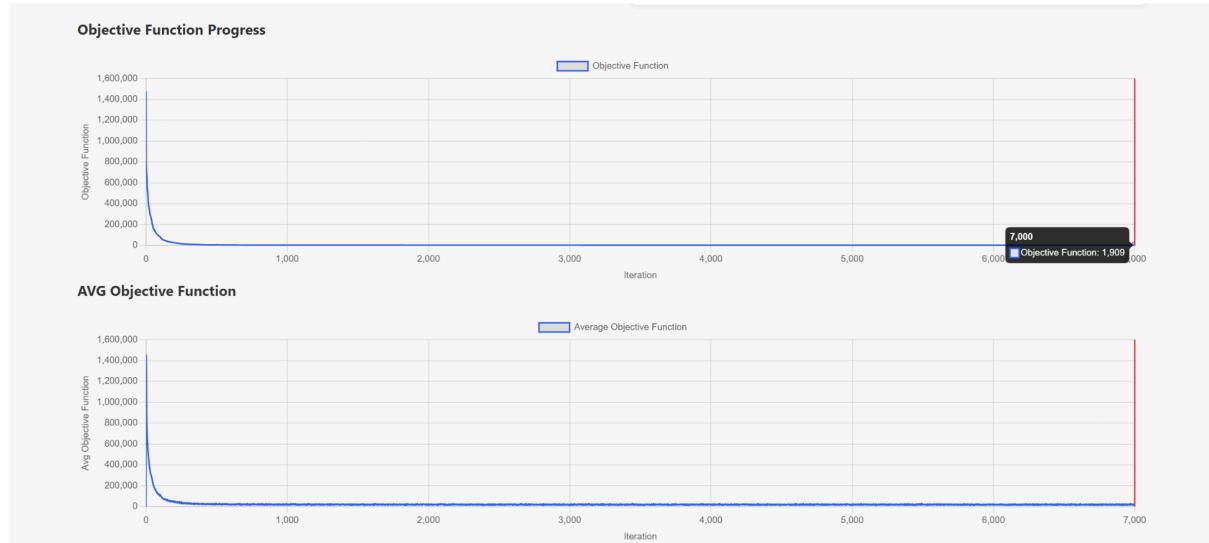
### Diagonal Magic Cube Display



Gambar 57. State Akhir Uji Coba Ke - 5 Genetic Algorithm

Percobaan kelima *Genetic Algorithm* menghasilkan reduksi >99% dengan *Initial Objective Value* sebesar 1475202 dan *Final Objective Value* sebesar 1909. Percobaan dilakukan dengan jumlah populasi 150, iterasi sejumlah 7000, dan dilakukan durasi

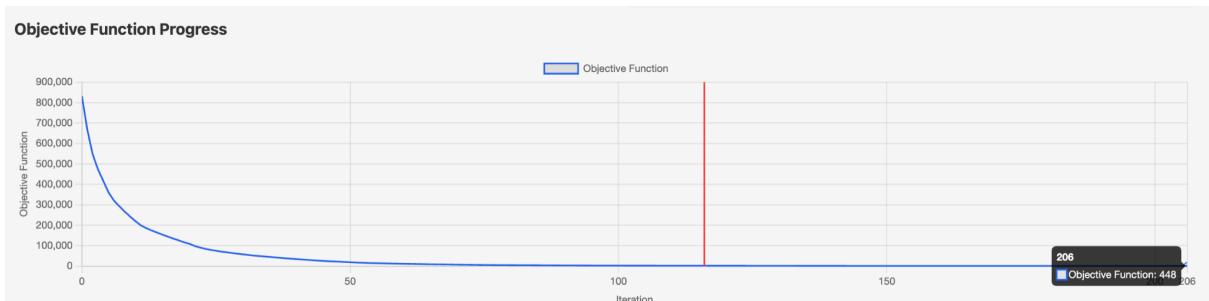
percobaan adalah 1.06s. Berikut plot nilai *objective function* sepanjang iterasi selama algoritma berlangsung.



Gambar 58. Plot *Objective Function* Uji Coba ke-5 *Genetic Algorithm*

## 2. Analisis

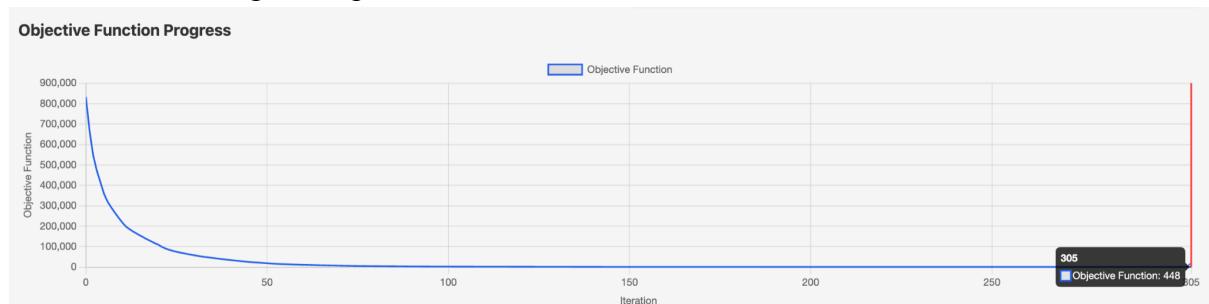
Berdasarkan hasil percobaan yang telah dilakukan pada setiap algoritma *local search*, dapat dilakukan sebuah analisa dari hasil *value objective function* yang didapatkan oleh setiap algoritma berdasarkan total durasi yang dilakukan pada setiap iterasi, dimana masing-masing algoritma *local search* menghasilkan nilai *objective function* yang berbeda juga. Apabila dilihat pada Algoritma *Steepest Ascent Hill-Climbing Search*, rata-rata nilai *objective function* yang diperoleh ialah senilai **1.446,33**. Selama tiga kali percobaan yang dilakukan oleh algoritma *local search* tersebut, durasi yang dihasilkan pun juga terdapat perbedaan yang signifikan dimana rata-rata durasi yang dibutuhkan oleh algoritma *local search* untuk sampai pada titik *final state*, dibutuhkan durasi sebesar **2,23 seconds** dengan total durasi yang paling efisien dicapai pada saat percobaan kedua dilakukan yakni **1,64 seconds**. Berikut merupakan kondisi algoritma *Steepest Ascent Hill-Climbing Search* mendekati kondisi global optima.



Gambar 59. Kondisi *Best Local Optimum Steepest Ascent Hill-Climbing*

Apabila dilihat pada grafik plot tersebut, hasil *final state* menunjukkan nilai *objective function* yang paling optimal dan mendekati global optima adalah senilai 448 dimana nilai *objective function* tersebut apabila dibandingkan dengan *initial state* di awal memiliki perbedaan yang sangat jauh sehingga dapat dinyatakan bahwa percobaan kedua pada Algoritma *Steepest Ascent Hill-Climbing* memiliki hasil *objective function* yang paling efisien dibandingkan uji coba ke-1 ataupun ke-2. Hal tersebut disebabkan karena prinsip utama dari *Steepest Ascent Hill-Climbing Search* adalah untuk mencari *successor* terbaik dari perbandingan *neighbor* yang ada pada setiap pengecekan. Selain itu, tingkat efektivitas Algoritma *Steepest Ascent Hill-Climbing* dalam memilih tetangga yang paling optimal juga mempercepat proses pencarian solusi yang optimal.

Jika dilihat pada hasil percobaan *Hill-Climbing Search With Sideways Moves*, rata-rata nilai *objective function* yang didapatkan dari hasil uji coba tersebut ialah sebesar **1.446,33** dimana hasil rata-rata dapat dikatakan sama dengan hasil uji coba pada algoritma *Steepest Ascent Hill-Climbing Search*. Rata-rata dari total durasi yang dibutuhkan oleh algoritma untuk dapat mencapai titik global optima tersebut ialah sebesar **3,03 seconds** dengan total durasi yang paling cepat berada pada percobaan kedua, yakni hanya sekitar **2.45 seconds**. Berikut merupakan kondisi Algoritma *Hill-Climbing Search With Sideways Moves* mendekati kondisi global optima.

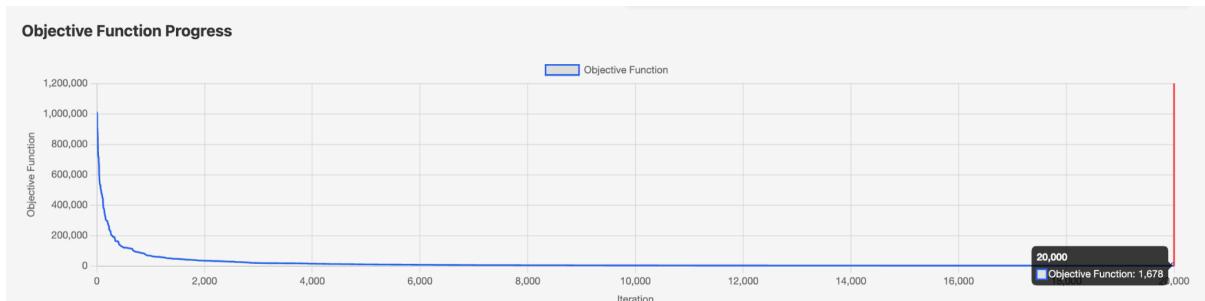


Gambar 60. Kondisi *Best Local Optimum Hill-Climbing Search With Sideways Moves*

Apabila dilihat pada grafik plot tersebut, dapat dilihat bahwa hasil *final state* pada percobaan *Hill-Climbing Search With Sideways Moves* menunjukkan nilai *objective function* sebesar 448 pada iterasi ke-305 dimana nilai *objective function* tersebut apabila dibandingkan dengan *initial state* di awal percobaan algoritma *local search*, tentunya memiliki perbedaan yang sangat jauh sehingga dapat dinyatakan bahwa percobaan kedua pada Algoritma *Hill Climbing With Sideways Moves* memiliki hasil *objective function* yang paling efisien dibandingkan uji coba ke-1 ataupun ke-2. Hal tersebut disebabkan oleh beberapa faktor dimana salah satunya karena adanya penggunaan *Sideways Moves* yang memberikan proses

pengecekan *successor* menjadi lebih efisien untuk algoritma *local search* tersebut. Adanya batasan jumlah *Sideways Moves* yang diperbolehkan selama proses iterasi dapat membuat algoritma *local search* untuk dapat menghindari jebakan *local maximum* sehingga proses iterasi dapat berlanjut meskipun tidak terdapat peningkatan nilai *objective function* tersebut, maka dari itu algoritma dalam melakukan proses pengecekan *state neighbor* lainnya yang memiliki *value* lebih baik daripada *state* awal *cube* sehingga proses tersebut sangat mendukung tingkat efisiensi algoritma *local search* untuk mencapai kondisi global optima.

Jika dilihat pada sisi percobaan *Stochastic Hill-Climbing Search*, hasil percobaan yang telah dilakukan memiliki rata-rata nilai *objective function* yang didapatkan pada pada percobaan tersebut adalah **2.752** dimana hasil rata-rata dapat dikatakan cenderung lebih besar dibandingkan hasil uji coba *Steepest Ascent Hill-Climbing* ataupun dengan *Hill-Climbing With Sideways Moves*. Jumlah batas maksimum iterasi yang dapat dilakukan algoritma adalah sebesar 20.000 kali. Rata-rata durasi yang dibutuhkan oleh algoritma untuk mencapai titik global optima ialah sebesar **24,73 seconds** dimana tidak terdapat perbedaan yang cukup jauh kepada masing-masing percobaan yang dilakukan pada algoritma *Stochastic Hill-Climbing* tersebut, namun di antara ketiga percobaan tersebut yang memiliki waktu durasi pencarian paling efisien dan cepat ialah percobaan kedua dengan total durasi **24,57 seconds**. Berikut merupakan kondisi Algoritma *Stochastic Hill-Climbing* mendekati kondisi global optima.

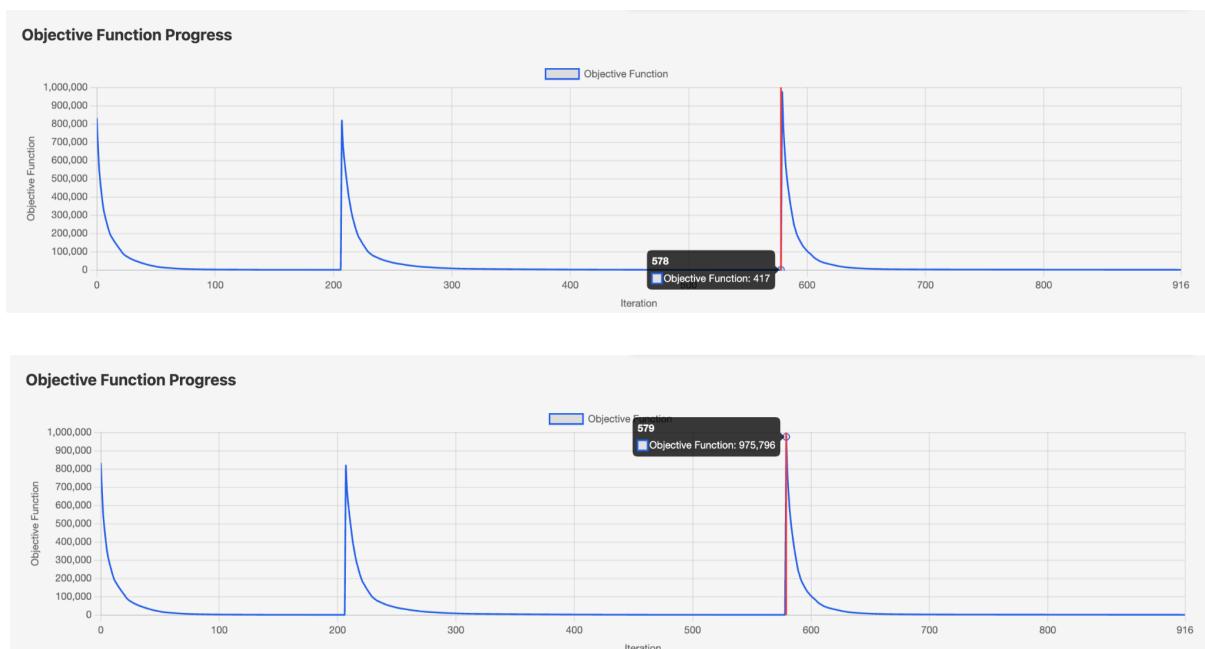


Gambar 61. Kondisi *Best Local Optimum Stochastic Hill-Climbing*

Apabila dilihat pada grafik plot tersebut, dapat dilihat bahwa hasil *final state* pada percobaan *Stochastic Hill-Climbing* menunjukkan nilai *objective function* sebesar **1.678** pada iterasi ke-20.000 dimana nilai *objective function* tersebut apabila dibandingkan dengan *initial state* di awal percobaan algoritma *local search*, tentunya memiliki perbedaan yang sangat jauh sehingga dapat dinyatakan bahwa percobaan pertama pada Algoritma *Stochastic Hill-Climbing* memiliki hasil *objective function* yang paling efisien dibandingkan uji coba ke-2 ataupun ke-3. Hal tersebut dapat terjadi karena beberapa pengaruh dari penerapan algoritma *Stochastic Hill-Climbing* yang melakukan pembangkitan *successor* secara acak

untuk setiap iterasi yang dilakukan. Pemilihan *successor* secara *random* tersebut juga memberikan peluang bagi algoritma *local search* untuk menghindari jebakan *local maximum* sehingga algoritma dapat terus melakukan proses pengecekan iterasi untuk menentukan *neighbor* dengan *value* terbaik. Selain itu, adanya batasan maksimum iterasi sebanyak 20.000 kali juga mendukung proses pencarian *successor* sehingga menjadi lebih mudah untuk algoritma dalam melakukan *successor* dengan *value* yang lebih baik.

Terakhir, pada *Random Restart Hill-Climbing* rata-rata nilai *objective function* yang didapatkan ialah **1.104,67** dengan nilai *objective function* paling efisien terletak pada percobaan kedua dengan best *objective value* yang didapatkan senilai 764. Rata-rata durasi yang dibutuhkan oleh *objective function* untuk melakukan pencarian *successor* adalah **8,11 seconds** dengan durasi yang paling cepat dan efisien terletak pada percobaan kedua dengan durasi sebesar **7,32 seconds**. Pada algoritma ini, diberikan batasan jumlah *restart* yang dapat dilakukan oleh algoritma *local search* sebanyak tiga kali. Berikut merupakan kondisi Algoritma *Random Restart Hill-Climbing Search* untuk mendekati titik global optima.

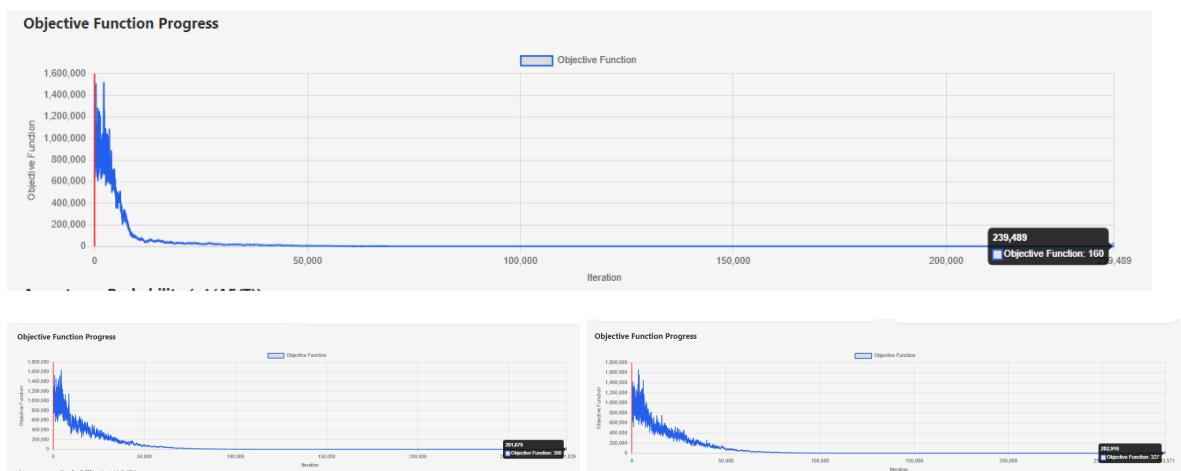


Gambar 62. Kondisi *Best Local Optimum & Restart* pada *Random Restart Hill Climbing*

Apabila dilihat pada grafik plot tersebut, dapat dilihat bahwa hasil *final state* pada percobaan *Random Restart Hill-Climbing* menunjukkan nilai *objective function* sebesar 417 pada iterasi ke-578 dimana nilai *objective function* tersebut merupakan *best objective function* dari hasil pencarian *successor* oleh algoritma tersebut. Namun apabila dilihat, pada iterasi ke-579 algoritma langsung melakukan *restart* ulang terhadap nilai *objective function* yang

relatif tidak lebih baik dari *state* sebelumnya. Adapun beberapa faktor yang menyebabkan terjadinya *random restart* tersebut seperti adanya nilai *objective function* yang cenderung lebih buruk ataupun sama dengan *state* awal yang dimiliki oleh *diagonal magic cube*, akan tetapi pada kasus seperti yang ada pada gambar, nilai *objective function* yang ditemukan tidak lebih baik dari *state* awal sehingga algoritma akan melakukan *random restart* untuk melakukan pengecekan *successor* dari awal kembali. Karena seringkali melakukan proses *random restart* tersebut, algoritma memiliki kesempatan lebih untuk tidak terjebak dalam kondisi *local maximum* sehingga algoritma dapat melakukan pencarian *objective function* dengan *value* yang paling optimal.

Pada percobaan algoritma *Simulated Annealing* ditemukan rata - rata dari *final objective value* nya adalah **295** dengan *final objective value* paling kecil mencapai angka **160**. Ini menunjukkan bahwa hasil dari algoritma *Simulated Annealing* sangat dekat dengan *global optimum*. Dibandingkan dengan algoritma lain, algoritma *Simulated Annealing* memiliki nilai *objective* yang paling mendekati *global optimum*. Rata - rata durasi berjalannya algoritma ini adalah **764.22 ms**, durasi tersebut jauh lebih cepat dibandingkan dengan algoritma lain. Hasil akhir yang didapat dari algoritma *Simulated Annealing* juga sangat konsisten. Perbedaan antara hasil akhir uji coba pertama, kedua, dan ketiga sangatlah kecil. Berikut adalah plot dari ketiga uji coba algoritma *Simulated Annealing*.



Gambar 63. Plot Uji Coba Algoritma *Simulated Annealing* Dengan *Final Objective Value*

Berdasarkan gambar di atas, dapat dilihat seberapa konsisten hasil akhir dari algoritma *Simulated Annealing*. Dibandingkan dengan algoritma lain yang telah diuji coba, algoritma *Simulated Annealing* dapat menghasilkan solusi yang lebih baik, lebih konsisten, dan lebih efisien.

Pada percobaan *Genetic Algorithm*, dilakukan 5 percobaan dengan 2 parameter berbeda, yaitu jumlah populasi dan jumlah iterasi. Percobaan pertama menjadi kontrol dengan populasi 100 dan iterasi 5000. Pada percobaan kedua dan ketiga, parameter populasi diubah sehingga percobaan kedua berjalan dengan populasi 150, dan percobaan ketiga dengan populasi 200. Sedangkan, pada percobaan keempat dan kelima, parameter iterasi diubah sehingga percobaan keempat berjalan selama 6000 iterasi, dan percobaan ketiga selama 7000 iterasi.

Berdasarkan hasil percobaan kedua dan ketiga, *Objective Value* masing-masing percobaan lebih optimal daripada kontrol, dengan percobaan kedua dengan *Objective Value* 917, dan percobaan ketiga dengan *Objective Value* 912. Dapat disimpulkan bahwa jumlah populasi dapat memengaruhi *Objective Value*, dengan lebih banyak populasi, lebih optimal (lebih rendah) *Objective Value*.

Sedangkan, hasil percobaan keempat dan kelima memiliki hasil yang berbeda. Hasil percobaan keempat memiliki *Objective Value* yang lebih optimal dengan nilai 1577, namun percobaan kelima menghasilkan *Objective Value* yang kurang optimal dengan nilai 1909. Dapat disimpulkan bahwa jumlah iterasi dapat tapi tidak selalu memengaruhi *Objective Value*, dan terdapat banyak faktor lain yang dapat memengaruhi *Objective Value*, seperti *state* kubus inisial, probabilitas mutasi, pemilihan *parents* ketika reproduksi, maupun *crossover point*.

Dari hasil analisis yang telah dilakukan terhadap masing-masing algoritma *local search*, dapat dinyatakan bahwa terdapat kelebihan dan kelemahan terhadap perbandingan masing-masing algoritma *local search* yang telah diuji tersebut. Pada hasil uji coba algoritma *Steepest Ascent Hill-Climbing Search* dan *Hill-Climbing Search With Sideways Moves* memberikan hasil nilai *objective function* yang cukup baik dan durasi yang relatif lebih efisien dengan total rata-rata durasi sebesar 2,23 *seconds* dan 3,03 *seconds*. Akan tetapi dari hasil kedua percobaan algoritma *local search* tersebut dapat dinyatakan bahwa keduanya dapat dengan mudah terjebak pada kondisi *local maximum*. Selain itu, bila dilihat pada hasil percobaan Stochastic Hill-Climbing Search memiliki durasi yang cenderung lebih lama dengan total rata-rata durasi sebesar 24,73 *seconds*. Namun, nilai *objective function* yang dihasilkan pada algoritma ini cenderung lebih tinggi dibandingkan dengan algoritma *local search* lainnya, yakni berkisar 1.678 yang disebabkan oleh adanya pembangkitan *successor* secara acak pada proses iterasi. *Random Restart Hill-Climbing Search* memiliki keunggulan dalam mengatasi kondisi jebakan kondisi *local maximum* dengan melakukan *random restart*, namun dengan cara ini algoritma *local search* cenderung membutuhkan total durasi waktu

yang lebih lama dibandingkan dengan algoritma *local search* lainnya. *Genetic Algorithm* memberikan kesempatan yang lebih besar untuk algoritma *local search*, namun memiliki batasan parameter tersendiri karena memiliki probabilitas mutasi, pemilihan parents, serta kondisi crossover points yang dilakukan oleh algoritma *local search* yang mengakibatkan hasil objective function yang dihasilkan cenderung tidak konsisten dan tidak optimal dibandingkan hasil *objective function* algoritma *local search* lainnya. Terakhir, pada hasil percobaan *Simulated Annealing* cenderung menghasilkan nilai *objective function* yang lebih optimal dibandingkan *local search* lainnya, yakni mencapai nilai **160** dan lebih efisien terhadap penggunaan durasi waktu yakni berkisar **0,4031 seconds**.

Berdasarkan analisis yang telah dilakukan terkait tiap-tiap eksperimen, dapat dilihat bahwa tidak semua hasil dari uji coba memberikan hasil akhir yang konsisten. Algoritma *local search Hill - Climbing* seperti *Steepest Ascent Hill-Climbing Search*, *Hill-Climbing Search with Sideway Moves*, *Stochastic Hill-Climbing Search*, dan *Random Restart Hill - Climbing* memiliki nilai akhir yang relatif tidak konsisten terutama pada *Stochastic Hill - Climbing*. Ketidakkonsistenan *Steepest Ascent Hill-Climbing Search*, *Hill-Climbing Search with Sideway Moves*, serta *Random Restart Hill - Climbing* terjadi karena algoritmanya yang sensitif pada konfigurasi, sehingga dengan sifatnya yang selalu mengambil jalur terbaik menyebabkan minim explorasi dan cepat terjebak di *local optimum*. Untuk *Stochastic Hill-Climbing*, hasil akhir semakin tidak konsisten karena *successor* diambil secara random (dapat dipengaruhi iterasi juga).

Dibandingkan dengan *Hill-Climbing Algorithm*, algoritma *Simulated Annealing* memiliki hasil akhir yang relatif konsisten. Hal ini dapat dilihat dari perbedaan nilai akhir yang kecil antara uji coba ke-1 sampai ke-3. Ini disebabkan oleh cara kerja *Simulated Annealing* yang dapat mengambil tindakan tertentu agar tidak terjebak di *local optimum*. Untuk genetic algorithm, hasil akhir dari algoritma sangat bergantung dari berbagai faktor, seperti *state* kubus inisial, probabilitas mutasi, pemilihan *parents*, dan populasinya sendiri sehingga nilai dari *final objective value* algoritma ini relatif tidak konsisten.

## KESIMPULAN & SARAN

### A. Kesimpulan

Berdasarkan hasil analisis yang telah kami lakukan, kami menemukan bahwa jenis algoritma *local search* yang paling sesuai untuk mencari solusi dari permasalahan *Diagonal Magic Cube* adalah algoritma ***Simulated Annealing***. Hal tersebut dikarenakan algoritma *Simulated Annealing* dapat menemukan hasil *objective value* yang paling mendekati *global optimum* secara konsisten dan cepat dibandingkan dengan algoritma lain. Sedangkan, algoritma yang paling tidak efisien dan optimal untuk mencari solusi permasalahan *Diagonal Magic Cube* adalah algoritma ***Stochastic Hill-Climbing***. Hal ini disebabkan karena dari hasil uji coba yang telah dilakukan dan hasil analisis sebelumnya, rata-rata *final objective value* yang dihasilkan oleh algoritma *Stochastic Hill-Climbing* mencapai lebih dari 2.000 yang jauh lebih besar dibandingkan dengan hasil algoritma lain meskipun durasi berjalannya algoritma ini tidak jauh berbeda dari algoritma *Hill-Climbing* lainnya.

### B. Saran

- Untuk percobaan selanjutnya agar mendapatkan solusi *Diagonal Magic Cube*, algoritma harus dijalankan berkali-kali
- Untuk percobaan selanjutnya *Objective Function* yang digunakan bisa dimodifikasi atau berbeda.
- Untuk percobaan selanjutnya *Stochastic Hill Climb*, *Simulated Annealing*, dan *Genetic Algorithm* dapat diperbanyak iterasi, temperatur, dan/atau populasinya.
- Untuk percobaan selanjutnya bisa gunakan satu komputer untuk menghasilkan hasil waktu yang konsisten (karena hasil dapat bergantung pada spesifikasi komputer)

## PEMBAGIAN TUGAS KELOMPOK

Nama	NIM	Pembagian Kerja
Eldwin Pradipta	18222042	<ul style="list-style-type: none"> <li>● Membuat Objective Function</li> <li>● Membuat Simulated Annealing</li> <li>● Menyatukan dan mengkonversi algoritma dari python ke golang</li> <li>● Membuat Tipe data penyimpanan</li> <li>● Membuat konversi data ke json</li> <li>● Membuat main</li> <li>● Membuat Tampilan</li> <li>● Debugging</li> <li>● Menuliskan Saran</li> </ul>
Mattheuw Suciadi Wijaya	18222048	<ul style="list-style-type: none"> <li>● Membuat implementasi secara tahap kasar pada Algoritma <i>Random Restart Hill-Climbing</i> (Python)</li> <li>● Membuat implementasi secara tahap kasar pada Algoritma <i>Stochastic Hill-Climbing</i> (Python)</li> <li>● Membuat Deskripsi Persoalan</li> <li>● Membuat dan merevisi bagian Pemilihan</li> </ul>

		<p><i>Objective Function</i></p> <ul style="list-style-type: none"> <li>• Membuat Hasil Perbandingan Terhadap Masing-Masing Algoritma Local Search</li> <li>• Membuat Bagian Hasil Eksperimen Terhadap Setiap algoritma <i>Hill-Climbing</i></li> <li>• Membuat Bagian Hasil Analisis Hill-Climbing</li> <li>• Membuat &amp; merapikan format laporan</li> <li>• Membantu membuat bagian kesimpulan</li> <li>• Membuat referensi</li> </ul>
Theo Livius Nathanel	18222076	<ul style="list-style-type: none"> <li>• Membuat laporan bagian hasil eksperimen uji coba simulated annealing</li> <li>• Membuat analisis simulated annealing, analisis konsistensi hasil algoritma secara keseluruhan</li> <li>• Membuat kesimpulan</li> <li>• Membantu membuat algoritma steepest ascent dan sideway moves di python</li> </ul>
Muhammad Ridho Rabbani	18222098	<ul style="list-style-type: none"> <li>• Membantu laporan bagian implementasi <i>Genetic Algorithm</i>, hasil</li> </ul>

		<p>percobaan <i>Genetic Algorithm</i> dan analisis <i>Genetic Algorithm</i></p> <ul style="list-style-type: none"><li>• Membantu membuat algoritma kasar untuk <i>Genetic Algorithm</i></li></ul>
--	--	---

## REFERENSI

- [1] “The Successful Search for The Smallest Perfect Magic Cube” – <https://www.trump.de/magic-squares/magic-cubes/cubes-1.html>
- [2] “Features of The Magic Cube” – <https://www.magischvierkant.com/three-dimensional-eng/magic-features/>
- [3] “Magic Cube” – [https://en.wikipedia.org/wiki/Magic\\_cube](https://en.wikipedia.org/wiki/Magic_cube)
- [4] S. Russel and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Accessed: Oct. 01, 2024. [Online]. Available: <https://aima.cs.berkeley.edu/>
- [5] “Mathworld News: Perfect Magic Cube of Order 5 Discovered” – <https://mathworld.wolfram.com/news/2003-11-18/magiccube/>