

IF3170 INTELIGENSI ARTIFISIAL
Pencarian Solusi Diagonal Magic Cube dengan Local Search

Tugas Besar 1

Disusun untuk memenuhi tugas mata kuliah Inteligensi Artifisial
pada Semester 1 (satu) Tahun Akademik 2024/2025
Tugas Besar IF3170 Inteligensi Artifisial



Oleh

Shabrina Maharani	13522134
Muhammad Fauzan Azhim	13522153
Muhammad Davis Adhipramana	13522157
Valentino Chrylie Triadi	13522164

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2024

DAFTAR ISI

DAFTAR ISI	2
BAB 1 DESKRIPSI PERSOALAN	3
BAB 2 PEMBAHASAN	4
2.1 Pemetaan Permasalahan	4
2.2 Pemilihan Objective Function	5
2.2.1 Deskripsi Kelas/Fungsi Objective Function	7
2.3 Implementasi Algoritma Local Search	11
2.3.1 Steepest Ascent Hill-climbing	12
2.3.1.1 Deskripsi Kelas/Fungsi Steepest Ascent Hill-climbing	12
2.3.1.2 Source Code Steepest Ascent Hill-climbing	13
2.3.2 Hill-climbing with Sideways Move	15
2.3.2.1 Deskripsi Kelas/Fungsi Hill-climbing with Sideways Move	15
2.3.2.2 Source Code Hill-climbing with Sideways Move	16
2.3.3 Random Restart Hill-climbing	18
2.3.3.1 Deskripsi Kelas/Fungsi Random Restart Hill-climbing	18
2.3.3.2 Source Code Random Restart Hill-climbing	19
2.3.4 Stochastic Hill-Cimbing	21
2.3.4.1 Deskripsi Kelas/Fungsi Stochastic Hill-Cimbing	21
2.3.4.2 Source Code Stochastic Hill-Cimbing	22
2.3.5 Simulated Annealing	23
2.3.5.1 Deskripsi Kelas/Fungsi Simulated Annealing	24
2.3.5.2 Source Code Simulated Annealing	25
2.3.6 Genetic Algorithm	27
2.3.6.1 Deskripsi Kelas/Fungsi Genetic Algorithm	28
2.3.6.2 Source Code Genetic Algorithm	30
2.3.7 Kelas/Fungsi Bantuan	32
2.3.7.1 Deskripsi Kelas/Fungsi Menemukan Highest Neighbor	32
2.3.7.2 Deskripsi Kelas/Fungsi Membangkitkan State secara Random	33
2.3.7.3 Deskripsi Kelas/Fungsi Mengecek Keberadaan dalam State	34
2.3.7.4 Deskripsi Kelas/Fungsi Membangkitkan Populasi	35
2.4 Hasil Eksperimen dan Analisis	36
2.4.1 Steepest Ascent Hill-Climbing	36
2.4.2 Hill-Climbing with Sideways Move	48
2.4.3 Random Restart Hill-Climbing	60
2.4.4 Stochastic Hill-Climbing	72
2.4.5 Simulated Annealing	83
2.4.6 Genetic Algorithm	93
2.4.7 Plot Objective Function	104
2.4.8 Analisis	107
BAB 3 KESIMPULAN DAN SARAN	113
3.1 Kesimpulan	113
3.1 Saran	113
Pembagian Tugas	115
Referensi	116

BAB 1 DESKRIPSI PERSOALAN

Diagonal Magic Cube merupakan sebuah kubus yang tersusun dari angka 1 hingga n^3 tanpa pengulangan, dengan n merupakan panjang sisi pada kubus tersebut. Persoalan ini diminta untuk menganalisis dan mengimplementasikan algoritma *local search* untuk menyelesaikan permasalahan *Diagonal Magic Cube* berukuran $5 \times 5 \times 5$. Permasalahan ini memiliki sifat-sifat khusus, yaitu bahwa jumlah angka dalam setiap baris, kolom, tiang, serta diagonal baik dalam potongan bidang maupun ruang, harus sama dengan sebuah angka yang disebut magic number.

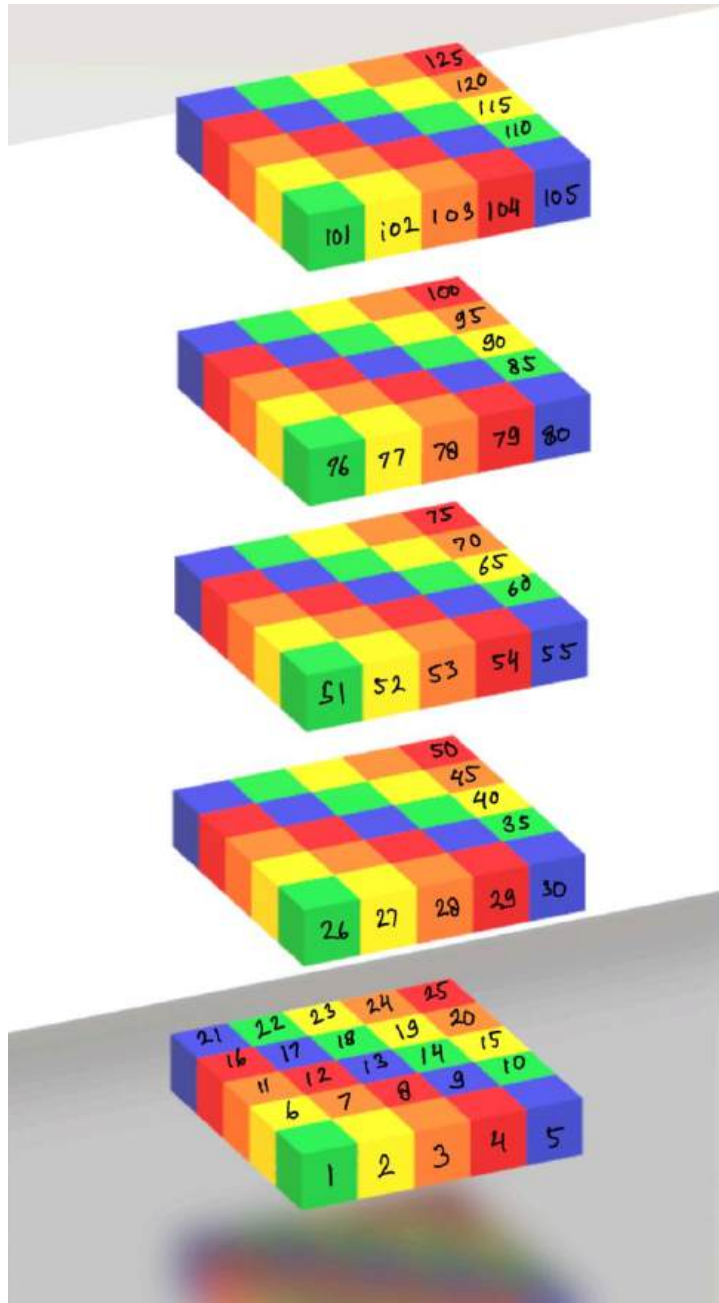
Magic number ini tidak harus berupa angka yang berada dalam rentang angka 1 hingga n^3 dan tidak harus dimasukkan ke dalam kubus. *Initial state* dari permasalahan ini adalah susunan angka acak dari 1 hingga 125 (5^3). Dalam proses pencarian solusi, setiap iterasi dari algoritma *local search* mengharuskan peserta untuk menukar posisi dua angka di dalam kubus tersebut. Penukaran angka akan menjadi mekanisme pembuatan solusi sampai kondisi yang diinginkan terpenuhi.

Beberapa algoritma *local search* seperti *Steepest Ascent Hill-climbing*, *Hill-climbing with Sideways Move*, *Random Restart Hill-climbing*, *Stochastic Hill-climbing*, *Simulated Annealing*, dan *Genetic Algorithm* akan dianalisis langkahnya dan dievaluasi untuk mencari solusi optimal. Tentunya, setiap algoritma memiliki pendekatan berbeda dalam melakukan proses menemukan *final state*. Dalam tugas besar ini, penulis diharapkan mampu mengimplementasikan tiap algoritma tersebut untuk menyelesaikan permasalahan. Penulis juga diharapkan untuk dapat menganalisis setiap algoritma dalam menyelesaikan persoalan ini melalui pertimbangan eksekusi waktu, jumlah iterasi, dan nilai objective function dari setiap algoritma.

BAB 2 PEMBAHASAN

2.1 Pemetaan Permasalahan

Dalam permasalahan *Magic Cube* 5x5x5, kami memetakan setiap kubus kecil didalamnya dengan angka 1 hingga 125 (5^3) yang melambangkan posisi kubus. Penomoran angka tersebut penulis gambarkan dengan ilustrasi berikut:



Gambar 1. Ilustrasi pemetaan sel kubus

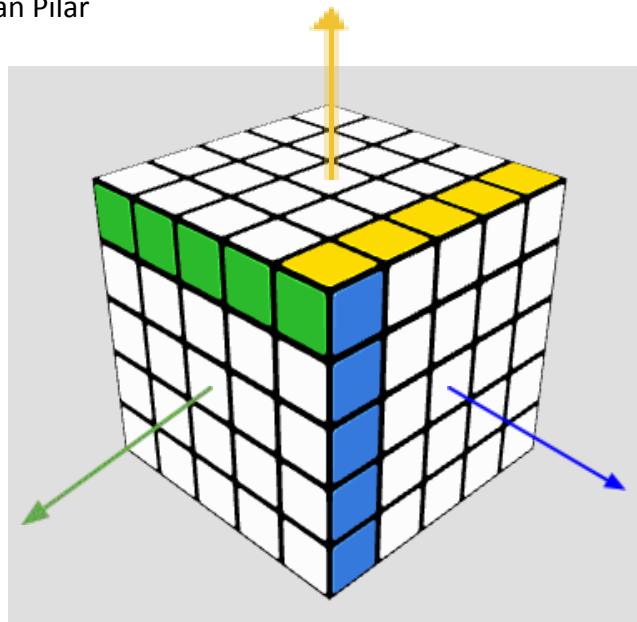
2.2 Pemilihan *Objective Function*

Objective function adalah sebuah fungsi yang merepresentasikan nilai seberapa jauh suatu state saat ini dari state tujuan yang diinginkan. Semakin tinggi nilainya, semakin mendekati state tersebut pada tujuan, yang berarti fungsi ini digunakan untuk menilai tingkat keberhasilan dalam mencapai target.

Dalam permasalahan ini, penulis memilih *objective function* yang mengukur seberapa banyak angka yang jika dipasangkan dengan 4 angka lainnya dalam satu pilar, kolom, baris, diagonal bidang, atau diagonal ruang, jumlah dari kelima angka tersebut merupakan *magic number*. Alasan pemilihan *objective function* dengan menjumlahkan banyak sel yang memenuhi pernyataan tersebut karena nilai *objective function* yang kami inginkan memiliki nilai yang semakin besar jika semakin mendekati tujuan. Dibandingkan dengan menggunakan frekuensi dari nilai-nilai yang membentuk *magic number*, penjumlahan ini membuat pemahaman dan analisis masing-masing algoritma lebih mudah dilakukan.

Pada permasalahan ini, penulis memandang kubus dengan representasi bidang X sebagai baris (panah berwarna hijau pada gambar pertama di rincian), bidang Y sebagai kolom (panah berwarna kuning), dan Z sebagai pilar (panah berwarna biru). Berdasarkan perhitungan matematika yang telah dilakukan, didapatkan nilai maksimum dari *objective function* pada kubus 5x5x5 adalah 109 dengan rincian sebagai berikut :

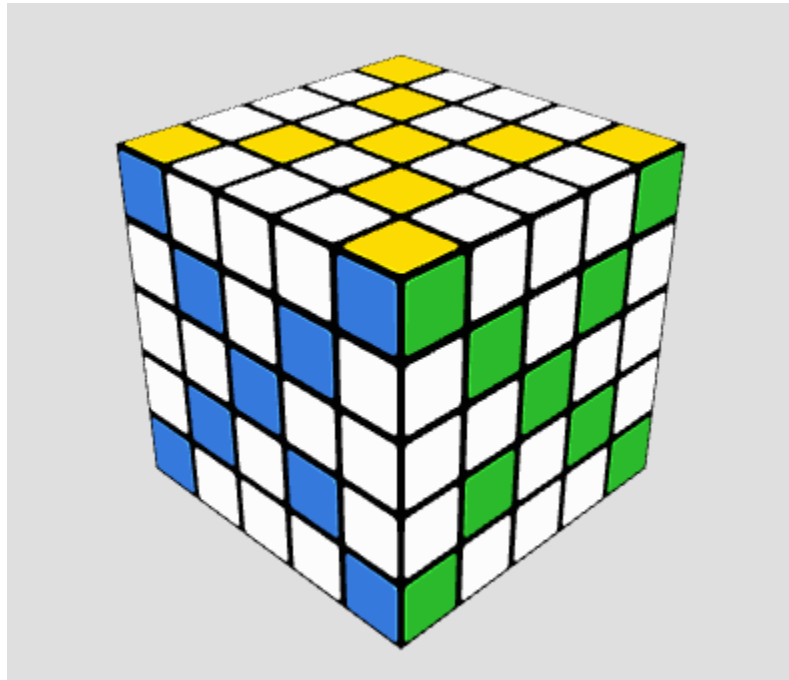
- Baris, Kolom, dan Pilar



Gambar 2. Ilustrasi baris, kolom, dan pilar

Jumlah dari masing-masing baris (5 sel berwarna hijau = 1 baris), kolom (5 sel berwarna kuning = 1 kolom), dan pilar (5 sel berwarna biru = 1 pilar) adalah sebanyak 25 atau n^2 . Sehingga total dari ketiganya adalah 75 atau $3 \times n^2$.

- Diagonal Bidang



Gambar 3. Ilustrasi diagonal di setiap bidang

Jumlah diagonal pada setiap bidang terdiri dari dua diagonal sehingga total dari diagonal bidang pada kubik tersebut adalah

$$\text{Jumlah diagonal bidang } X + \text{jumlah diagonal bidang } Y + \text{jumlah diagonal bidang } Z$$

$$= 5(2) + 5(2) + 5(2)$$

$$= 30$$

- Diagonal Ruang

Untuk seluruh kubus dengan ukuran berapapun, diagonal ruangnya akan selalu berjumlah 4.

Dengan demikian, bisa dilakukan nilai paling maksimal dari *objective function* yang berupa penjumlahan antara baris, kolom, pilar, diagonal bidang, dan diagonal ruang yang membentuk *magic number* sebagai berikut :

Jumlah baris, kolom, pilar + jumlah diagonal bidang + jumlah diagonal ruang

= 75 + 30 + 4

= 109

2.2.1 Deskripsi Kelas/Fungsi Objective Function

Fungsi ObjectiveFunction ini bertujuan untuk menghitung nilai objective function dari suatu state, di mana state tersebut direpresentasikan sebagai array dengan panjang 125 integer yang menggambarkan kubus berukuran 5x5x5. Fungsi ini menerima input berupa state tersebut, dan menghasilkan nilai cost yang menunjukkan berapa banyak baris, kolom, pilar, atau diagonal di dalam kubus yang memiliki jumlah sama dengan magic number yang direpresentasikan oleh magicNb, yang sudah didefinisikan sebagai 315. Semakin tinggi nilai *cost* menunjukkan bahwa semakin banyak bagian dari kubus yang sesuai dengan kondisi magicNb, yang berarti state tersebut lebih mendekati global maksimum.

Pada awal fungsi, variabel cost diinisialisasi dengan nilai 0 untuk menampung jumlah bagian yang sesuai dengan magicNb. Konstanta n diset menjadi 5 untuk mewakili ukuran sisi kubus, dan nn yang merupakan $n * n$, mewakili total elemen pada satu bidang kubus.

Fungsi akan menghitung jumlah setiap 5 elemen berturut-turut dalam setiap baris di satu bidang (XY) dalam kubus. Loop pertama dengan indeks i berjalan dari 0 hingga nn (25 kali) untuk menjangkau semua baris pada setiap bidang. Pada loop dalam, indeks j berjalan dari 0 hingga 4 (5 kali) untuk menambahkan elemen-elemen pada baris ke-i dalam bidang tersebut. Jika hasil penjumlahan baris tersebut sama dengan magicNb, maka cost akan ditambahkan 1.

Setelah penghitungan baris, fungsi akan menghitung setiap kolom. Loop dengan indeks i berjalan dari 0 hingga 4 untuk menjangkau setiap kolom pada bidang tersebut. Setiap kali i bertambah, sebuah variabel base ditetapkan sebagai $i * nn$, yang menunjukkan elemen awal dari kolom tersebut pada bidang kubus. Di dalam loop kedua dengan indeks j dari 0 hingga 4, fungsi akan

menambahkan elemen-elemen pada kolom tersebut. Jika jumlah elemen kolom sama dengan magicNb, cost bertambah 1.

Selanjutnya, pilar dihitung di seluruh kubus, yang berarti menghitung penjumlahan elemen-elemen yang vertikal di setiap titik bidang, atau bidang dengan panah berwarna biru yang sudah dijelaskan pada bagian sebelumnya. Dalam loop dengan indeks i dari 0 hingga nn (25 kali), loop dalam dengan j dari 0 hingga 4 menambahkan elemen-elemen pilar dari setiap titik pada bidang tersebut. Jika penjumlahan pilar tersebut mencapai magicNb, cost bertambah 1.

Kemudian, fungsi akan menghitung dua diagonal pada setiap bidang XY di sepanjang sumbu Z. Untuk setiap bidang (dengan indeks i dari 0 hingga 4), dua variabel sumAD1 dan sumAD2 diinisialisasi untuk dua diagonal. baseAD1 dan baseAD2 ditentukan untuk memulai perhitungan diagonal dari dua ujung bidang tersebut. Loop dalam j dari 0 hingga 4 menghitung jumlah dari kedua diagonal. Jika salah satu diagonal memiliki jumlah sama dengan magicNb, cost bertambah 1. Dua diagonal pada bidang XZ dihitung dalam cara yang sama. Dengan indeks i dari 0 hingga 4, setiap i menunjukkan bidang XZ yang berbeda. sumAD1 dan sumAD2 menghitung diagonal bidang, dengan baseAD1 dan baseAD2 menunjukkan elemen awal pada setiap bidang. Jika jumlah pada salah satu diagonal sama dengan magicNb, maka cost akan bertambah 1. Selanjutnya, dua diagonal pada bidang YZ dihitung dengan pola yang mirip. Untuk setiap indeks i (0 hingga 4) yang menunjukkan bidang YZ tertentu, elemen pada dua diagonal dihitung dan ditambahkan dalam sumAD1 dan sumAD2. Jika salah satu diagonal tersebut memenuhi magicNb, maka cost ditambah 1.

Terakhir, empat diagonal utama yang menghubungkan sudut-sudut kubus dihitung. Loop dengan indeks i berjalan dari 0 hingga 4, menambahkan elemen-elemen diagonal yang bersesuaian untuk setiap i . Keempat sum (sumSD1, sumSD2, sumSD3, sumSD4) masing-masing menghitung satu diagonal dari kubus. Jika salah satu jumlah diagonal ruang sama dengan magicNb, cost ditambahkan 1.

Setelah semua perhitungan selesai, fungsi ObjectiveFunction mengembalikan nilai cost, yang merupakan jumlah total baris, kolom, pilar, atau diagonal yang memiliki jumlah sama dengan magicNb. Nilai ini merepresentasikan seberapa dekat state saat ini dengan global maksimum sesuai kriteria yang ditentukan. Berikut adalah source code dari fungsi penghitungan nilai *objective function*.


```
1 package lib
2
3 const magicNb = 315
4
5 func ObjectiveFunction(state [125]int) int {
6     // Calculate the cost of the state
7     cost := 0
8
9     n := 5
10    nn := n * n
11
12    // Count Row
13    for i := 0; i < nn; i++ {
14        sum := 0
15        for j := 0; j < n; j++ {
16            sum += state[i*n+j]
17        }
18        if sum == magicNb {
19            cost++
20        }
21    }
22
23    // Count Column
24    for i := 0; i < n; i++ {
25        sum := 0
26        base := i * nn
27        for j := 0; j < n; j++ {
28            sum += state[base+(j*n)]
29        }
30        if sum == magicNb {
31            cost++
32        }
33    }
```

```

1
2 // Count Pillar
3 for i := 0; i < nn; i++ {
4     sum := 0
5     for j := 0; j < n; j++ {
6         sum += state[i+(j*nn)]
7     }
8     if sum == magicNb {
9         cost++
10    }
11 }
12
13 // Count Area Z Diagonal
14 for i := 0; i < n; i++ {
15     sumAD1, sumAD2 := 0, 0
16     baseAD1 := i * n
17     baseAD2 := i*n + n - 1
18     for j := 0; j < n; j++ {
19         sumAD1 += state[baseAD1+j*(nn+1)]
20         sumAD2 += state[baseAD2+j*(nn-1)]
21     }
22     if sumAD1 == magicNb {
23         cost++
24     }
25     if sumAD2 == magicNb {
26         cost++
27     }
28 }
29
30 // Count Area Y Diagonal
31 for i := 0; i < n; i++ {
32     sumAD1, sumAD2 := 0, 0
33     baseAD1 := i * nn
34     baseAD2 := i*nn + nn - n
35     for j := 0; j < n; j++ {
36         sumAD1 += state[baseAD1+j*(n+1)]
37         sumAD2 += state[baseAD2-j*(n-1)]
38     }
39     if sumAD1 == magicNb {
40         cost++
41     }
42     if sumAD2 == magicNb {
43         cost++
44     }
45 }
46
47 // Count Area X Diagonal
48 for i := 0; i < n; i++ {
49     sumAD1, sumAD2 := 0, 0
50     baseAD1 := i
51     baseAD2 := nn - n + i
52     for j := 0; j < n; j++ {
53         sumAD1 += state[baseAD1+j*(nn+n)]
54         sumAD2 += state[baseAD2+j*(nn-n)]
55     }
56     if sumAD1 == magicNb {
57         cost++
58     }
59     if sumAD2 == magicNb {
60         cost++
61     }
62 }

```

```

1      // Count Space Diagonal
2      sumSD1, sumSD2, sumSD3, sumSD4 := 0, 0, 0, 0
3      for i := 0; i < n; i++ {
4          sumSD1 += state[i*(nn+n+1)]
5          sumSD2 += state[nn-1+i*(nn-n-1)]
6          sumSD3 += state[n-1+i*(nn+n-1)]
7          sumSD4 += state[nn-n+i*(nn-n+1)]
8      }
9      if sumSD1 == magicNb {
10         cost++
11     }
12     if sumSD2 == magicNb {
13         cost++
14     }
15     if sumSD3 == magicNb {
16         cost++
17     }
18     if sumSD4 == magicNb {
19         cost++
20     }
21
22     return cost
23 }

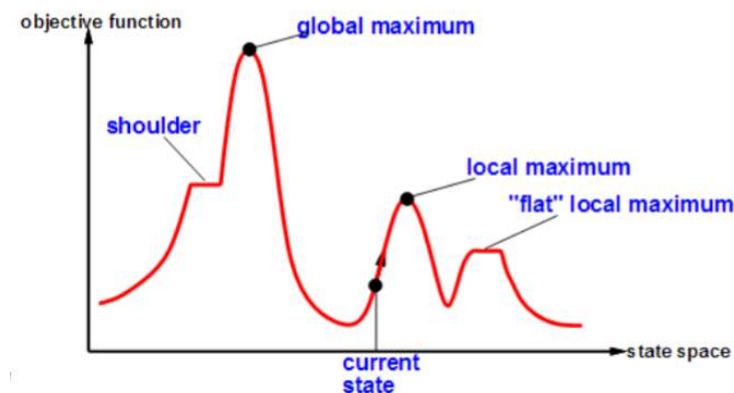
```

2.3 Implementasi Algoritma Local Search

Local search adalah algoritma pencarian yang beroperasi dengan mengeksplorasi dari satu state ke state lain yang berdekatan (*neighbor state*) tanpa perlu melacak jalur yang ditempuh sebelumnya. Metode ini lebih mengutamakan penilaian dari state saat ini berdasarkan *objective function*. Keunggulan utama *local search* adalah penggunaan memori yang minimal dan kemampuannya untuk menangani masalah optimisasi di mana yang paling penting adalah menemukan state terbaik, bukan bagaimana caranya mencapai state tersebut. Algoritma ini biasanya digunakan dalam permasalahan yang sangat besar atau tak terbatas. Tujuan akhirnya adalah menemukan state yang optimal dengan nilai maksimum dari *objective function* yang digunakan.

2.3.1 Steepest Ascent Hill-climbing

Algoritma Steepest Ascent Hill Climbing adalah varian dari algoritma pencarian Hill-climbing yang setiap iterasinya bergerak menuju *neighbor state* dengan nilai objective function tertinggi, atau berdasarkan gambar *landscape* Hill-climbing, iterasinya akan bergerak ke arah dengan kenaikan paling curam. Proses ini akan berhenti (*terminate*) ketika mencapai *local optimum*, yaitu suatu state di mana tidak ada neighbor yang memiliki nilai lebih tinggi dari state saat ini.



Gambar 4. Landscape Hill-climbing

Berikut adalah implementasi proses pencarian solusi *Diagonal Magic Cube* dengan menggunakan Steepest ascent hill climbing.

2.3.1.1 Deskripsi Kelas/Fungsi Steepest Ascent Hill-climbing

Fungsi `SteepestAscent` disimpan dalam sebuah file `SteepestAscent.go`. Fungsi ini menerima input berupa `initialState`, yang merupakan state awal dari kubus berukuran 5x5x5 yang akan dioptimalkan. Pertama, fungsi menginisialisasi beberapa variabel seperti variabel `iteration` untuk menghitung jumlah iterasi yang dilakukan, dan `currentTime` untuk mencatat waktu mulai eksekusi. Variabel `bestState` diinisialisasi dengan nilai `initialState`, dan `bestCost` diatur dengan nilai hasil dari fungsi `ObjectiveFunction` dari `bestState`, yang berarti menghitung nilai objective dari initial state itu sendiri. Fungsi ini kemudian memasuki loop tanpa batas, di mana pada setiap iterasi, terdapat fungsi `FindHighestNeighbor` dari package `lib` yang dipanggil untuk mencari neighbor dengan nilai objective tertinggi dari seluruh neighbor yang didapatkan oleh `bestState`. Neighbor dengan nilai objective

tertinggi tersebut nantinya akan dimasukkan ke dalam variabel `neighbor` dan nilai `objective function` dari `neighbor` tersebut akan dimasukkan ke dalam variabel `neighborCost`. Jika nilai `neighborCost` tidak lebih besar dari `bestCost`, berarti tidak ada perubahan yang terjadi dan loop akan berakhir. Jika `neighborCost` lebih besar dari `bestCost`, `bestState` dan `bestCost` diperbarui dengan `neighbor` dan `neighborCost`, dan `iteration` bertambah satu.

Setelah keluar dari loop, durasi eksekusi dihitung dengan menghitung selisih waktu dari `currentTime` hingga saat ini. Fungsi kemudian mengonversi `initialState` dan `bestState` menjadi bentuk hasil menggunakan `lib.ConvertToResult`, menghasilkan `firstState` dan `lastState`. Selanjutnya, fungsi akan membuat sebuah objek `res` berupa map yang menyimpan informasi hasil akhir algoritma, termasuk nama algoritma, deskripsi dengan nilai `Objective Function` dari `bestCost`, durasi eksekusi, dan jumlah iterasi. Data state awal dan state akhir disimpan sebagai `firstState` dan `lastState` dalam `res`. Data hasil ini kemudian disimpan dalam format JSON menggunakan `lib.SaveToJson`. Terakhir, fungsi akan mengembalikan `true` sebagai tanda bahwa proses telah selesai.

2.3.1.2 Source Code Steepest Ascent Hill-climbing

Berikut adalah source code dari fungsi Steepest Ascent.

```

1  package Algorithm
2
3  import (
4      "magic-cube/lib"
5      "time"
6      "fmt"
7  )
8
9
10 func SteepestAscent(initialState [125]int) bool{
11
12     lib.PrintState(initialState)
13     iteration := 0
14     currentTime := time.Now()
15
16     bestState := initialState
17     bestCost := lib.ObjectiveFunction(bestState)
18
19     for {
20
21         neighbor, neighborCost, _ := lib.FindHighestNeighbor(bestState)
22
23         if neighborCost <= bestCost {
24             break
25         }
26
27         bestState = neighbor
28         bestCost = neighborCost
29         iteration++
30     }
31     executeTime := time.Since(currentTime)
32
33     firstState := lib.ConvertToResult(initialState)
34     lastState := lib.ConvertToResult(bestState)
35     fmt.Println("Steepest Ascent complete")
36     lib.PrintState(bestState)
37
38     res := map[string] interface{}{
39         "algorithm" : "Steepest Ascent Algorithm",
40         "description" : map[string] interface{}{
41             "Objective Function" : bestCost,
42             "Duration" : executeTime.String(),
43             "Jumlah Iterasi" : iteration,
44         },
45         "firstState" : firstState,
46         "lastState" : lastState,
47     }
48
49     lib.SaveToJson(res)
50
51     return true;
52
53 }

```

2.3.2 Hill-climbing with Sideways Move

Sedikit berbeda dengan Steepest Ascent Hill Climbing, algoritma Hill-climbing with Sideways Move juga merupakan varian dari algoritma Hill Climbing yang setiap iterasinya bergerak menuju *neighbor state* dengan nilai *objective function* tertinggi. Namun, dalam algoritma ini, proses tidak akan berhenti jika *neighbor state* memiliki nilai *objective function* yang sama atau lebih tinggi dari current state. Hal ini memungkinkan terjadinya kondisi datar (*flat*) pada *landscape* Hill Climbing, di mana algoritma dapat terus bergerak meskipun tidak ada peningkatan nilai *objective function*.

Berikut adalah implementasi dari proses pencarian solusi *Diagonal Magic Cube* dengan menggunakan Hill-climbing with Sideways Move.

2.3.2.1 Deskripsi Kelas/Fungsi Hill-climbing with Sideways Move

Fungsi Sideways disimpan dalam file Sideways.go. Fungsi ini menerima parameter `max_iteration`, yang berfungsi sebagai batas jumlah iterasi maksimum untuk langkah Hill-climbing with Sideways Move ketika bertemu dengan nilai yang sama. Fungsi ini dimulai dengan membangkitkan `initialState` dari kubus berukuran 5x5x5. Beberapa variabel diinisialisasi, diantaranya variabel `iteration` untuk menghitung jumlah iterasi yang dilakukan dan `currentTime` untuk mencatat waktu mulai eksekusi. Variabel `bestState` diinisialisasi dengan `initialState`, dan `bestCost` diset dengan hasil dari fungsi `ObjectiveFunction` dari `initialState` tersebut.

Fungsi kemudian memasuki loop, di mana pada setiap iterasi, fungsi `FindHighestNeighbor` dari package `lib` dipanggil untuk menemukan *neighbor* dengan nilai objektif tertinggi dari semua *neighbor* yang dapat dihasilkan dari `bestState`. Neighbor ini beserta nilai objektifnya disimpan dalam variabel `neighbor` dan `neighborCost`. Jika `neighborCost` lebih rendah dari `bestCost`, loop berhenti yang berarti state terjebak di local maxima. Namun, jika nilai `neighborCost` sama dengan `bestCost`, maka algoritma melakukan langkah *sideways*, di mana nilai `max_iteration` berkurang satu untuk mencatat berapa kali langkah *sideways* telah dilakukan. Jika `max_iteration` mencapai nol, berarti batas langkah *sideways* sudah habis, dan loop berakhir. Jika `neighborCost` lebih besar dari `bestCost`, `bestState` dan `bestCost` diperbarui dengan nilai `neighbor` dan `neighborCost`, dan variabel `iteration` bertambah.

Setelah keluar dari loop, durasi eksekusi dihitung dengan membandingkan waktu saat ini dengan `currentTime`. `initialState` dan `bestState` diubah menjadi bentuk yang dapat dibaca hasilnya menggunakan `lib.ConvertToResult`, menghasilkan `firstState` dan `lastState`. Kemudian, sebuah objek `res` berupa map dibuat untuk menyimpan informasi hasil akhir algoritma, termasuk nama algoritma, deskripsi dengan nilai fungsi objektif (Objective Function) dari `bestCost`, durasi eksekusi, dan jumlah iterasi. Data keadaan awal (`firstState`) dan keadaan akhir (`lastState`) juga disimpan dalam `res`. Akhirnya, hasil ini disimpan dalam format JSON menggunakan `lib.SaveToJson`, dan fungsi mengembalikan `true` sebagai tanda bahwa proses telah selesai.

2.3.2.2 Source Code Hill-climbing with Sideways Move

Berikut adalah source code dari fungsi Hill-climbing with Sideways Move.


```

1 package Algorithm
2
3 import (
4     "magic-cube/lib"
5     "time"
6 )
7
8 func Sideways(max_iteration int) bool{
9     // Generate Initial Variable
10    initialState := lib.GenerateSuccessor()
11    iteration := 0
12    currentTime := time.Now()
13
14    bestState := initialState
15    bestCost := lib.ObjectiveFunction(bestState)
16
17    for {
18
19        neighbor, neighborCost, _ := lib.FindHighestNeighbor(bestState)
20
21        if neighborCost < bestCost{
22            break
23        }else if(neighborCost == bestCost){
24            // Asumsi: Ga Reset ya bang klo misal dia berhasil naik
25            if(max_iteration == 0){
26                break
27            }
28            max_iteration--;
29        }
30
31        bestState = neighbor
32        bestCost = neighborCost
33        iteration++
34    }
35    executeTime := time.Since(currentTime)
36
37    firstState := lib.ConvertToResult(initialState)
38    lastState := lib.ConvertToResult(bestState)
39
40    res := map[string] interface{}{
41        "algorithm" : "Sideways Algorithm",
42        "description" : map[string] interface{}{
43            "Objective Function" : bestCost,
44            "Duration" : executeTime.String(),
45            "Jumlah Iterasi" : iteration,
46        },
47        "firstState" : firstState,
48        "lastState" : lastState,
49    }
50
51    lib.SaveToJson(res)
52
53    return true;
54 }
55
56 }

```

2.3.3 Random Restart Hill-climbing

Random Restart Hill-climbing adalah varian dari algoritma Hill-climbing yang tetap mencari *neighbor state* dengan nilai *objective function* tertinggi di setiap iterasi. Namun, jika *neighbor state* tersebut memiliki nilai *objective function* yang lebih rendah daripada current state, algoritma akan melakukan restart secara acak. Ini berarti algoritma akan membangkitkan *initial state* baru dan memulai kembali proses pencarian dari awal.

Berikut adalah proses pencarian solusi *Diagonal Magic Cube* dengan menggunakan Random Restart Hill-climbing :

2.3.3.1 Deskripsi Kelas/Fungsi Random Restart Hill-climbing

Fungsi RandomRestart adalah implementasi dari algoritma Random Restart Hill-climbing yang menerima parameter max_restart, yaitu jumlah maksimum restart yang diperbolehkan. Pertama, fungsi melakukan inisialisasi variabel restart dan iteration untuk menghitung jumlah restart dan iterasi, sementara currentTime menyimpan waktu mulai eksekusi. State awal dibangkitkan menggunakan fungsi GenerateSuccessor dari package lib dan disimpan dalam initialState, yang sekaligus menjadi inisialisasi dari globalBestState. Nilai objective function dari globalBestState dihitung menggunakan fungsi ObjectiveFunction dari package lib dan disimpan dalam globalBestCost sebagai nilai inisialisasi.

Kemudian, proses algoritma akan memasuki loop utama yang berjalan hingga jumlah restart mencapai max_restart. Dalam loop ini, terdapat inner loop yang melakukan pencarian neighbor state dengan nilai objective function tertinggi menggunakan fungsi FindHighestNeighbor dari package lib. Jika nilai objective function dari neighbor lebih rendah atau sama dengan nilai current state, inner loop akan berhenti dan akan melanjutkan ke proses pengecekan apakah currentCost lebih besar dari globalBestCost. Jika nilai objective function lebih tinggi dari currentCost, currentState dan currentCost diperbarui menjadi neighbor dan neighborCost, kemudian nilai iteration akan bertambah satu dan proses akan berlanjut dengan mengulangi pencarian highest neighbor. Setelah keluar dari inner loop, fungsi akan memeriksa apakah currentCost lebih besar dari globalBestCost. Jika benar, globalBestState dan globalBestCost diperbarui.

Sebelum melakukan restart, fungsi akan mengecek apakah restart kurang dari `max_restart - 1`. Jika demikian, fungsi akan membangkitkan state acak baru sebagai `currentState` menggunakan fungsi `GenerateSuccessor` dari package `lib` dan menghitung nilai `objective function` dari state baru ini sebagai `currentCost`. Jumlah restart kemudian ditambah satu, dan loop utama berulang hingga `max_restart` tercapai. Setelah loop selesai, waktu eksekusi dihitung dari `currentTime`. `initialState` dan `globalBestState` diubah menjadi format hasil menggunakan `lib.ConvertToResult`, dan variabel `firstState` dan `lastState` menyimpan state awal dan state akhir terbaik Sebuah objek `res` kemudian dibuat sebagai map yang menyimpan informasi akhir algoritma, termasuk nama algoritma, nilai `objective function` dari state terbaik (`globalBestCost`), durasi eksekusi, jumlah iterasi, dan jumlah restart yang dilakukan. Data state awal dan state akhir disimpan sebagai `firstState` dan `lastState` dalam `res`, dan hasilnya disimpan dalam format JSON menggunakan fungsi `SaveToJson` dari package `lib`. Terakhir, fungsi mengembalikan `true` sebagai tanda bahwa proses selesai.

2.3.3.2 Source Code Random Restart Hill-climbing

Berikut adalah source code dari fungsi Random Restart Hill-climbing.

```

1 package Algorithm
2
3 import (
4     "magic-cube/lib"
5     "time"
6 )
7
8 func RandomRestart(max_restart int) bool {
9     restart := 0
10    iteration := 0
11    currentTime := time.Now()
12
13    initialState := lib.GenerateSuccessor()
14    globalBestState := initialState
15    globalBestCost := lib.ObjectiveFunction(globalBestState)
16    currentState := globalBestState
17    currentCost := globalBestCost
18
19
20    for restart < max_restart {
21
22        for {
23            neighbor, neighborCost, _ := lib.FindHighestNeighbor(currentState)
24
25            if neighborCost <= currentCost {
26                break
27            }
28
29            currentState = neighbor
30            currentCost = neighborCost
31            iteration++
32        }
33
34        if currentCost > globalBestCost {
35            globalBestState = currentState
36            globalBestCost = currentCost
37        }
38
39
40        if restart < max_restart-1 {
41            currentState = lib.GenerateSuccessor()
42            currentCost = lib.ObjectiveFunction(currentState)
43        }
44
45        restart++
46    }
47
48    executeTime := time.Since(currentTime)
49
50    firstState := lib.ConvertToResult(initialState)
51    lastState := lib.ConvertToResult(globalBestState)
52
53    res := map[string]interface{}{
54        "algorithm": "Random Restart Algorithm",
55        "description": map[string]interface{}{
56            "Objective Function": globalBestCost,
57            "Duration":           executeTime.String(),
58            "Jumlah Iterasi":      iteration,
59            "Jumlah Restart":     restart,
60        },
61        "firstState": firstState,
62        "lastState":  lastState,
63    }
64
65    lib.SaveToJson(res)
66
67    return true
68 }
69

```

2.3.4 Stochastic Hill-Cimbing

Stochastic Restart Hill-climbing adalah varian dari algoritma *Hill-climbing* yang mirip dengan *random restart Hill-climbing*, namun memiliki batasan pada jumlah *restart* yang dilakukan. Pada *random restart Hill-climbing*, algoritma akan terus mengulangi proses pencarian dan melakukan *restart* hingga menemukan solusi yang optimal. Di sisi lain, pada *stochastic restart Hill-climbing*, terdapat batas maksimum jumlah pengulangan.

Secara sederhana, *stochastic restart Hill-climbing* memberikan keseimbangan antara waktu pencarian dan keinginan untuk mencapai solusi optimal. Dengan adanya batasan jumlah pengulangan, algoritma dapat menghindari pencarian yang terlalu lama jika tidak ada solusi yang lebih baik ditemukan dalam batasan waktu yang wajar.

Berikut adalah proses pencarian solusi *Diagonal Magic Cube* dengan menggunakan Stochastic Hill-climbing

2.3.4.1 Deskripsi Kelas/Fungsi Stochastic Hill-Cimbing

Fungsi `StochasticHillClimbing` adalah implementasi algoritma *stochastic hill climbing* yang menerima parameter `MaxIteration` yang menjadi jumlah maksimum iterasi yang akan dijalankan. Pada tahap awal, fungsi akan membangkitkan keadaan awal secara acak menggunakan fungsi `GenerateSuccessor` dari package `lib` dan menginisialisasi variabel `bestState` dengan `initialState`. Nilai fungsi objektif dari `bestState` dihitung menggunakan fungsi `ObjectiveFunction` dan disimpan dalam `bestCost`. Kemudian, fungsi memasuki loop yang akan terus berjalan hingga mencapai `MaxIteration`. Di dalam loop, sebuah `neighbor` dari `bestState` akan dibangkitkan secara random dan dihasilkan menggunakan fungsi `RandomNeighbor` dari package `lib`, lalu nilai fungsi objektif dari `neighbor` tersebut dihitung. Jika nilai fungsi objektif dari `neighbor` tersebut lebih tinggi dari `bestCost`, maka `neighbor` tersebut akan dijadikan `bestState` baru, dan `bestCost` diperbarui. Jika nilai `objective function` dari `neighbor` tersebut lebih rendah atau sama dengan `bestCost`, proses akan melanjutkan perulangan dengan mencari state random kembali sampai maks iterasi tercapai. Setelah loop selesai, waktu eksekusi dalam dihitung dari waktu mulai (`currentTime`). Keadaan awal (`initialState`) dan keadaan akhir terbaik (`bestState`) dikonversi ke format hasil yang dapat dibaca

menggunakan lib.ConvertToResult. Semua data hasil, termasuk informasi algoritma, nilai fungsi objektif terbaik, durasi eksekusi, dan jumlah iterasi, disimpan dalam format JSON menggunakan lib.SaveToJson. Pada akhir fungsi, nilai true dikembalikan untuk menandakan bahwa eksekusi telah selesai.

2.3.4.2 Source Code Stochastic Hill-Cimbing

Berikut adalah source code dari fungsi Stochastic Hill-climbing.

```
1 package Algorithm
2
3 import (
4     "magic-cube/lib"
5     "strconv"
6     "time"
7 )
8
9 func StochasticHillClimbing(MaxIteration int) bool {
10     // Generate a random initial state
11     initialState := lib.GenerateSuccessor()
12
13     iteration := 0
14     currentTime := time.Now()
15
16     bestState := initialState
17     bestCost := lib.ObjectiveFunction(bestState)
18
19     for {
20         if iteration > MaxIteration {
21             break
22         }
23
24         neighbor, neighborCost := lib.RandomNeighbor(bestState)
25
26         if neighborCost < bestCost {
27             bestState = neighbor
28             bestCost = neighborCost
29         }
30
31         iteration++
32     }
33
34     executeTime := time.Since(currentTime).Milliseconds()
35     firstState := lib.ConvertToResult(initialState)
36     lastState := lib.ConvertToResult(bestState)
37
38     res := map[string]interface{}{
39         "algorithm": "Stochastic Hill Climbing",
40         "description": map[string]interface{}{
41             "Objective Function": bestCost,
42             "Duration":          strconv.FormatInt(executeTime, 10) + "ms",
43             "Jumlah Iterasi":     iteration,
44         },
45         "firstState": firstState,
46         "lastState":  lastState,
47     }
48
49     lib.SaveToJson(res)
50
51     return true
52 }
```

2.3.5 Simulated Annealing

Simulated Annealing adalah algoritma optimisasi yang menggabungkan pendekatan *Hill-climbing* dengan elemen *random walk* untuk mencapai keseimbangan antara efisiensi dan *completeness* dalam pencarian solusi optimal. Dalam konteks *Hill-climbing*, algoritma ini berusaha menghindari jebakan *local maxima* dimana pencarian terhenti pada local optimum, yang mungkin tidak optimal secara keseluruhan. Sebaliknya, pada *random walk*, pencarian dapat mencapai solusi global, tetapi seringkali tidak efisien karena langkah-langkahnya bisa terlalu banyak.

Simulated annealing terinspirasi oleh proses fisik *annealing* dalam metalurgi, di mana logam atau kaca dipanaskan hingga suhu tinggi, lalu secara perlahan didinginkan untuk mencapai struktur energi rendah yang stabil. Dalam algoritma ini, perumpamaan dari *annealing* diterapkan pada pencarian solusi optimal di mana kita mencoba menemukan solusi dengan nilai terendah (untuk masalah minimisasi biaya) atau nilai tertinggi (untuk masalah maksimisasi keuntungan).

Secara sederhana, simulated annealing ini beroperasi juga dengan mengganti current state dengan neighbor yang memiliki nilai *objective function* lebih dari nilai *objective function* dari current state. Namun, jika neighbor state memiliki nilai *objective function* yang lebih rendah dari yang dimiliki oleh current state masih ada kemungkinan *neighbor state* tersebut dipilih untuk menjadi current state tergantung dengan nilai probabilitasnya. Probabilitas tersebut dapat dihitung dengan menggunakan rumus berikut:

$$e^{\Delta E/T}$$

Gambar 6. Rumus probabilitas worse neighbor

Keterangan :

e = bilangan euler

ΔE = *neighbor state objective function value* - *current state objective function value*

T = Temperatur, dalam konteks penyelesaian permasalahan ini merupakan sebuah bilangan yang lama kelamaan nilainya akan menurun.

Berikut adalah proses pencarian solusi *Diagonal Magic Cube* dengan menggunakan Simulated Annealing.

2.3.5.1 Deskripsi Kelas/Fungsi Simulated Annealing

Fungsi SimulatedAnnealing adalah implementasi algoritma Simulated Annealing untuk menyelesaikan masalah Magic Cube. Algoritma ini diawali dengan mendefinisikan beberapa konstanta yang berperan penting dalam prosesnya. `startingTemperature` adalah suhu awal algoritma yang diset pada nilai 945.0, memberikan ruang eksplorasi yang cukup pada awal pencarian. `coolingRate` dengan nilai 0.9999 menentukan laju penurunan suhu setiap iterasi, yang menyebabkan suhu menurun secara bertahap dan mempersempit ruang pencarian seiring waktu. `probabilityThreshold` bernilai 0.9999 mengatur ambang penerimaan solusi yang lebih buruk untuk mencegah algoritma terjebak di local optimum. `maxIteration` sebesar 100,000 adalah batas maksimum jumlah iterasi yang diperbolehkan untuk proses pencarian solusi.

Pada tahap awal, fungsi menghasilkan initial state menggunakan fungsi `GenerateSuccessor` dari paket `lib`. State ini disimpan dalam `currentState`, dan nilai Objective Function dari state tersebut dihitung untuk menilai kualitasnya. Variabel `stuck` digunakan untuk menghitung frekuensi algoritma terjebak di solusi yang lebih buruk. Proses iterasi terus berjalan selama jumlah iterasi belum mencapai batas `maxIteration`. Pada setiap iterasi, suhu `temperature` dihitung dengan mengalikan suhu awal dengan laju pendinginan (`coolingRate`) yang dipangkatkan sesuai jumlah iterasi, sehingga suhu menurun secara eksponensial. Jika `temperature` mencapai nol, proses berhenti.

Algoritma mencari neighbor state baru menggunakan fungsi `RandomNeighbor`, yang mengembalikan neighbor dan nilai objective function dari neighbor tersebut. Jika nilai objective function dari neighbor lebih tinggi dari current state, neighbor tersebut langsung diterima sebagai current state baru. Namun, jika nilainya lebih rendah, algoritma menghitung probabilitas penerimaan menggunakan fungsi `acceptanceProbability` berdasarkan perbedaan nilai objective function

dan temperature saat ini. Jika probabilitas tersebut melampaui probabilityThreshold, neighbor yang lebih buruk dapat diterima sebagai solusi untuk menghindari jebakan local optimum.

Data probabilitas penerimaan untuk setiap iterasi disimpan dalam plotData, yang nantinya digunakan untuk analisis visual. Setelah semua iterasi selesai atau suhu mencapai nol, waktu eksekusi dihitung. Hasil akhir algoritma, termasuk nilai objective function terakhir, waktu eksekusi, jumlah iterasi, frekuensi stuck, initial state, last state, dan data grafik disimpan dalam format JSON menggunakan fungsi SaveToJson.

2.3.5.2 Source Code Simulated Annealing

Berikut adalah source code dari fungsi Simulated Annealing.

```

1 package Algorithm
2
3 import (
4     "magic-cube/lib"
5     "math"
6     "strconv"
7     "time"
8 )
9
10 const (
11     startingTemperature = 945.0
12     coolingRate          = 0.9999
13     probabilityThreshold = 0.9999
14     maxIteration        = 100000
15 )
16
17 func SimulatedAnnealing() bool {
18     // Init
19     initialState := lib.GenerateSuccessor()
20     lib.PrintStateWithLabel(initialState, "First State")
21     stateMap := map[[125]int]bool{}
22     var plotData = make([][]float64, 0)
23
24     // Set the current state to the initial state
25     currentState := initialState
26     stateMap[currentState] = true
27     currentCost := lib.ObjectiveFunction(currentState)
28
29     // Variables
30     stuck := 0
31     var neighbour [125]int
32     var neighborCost int
33     var temperature float64
34
35     currentTime := time.Now()
36     iteration := 1
37     for iteration <= maxIteration {
38         // Cool the temperature
39         temperature = startingTemperature * math.Pow(coolingRate, float64(iteration))
40         if temperature-0.1 < 0 {
41             break
42         }
43
44         // Get a new neighbour
45         neighbour, neighborCost = lib.RandomNeighbor(currentState, &stateMap)
46         if neighborCost == -1 {
47             break
48         }
49         stateMap[neighbour] = true
50
51         // Calculate the cost difference
52         deltaE := neighborCost - currentCost
53
54         // If the new solution is better, accept it
55         if deltaE > 0 {
56             currentState = neighbour
57             currentCost = neighborCost
58         } else { // If the new solution is worse, accept it with a probability
59             stuck++
60
61             probability := acceptanceProbability(currentCost, neighborCost, temperature)
62             plotData = append(plotData, [][]float64{float64(iteration), probability})
63
64             // If the probability is greater than the threshold, accept the new solution
65             if probability > probabilityThreshold {
66                 currentState = neighbour
67                 currentCost = neighborCost
68             }
69         }
70     }
71
72     iteration++
73 }

```

```

1  executeTime := time.Since(currentTime).Milliseconds()
2  firstState := lib.ConvertToResult(initialState)
3  lastState := lib.ConvertToResult(currentState)
4
5  lib.PrintStateWithLabel(currentState, "Last State")
6  res := map[string]interface{}{
7      "algorithm": "Simulated Annealing",
8      "description": map[string]interface{}{
9          "Objective Function": currentCost,
10         "Duration":          strconv.FormatInt(executeTime, 10) + "ms",
11         "Jumlah Iterasi":     iteration,
12         "Frekuensi Stuck":   stuck,
13     },
14     "firstState": firstState,
15     "lastState":  lastState,
16     "plotData":   plotData,
17 }
18
19 lib.SaveToJson(res)
20 return true
21 }
22
23 func acceptanceProbability(currentCost int, neighborCost int, temperature float64) float64 {
24     if temperature == 0 {
25         return 0
26     }
27
28     probability := math.Exp(float64(neighborCost-currentCost) / temperature)
29     if math.IsInf(probability, 0) {
30         return 1
31     }
32
33     return probability
34 }

```

2.3.6 Genetic Algorithm

Genetic Algorithm adalah algoritma yang terinspirasi dari proses evolusi alam. Mirip dengan proses biologis, algoritma ini bekerja dengan populasi solusi potensial yang mengalami *selection*, *mutation*, dan *cross-over* untuk mencari solusi optimal. Seperti pada algoritma optimisasi lainnya, tujuan utama *genetic algorithm* adalah menemukan solusi terbaik dari sebuah ruang solusi yang besar.

Pada *genetic algorithm*, *initial state* diwakili dari beberapa (jumlahnya sudah ditentukan) "individu" dalam suatu populasi. Satu individu pada *genetic algorithm* yang dimaksud adalah satu state. Algoritma akan terus menerus mengembangkan populasi ini melalui proses seperti *selection* (memilih individu secara random dengan menggunakan roulette wheel), *crossover* (menggabungkan dua parent state untuk menghasilkan individu baru), dan *mutation* (melakukan penukaran pada dua cell dalam suatu state untuk memperkenalkan variasi). Setiap iterasi atau generasi diharapkan membawa

individu baru yang lebih baik dibanding generasi sebelumnya. Algoritma ini akan berhenti jika sudah menemukan solusi yang optimal atau pencarian berlangsung terlalu lama (melewati batasan jumlah generasi).

Berikut adalah proses pencarian solusi *Diagonal Magic Cube* dengan menggunakan Genetic Algorithm.

2.3.6.1 Deskripsi Kelas/Fungsi Genetic Algorithm

Fungsi GeneticAlgorithm adalah implementasi Genetic Algorithm yang bertujuan mencari solusi optimal melalui proses evolusi, dengan menggabungkan populasi solusi dan mekanisme seperti seleksi, crossover, dan mutasi. Fungsi ini menerima dua parameter, yaitu jumlah_populasi, yang menentukan jumlah individu dalam satu populasi, dan jumlah_generasi, yang menentukan jumlah iterasi atau generasi maksimal yang dapat dibangkitkan. Di awal fungsi, populasi awal dibangkitkan menggunakan fungsi GeneratePopulation dari package lib, dan inisialisasi variabel dilakukan dengan menyimpan state terbaik awal (bestFirstState), state terbaik secara keseluruhan (bestState), dan nilai cost tertinggi (bestCost).

Dalam loop utama, yang berjalan selama jumlah iterasi kurang dari jumlah generasi, sebuah generasi baru dibentuk melalui beberapa tahap. Pertama, nilai fitness atau objective function dihitung untuk setiap individu dalam populasi saat ini, yang disimpan dalam array cost. Jika iterasi bukan merupakan iterasi yang pertama, individu dengan nilai fitness tertinggi di dalam populasi saat ini akan diperiksa, dan bestState serta bestCost diperbarui jika individu ini memiliki nilai cost lebih tinggi daripada state terbaik yang ditemukan sebelumnya. Jika iterasi saat ini adalah iterasi pertama, bestFirstState diperbarui dengan individu yang memiliki nilai cost tertinggi dalam populasi awal.

Setelah menghitung nilai fitness, algoritma melakukan proses seleksi menggunakan mekanisme roulette wheel, yaitu memilih individu secara acak berdasarkan nilai cost-nya. Individu yang terpilih akan disimpan dalam selectedPopulasi, yang nantinya akan digunakan dalam tahap crossover.

Selanjutnya, algoritma melakukan crossover atau penggabungan dua solusi (individu) untuk menghasilkan solusi baru. Dalam proses ini,

dua parent dipilih secara berurutan, kemudian masing-masing parent dibagi menjadi dua bagian berdasarkan titik crossover (crossover point) yang bernilai konstan, yaitu di indeks ke-75. Dua *child* (child1 dan child2) dihasilkan dengan menggabungkan sebagian data dari masing-masing parent, kemudian diperiksa dan ditambahkan data yang tidak ada pada masing-masing anak menggunakan pengecekan dengan fungsi `NotIn` dari package `lib` untuk memastikan tidak ada pengulangan dalam satu kubus. Penambahan data pada salah satu anak dilakukan dengan mencacah keseluruhan isi parent ke-dua dan menambahkan jika ada nilai yang tidak ada pada anak tersebut.

Setelah proses crossover, kedua *child* akan menjalani proses mutasi. Mutasi dilakukan dengan memilih dua posisi acak di dalam masing-masing *child* dan menukar elemen di kedua posisi tersebut. Hasil dari mutasi ini kemudian dimasukkan ke dalam `newGeneration`, yang akan menjadi populasi baru pada iterasi berikutnya.

Setelah generasi baru terbentuk, populasi diperbarui dengan `newGeneration`, dan iterasi bertambah satu hingga mencapai `jumlah_generasi`. Setelah loop selesai, waktu eksekusi dihitung, dan informasi tentang algoritma seperti jumlah populasi awal, jumlah generasi, nilai terbaik (`bestCost`), dan durasi disimpan dalam `map res`, bersama dengan state awal dan state terbaik akhir (`lastState`). Data ini kemudian disimpan sebagai file JSON menggunakan `lib.SaveToJson`, dan fungsi mengembalikan `true` untuk menunjukkan bahwa proses telah selesai.

2.3.6.2 Source Code Genetic Algorithm

```
1 package Algorithm
2
3 import (
4     "fmt"
5     "magic-cube/lib"
6     "time"
7 )
8
9 func GeneticAlgorithm(jumlah_populasi int, jumlah_generasi int) bool {
10     iterasi := 0
11     currentTime := time.Now()
12     populasi := lib.GeneratePopulation(jumlah_populasi)
13     bestFirstState := [125]int{}
14     // populasi_awal := populasi
15     bestState := [125]int{}
16     bestCost := 0
17
18     for iterasi < jumlah_generasi {
19         newGeneration := [][125]int{}
20
21         // Calculate the cost of the state
22         cost := make([]int, jumlah_populasi)
23         sumCost := 0
24         for i := 0; i < jumlah_populasi; i++ {
25             cost[i] = lib.ObjectiveFunction(populasi[i])
26             sumCost += cost[i]
27
28             if iterasi > 0 {
29                 // Get best child of the best child of the best 🧐
30                 if cost[i] > bestCost {
31                     bestCost = cost[i]
32                     bestState = populasi[i]
33                 }
34             }
35         }
36
37         if (iterasi == 0) {
38             tempBestCost := 0
39             tempBestIdx := 0
40             for i := 0; i < jumlah_populasi; i++ {
41                 if (tempBestCost < cost[i]) {
42                     tempBestCost = cost[i]
43                     tempBestIdx = i
44                 }
45             }
46             bestFirstState = populasi[tempBestIdx]
47         }
48
49         // Selection
50         selectedPopulasi := make([][125]int, jumlah_populasi)
51         for i := 0; i < jumlah_populasi; i++ {
52             randPoint := lib.RandomInt(0, sumCost)
53             for j := 0; j < jumlah_populasi; j++ {
54                 randPoint -= cost[j]
55                 if randPoint <= 0 {
56                     selectedPopulasi[i] = populasi[j]
57                     break
58                 }
59             }
60         }
```

```

1 // Crossover
2 for i := 0; i < jumlah_populasi; i += 2 {
3     // Randomly select 2 parents
4     parent1 := selectedPopulasi[i]
5     parent2 := selectedPopulasi[i+1]
6
7     crossoverPoint := 75
8
9     // Create 2 children
10    child1 := [125]int{}
11    child2 := [125]int{}
12
13    // Copy the first part of the parents to the children
14    for j := 0; j < crossoverPoint; j++ {
15        child1[j] = parent1[j]
16        child2[j] = parent2[j]
17    }
18
19    // Copy the second part of the parents to the children
20    currentIndex := 0
21    for j := crossoverPoint; j < 125; j++ {
22        for k := currentIndex; k < 125; k++ {
23            if lib.NotIn(parent2[k], child1) {
24                child1[j] = parent2[k]
25                currentIndex = k
26                break
27            }
28        }
29    }
30    currentIndex = 0
31    for j := crossoverPoint; j < 125; j++ {
32        for k := currentIndex; k < 125; k++ {
33            if lib.NotIn(parent1[k], child2) {
34                child2[j] = parent1[k]
35                currentIndex = k
36                break
37            }
38        }
39    }
40
41    // Mutate the children
42    // Swap two random elements in child1
43    idx1 := lib.RandomInt(0, 124)
44    idx2 := lib.RandomInt(0, 124)
45    child1[idx1], child1[idx2] = child1[idx2], child1[idx1]
46
47    // Swap two random elements in child2
48    idx1 = lib.RandomInt(0, 124)
49    idx2 = lib.RandomInt(0, 124)
50    child2[idx1], child2[idx2] = child2[idx2], child2[idx1]
51
52    newGeneration = append(newGeneration, child1)
53    newGeneration = append(newGeneration, child2)
54
55 }
56
57 populasi = newGeneration
58 iterasi++
59 }
60 executeTime := time.Since(currentTime)
61 fmt.Println("Time: ", executeTime)
62 fmt.Println("Best Cost: ", bestCost)
63
64 // fmt.Println("Best State: ")
65 // lib.PrintState(bestState)
66
67 lastState := lib.ConvertToResult(bestState)
68 firstState := lib.ConvertToResult(bestFirstState)
69
70 res := map[string]interface{}{
71     "algorithm": "Genetic Algorithm",
72     "description": map[string]interface{}{
73         "Jumlah Populasi Awal": jumlah_populasi,
74         "Jumlah Generasi": jumlah_generasi,
75         "Best Value": bestCost,
76         "Duration": executeTime.String(),
77     },
78     "firstState": firstState,
79     "lastState": lastState,
80 }
81
82 lib.SaveToJson(res)
83
84 return true
85 }

```

2.3.7 Kelas/Fungsi Bantuan

2.3.7.1 Deskripsi Kelas/Fungsi Menemukan Highest Neighbor

Fungsi FindHighestNeighbor merupakan fungsi yang digunakan untuk mencari suksesor dengan nilai tertinggi yang kemudian akan dijadikan neighbor dari suatu state. Fungsi menerima input berupa sebuah array state yang terdiri dari 125 elemen, kemudian mencari tetangga terbaik dari state berdasarkan nilai objektif tertinggi. Fungsi ini dimulai dengan mendeklarasikan bestNeighbor, bestCost, dan successorCount. bestNeighbor menyimpan tetangga dengan nilai objektif tertinggi yang ditemukan sejauh ini, sementara bestCost diinisialisasi dengan nilai -1 untuk memastikan setiap nilai yang lebih tinggi dari -1 akan menggantikan nilai tersebut. successorCount bertugas menghitung jumlah tetangga (suksesor) yang telah diperiksa. Dalam proses pencarian, fungsi melakukan dua loop bersarang di mana setiap elemen i pada state ditukar dengan elemen j setelahnya, menghasilkan newNeighbor yang mewakili tetangga baru dari state. Setelah itu, successorCount ditingkatkan dan fungsi ObjectiveFunction digunakan untuk menghitung nilai objektif dari tetangga baru tersebut. Jika nilai objektifnya lebih tinggi daripada bestCost, maka bestCost diperbarui dengan nilai tersebut, dan bestNeighbor diperbarui dengan tetangga baru. Setelah semua kemungkinan pasangan elemen dalam state ditukar sesuai aturan dan diperiksa, fungsi akan mengembalikan bestNeighbor, bestCost, dan successorCount sebagai hasil akhir. Berikut adalah source code dari fungsi FindHighestNeighbor


```

1 func FindHighestNeighbor(state [125]int) ([125]int, int, int) {
2     var bestNeighbor [125]int
3     bestCost := -1
4     successorCount := 0
5
6     for i := 0; i < 125; i++ {
7         for j := i + 1; j < 125; j++ {
8             newNeighbor := state
9             newNeighbor[i], newNeighbor[j] = newNeighbor[j], newNeighbor[i]
10            successorCount++
11
12            cost := ObjectiveFunction(newNeighbor)
13
14            if cost > bestCost {
15                bestCost = cost
16                bestNeighbor = newNeighbor
17            }
18        }
19    }
20
21    return bestNeighbor, bestCost, successorCount
22 }

```

2.3.7.2 Deskripsi Kelas/Fungsi Membangkitkan State secara Random

Fungsi GenerateSuccessor bertujuan untuk menghasilkan state acak dalam bentuk array dengan panjang 125 elemen yang terdiri dari angka 1 hingga 125. Pertama, fungsi mendeklarasikan sebuah array initList yang diisi dengan angka berurutan dari 1 sampai 125. Kemudian, fungsi menggunakan `rand.NewSource(time.Now().UnixNano())` untuk membuat perubahan random yang dilakukan bergantung pada waktu saat itu, memastikan hasil random berbeda setiap kali fungsi dijalankan. Setelah itu, fungsi `rand.Shuffle` digunakan untuk mengacak urutan elemen di dalam initList. Dalam proses ini, setiap elemen di dalam array akan dipertukarkan secara acak dengan elemen lainnya, menghasilkan urutan yang acak dari angka 1 hingga 125. Setelah pengacakan selesai, hasilnya disimpan ke dalam variabel successor, dan fungsi mengembalikan array successor sebagai output, yang merepresentasikan state baru yang urutannya random.

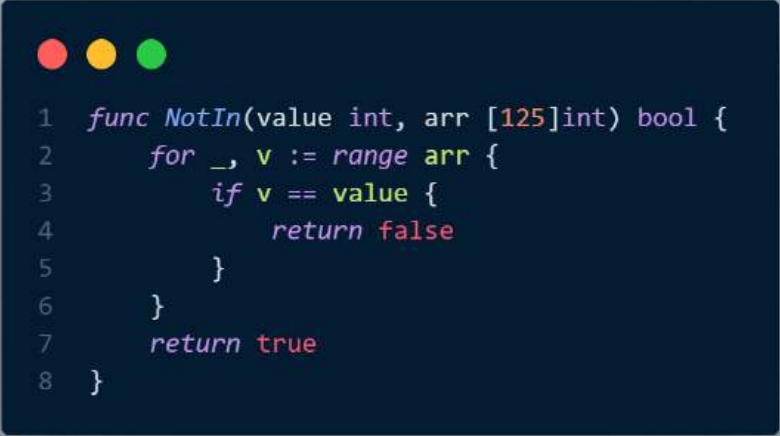
```

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

```

2.3.7.3 Deskripsi Kelas/Fungsi Mengecek Keberadaan dalam State

Fungsi `NotIn` bertujuan untuk memeriksa apakah sebuah nilai (value) tidak terdapat dalam array state. Fungsi ini menerima dua parameter: value yang ingin dicek keberadaannya dan `arr` sebagai array tempat pencarian. Fungsi akan melakukan iterasi melalui setiap elemen dalam `arr` menggunakan variabel `v`. Selama proses iterasi, jika ditemukan elemen dalam `arr` yang memiliki nilai sama dengan value, maka fungsi segera mengembalikan `false`, menunjukkan bahwa nilai tersebut ada dalam array. Jika iterasi selesai tanpa menemukan nilai yang sama, fungsi akan mengembalikan `true`, menandakan bahwa value tidak ada dalam array `arr`.




```

1 func NotIn(value int, arr [125]int) bool {
2     for _, v := range arr {
3         if v == value {
4             return false
5         }
6     }
7     return true
8 }

```

2.3.7.4 Deskripsi Kelas/Fungsi Membangkitkan Populasi

Fungsi `GeneratePopulation` digunakan untuk membangkitkan populasi awal dengan sejumlah state sesuai jumlah yang ditentukan dalam parameter `jumlah_populasi`. Fungsi ini mengembalikan sebuah slice dua dimensi `[125]int` dengan panjang sesuai jumlah populasi yang diinginkan. Di dalam fungsi, slice populasi dibuat menggunakan `make` dengan panjang `jumlah_populasi`, di mana setiap elemen pada slice ini berukuran 125 elemen integer. Fungsi kemudian melakukan iterasi sebanyak `jumlah_populasi`, dan dalam setiap iterasi, memanggil fungsi `GenerateSuccessor` untuk mengisi setiap elemen populasi dengan keadaan acak yang berbeda. Setelah semua elemen populasi terisi dengan keadaan acak, fungsi akan mengembalikan slice populasi tersebut sebagai hasil.



```

1 func GeneratePopulation(jumlah_populasi int) [][]int {
2     // Generate a random initial state
3     populasi := make([]int, jumlah_populasi)
4     for i := 0; i < jumlah_populasi; i++ {
5         populasi[i] = GenerateSuccessor()
6     }
7     return populasi
8 }

```

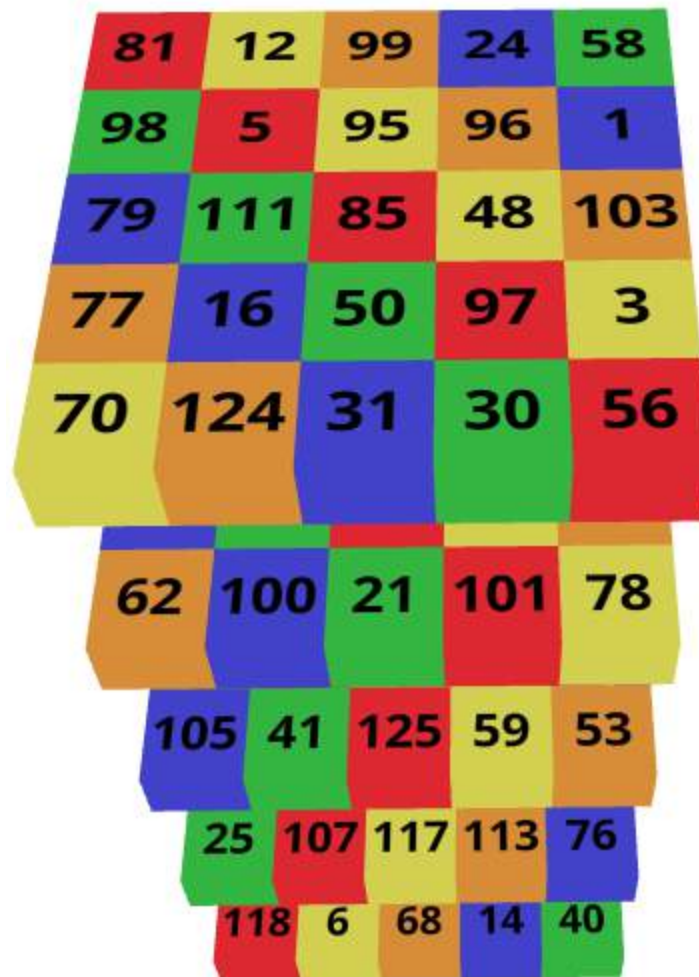
2.4 Hasil Eksperimen dan Analisis

2.4.1 Steepest Ascent Hill-Climbing

a. Percobaan 1

State Awal :

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



State Akhir :

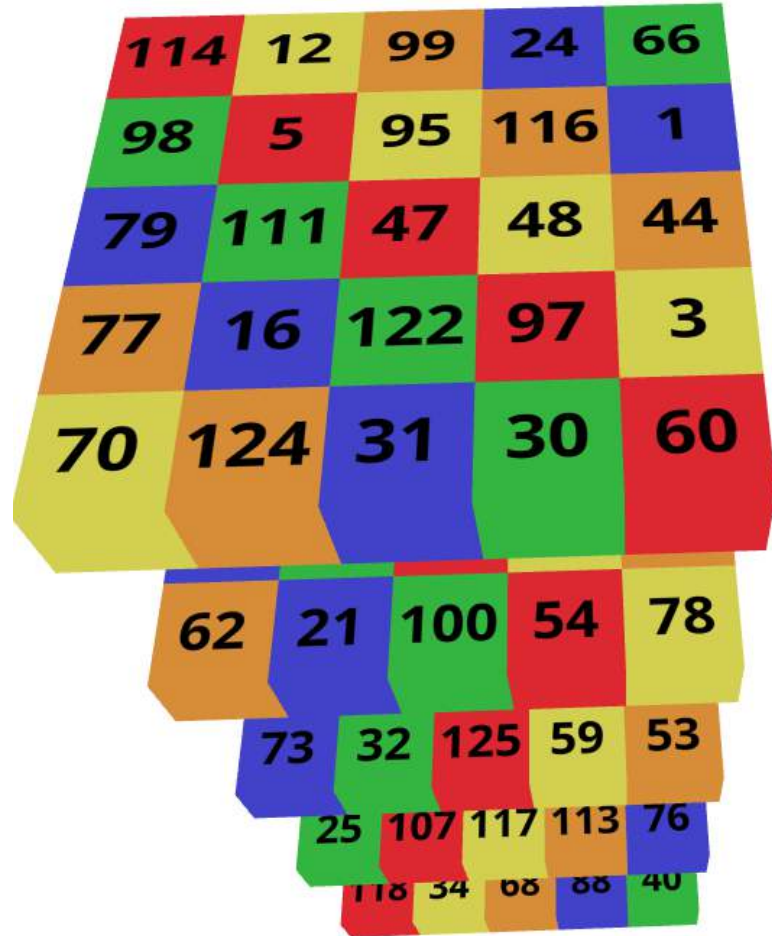
114	12	99	24	66
98	5	95	116	1
79	111	47	48	44
77	16	122	97	3
70	124	31	30	60
72	15	42	90	96
41	81	52	37	104
80	43	82	36	74

105 38 39 55 89
62 21 100 54 78

120 63 102 7 23
45 109 49 2 110
20 101 87 85 19
67 10 123 22 93
73 32 125 59 53

83 29 51 106 112
92 94 11 61 57
50 71 35 8 75
65 14 9 27 86
25 107 117 113 76

28 64 121 84 18
119 56 6 108 26
17 46 58 91 103
33 13 115 4 69
118 34 68 88 40



Keterangan :

Algoritma : Steepest Ascent Algorithm

Duration : 60.7568ms

Jumlah Iterasi : 27

Objective Function : 34

b. Percobaan 2

State Awal :

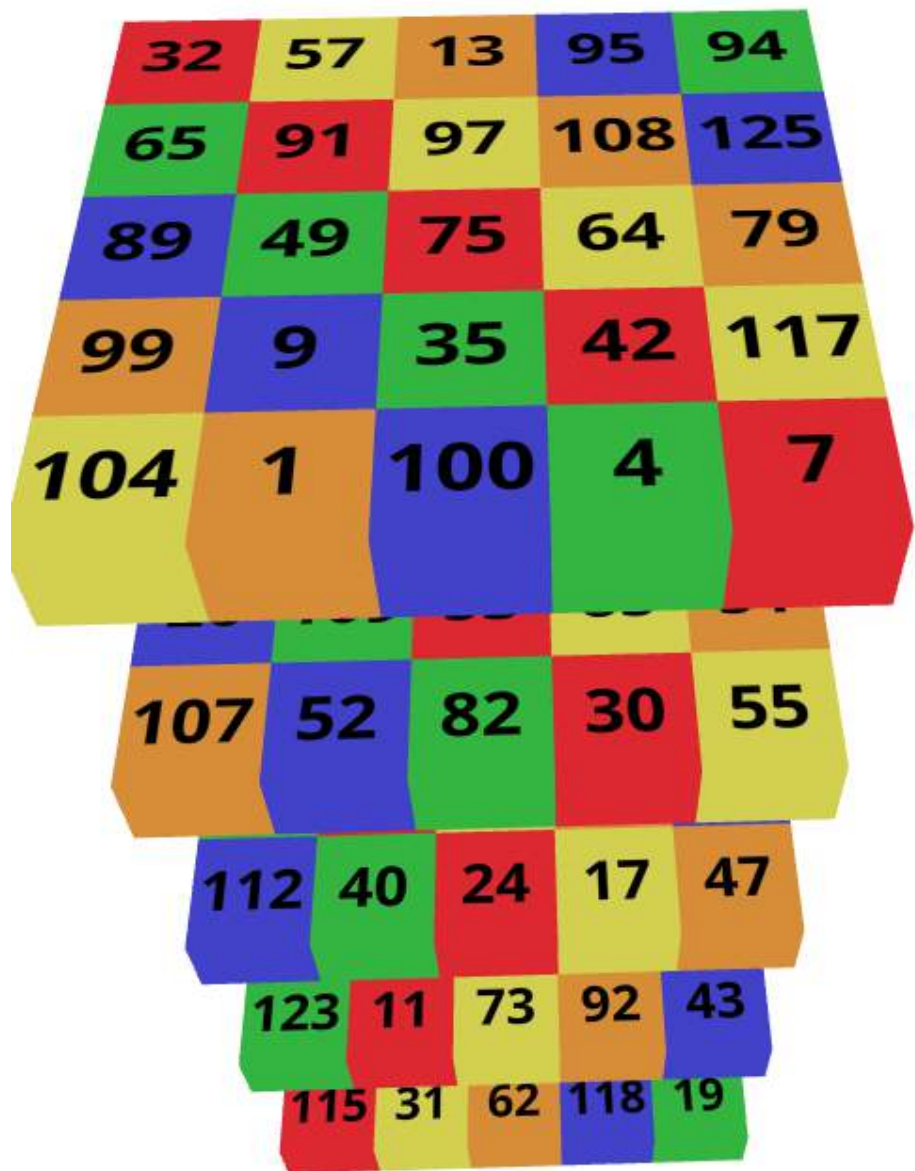
32	57	13	95	94
65	91	97	108	125
89	49	75	64	79
99	9	35	42	117
104	1	100	4	7

41	14	63	71	87
33	23	12	8	119
29	74	48	27	77
26	109	53	85	51
107	52	82	30	55

90	81	66	54	68
86	39	15	124	21
121	3	111	56	98
72	10	38	60	101
112	40	24	17	47

46	103	93	70	44
58	116	37	18	78
28	2	88	110	113
59	50	67	120	96
123	11	73	92	43

45	22	114	36	20
83	6	69	34	84
105	80	106	25	61
5	122	76	16	102
115	31	62	118	19



State akhir :

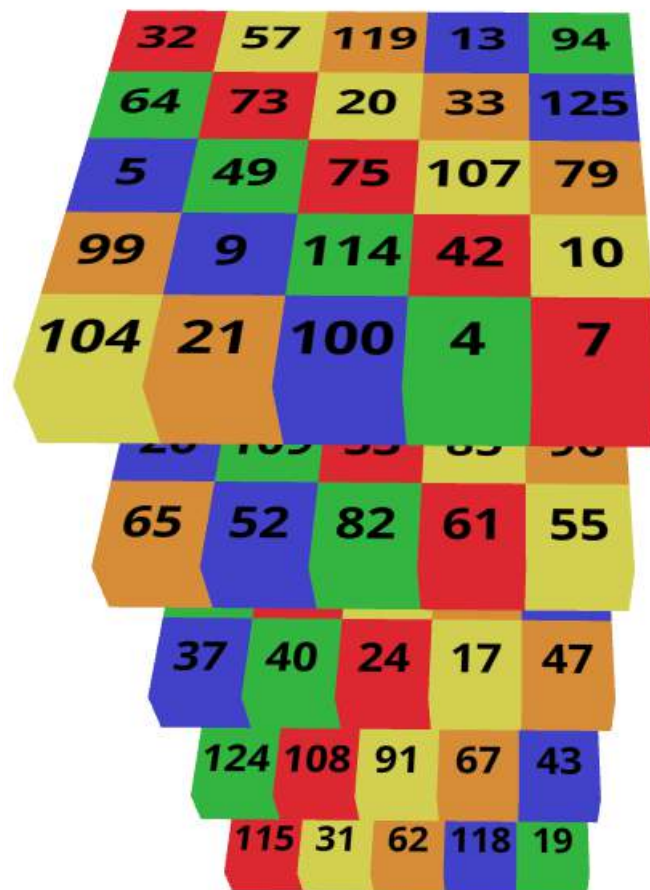
32	57	119	13	94
64	73	20	33	125
5	49	75	107	79
99	9	114	42	10
104	21	100	4	7

41	14	63	110	87
11	23	12	8	27
29	117	111	95	77
26	109	53	85	96
65	52	82	61	55

90	81	22	54	68
86	70	35	123	1
121	3	48	56	98
72	44	38	60	101
37	40	24	17	47

46	103	97	39	30
58	116	112	18	78
28	15	88	71	113
59	50	92	120	51
124	108	91	67	43

6	66	2	122	36
83	45	69	34	84
105	80	106	25	74
89	93	76	16	102
115	31	62	118	19



Keterangan :

Algoritma : **Steepest Ascent
Algorithm**

Duration : **62.5736ms**

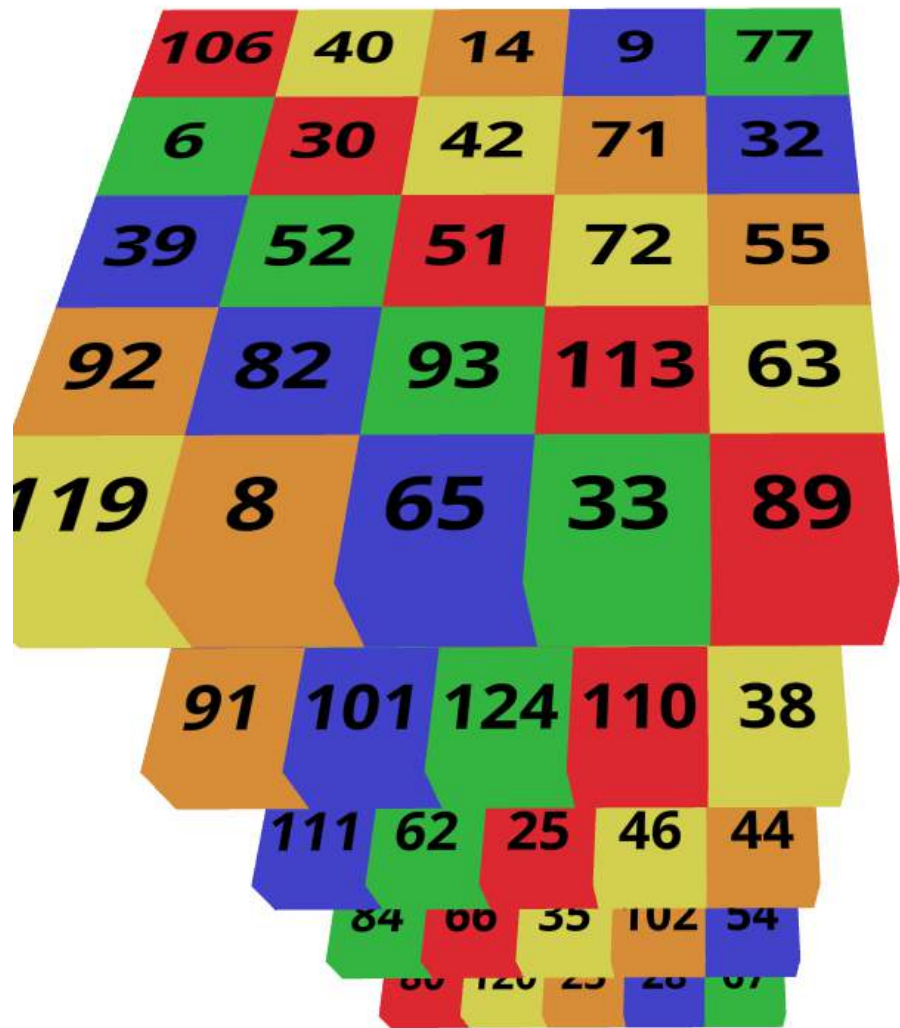
Jumlah Iterasi : **31**

Objective Function : **35**

c. Percobaan 3

State Awal :

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



State Akhir:

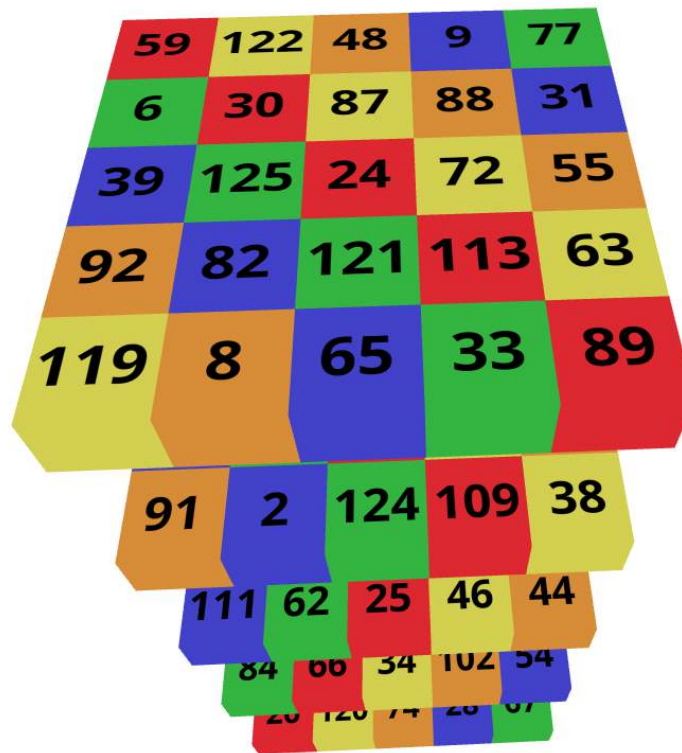
59	122	48	9	77
6	30	87	88	31
39	125	24	72	55
92	82	121	113	63
119	8	65	33	89

20	108	100	7	80
106	86	14	104	5
57	29	61	83	85
41	50	16	22	43
91	2	124	109	38

13	93	42	70	97
115	81	11	53	69
21	60	110	56	68
73	19	23	90	37
111	62	25	46	44

1	75	15	123	101
116	64	112	107	17
96	47	78	79	49
18	99	76	118	94
84	66	34	102	54

114	40	12	51	98
45	71	117	52	105
35	3	36	4	58
95	103	10	27	32
26	120	74	28	67



Keterangan

Algoritma : **Steepest Ascent Algorithm**

Duration : **66.7788ms**

Jumlah Iterasi : **32**

Objective Function : **36**

2.4.2 Hill-Climbing with Sideways Move

a. Percobaan 1

State Awal :

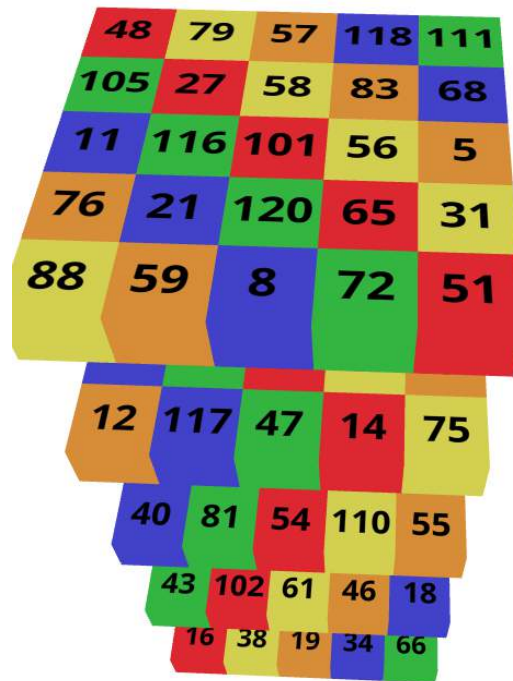
48	79	57	118	111
105	27	58	83	68
11	116	101	56	5
76	21	120	65	31
88	59	8	72	51

73	71	26	63	2
67	52	4	121	62
41	106	91	45	124
96	22	9	23	109
12	117	47	14	75

3	35	7	98	42
85	115	80	95	89
92	125	53	119	94
82	10	1	87	37
40	81	54	110	55

24	99	104	90	78
39	112	114	30	6
17	29	60	93	33
28	100	86	50	77
43	102	61	46	18

69	64	122	20	25
97	108	13	107	113
74	123	84	44	70
49	32	15	103	36
16	38	19	34	66



State Akhir :

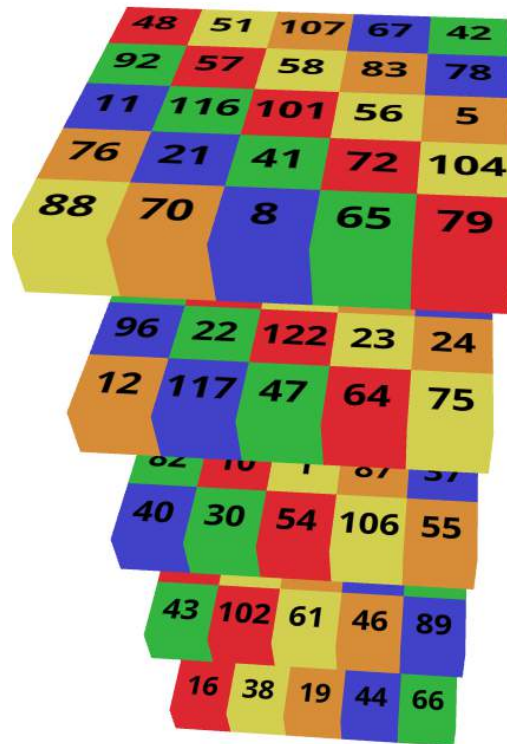
48 51 107 67 42
92 57 58 83 78
11 116 101 56 5
76 21 41 72 104
88 70 8 65 79

74 71 84 17 69
28 52 4 121 110
120 53 91 90 124
96 22 122 23 24
12 117 47 64 75

3 35 7 98 111
85 115 80 95 18
105 125 99 49 94
82 10 1 87 37
40 30 54 106 55

109 62 31 45 68
39 112 77 81 6
100 29 60 93 33
118 63 86 50 114
43 102 61 46 89

2 14 9 97 25
20 108 13 27 103
73 123 26 34 59
119 32 15 113 36
16 38 19 44 66



Keterangan :

Algoritma : Sideways Algorithm

Duration : 2.1268242s

Jumlah Iterasi : 1030

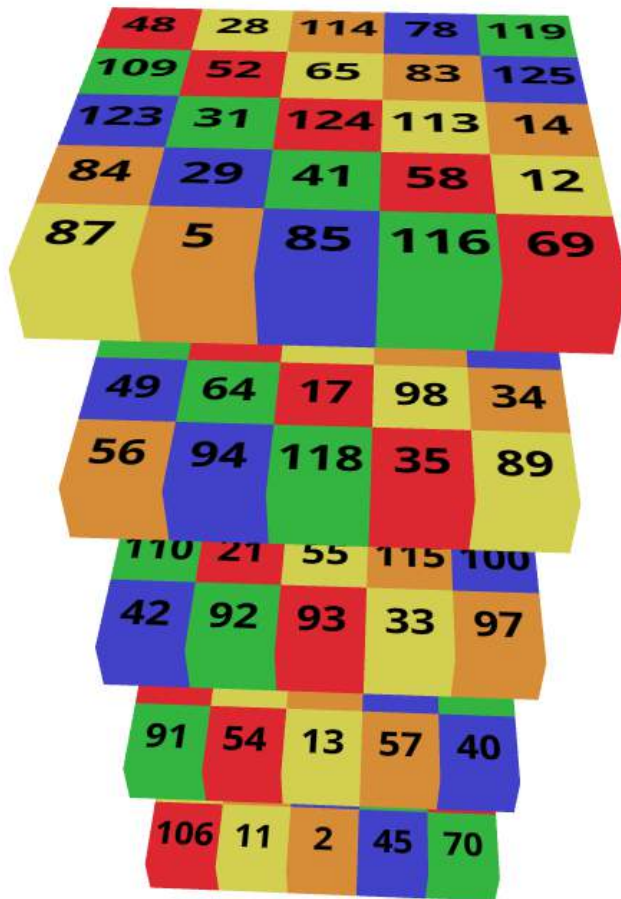
Max Sideways Iteration : 1000

Objective Function : 38

b. Percobaan 2

State Awal :

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



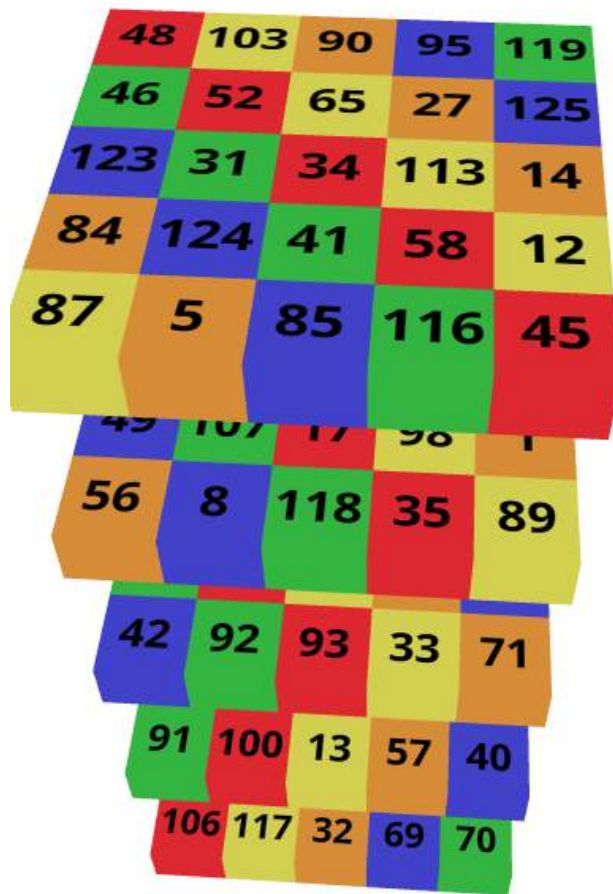
State Akhir :

48	103	90	95	119
46	52	65	27	125
123	31	34	113	14
84	124	41	58	12
87	5	85	116	45
81	9	88	38	99
104	97	83	22	96
25	76	62	122	30
49	107	17	98	1
56	8	118	35	89

77	43	109	59	16
18	51	78	105	11
68	108	20	3	102
110	21	15	115	54
42	92	93	33	71

26	73	60	112	44
74	55	7	64	47
4	120	114	2	75
24	67	121	80	23
91	100	13	57	40

66	101	29	82	37
61	79	53	86	36
39	10	28	6	94
63	111	19	72	50
106	117	32	69	70



Keterangan :

Algoritma : **Sideways Algorithm**

Duration : **1m42.4728199s**

Jumlah Iterasi : **50031**

Max Sideways Iteration : **50000**

Objective Function : **39**

c. Percobaan 3

State Awal :

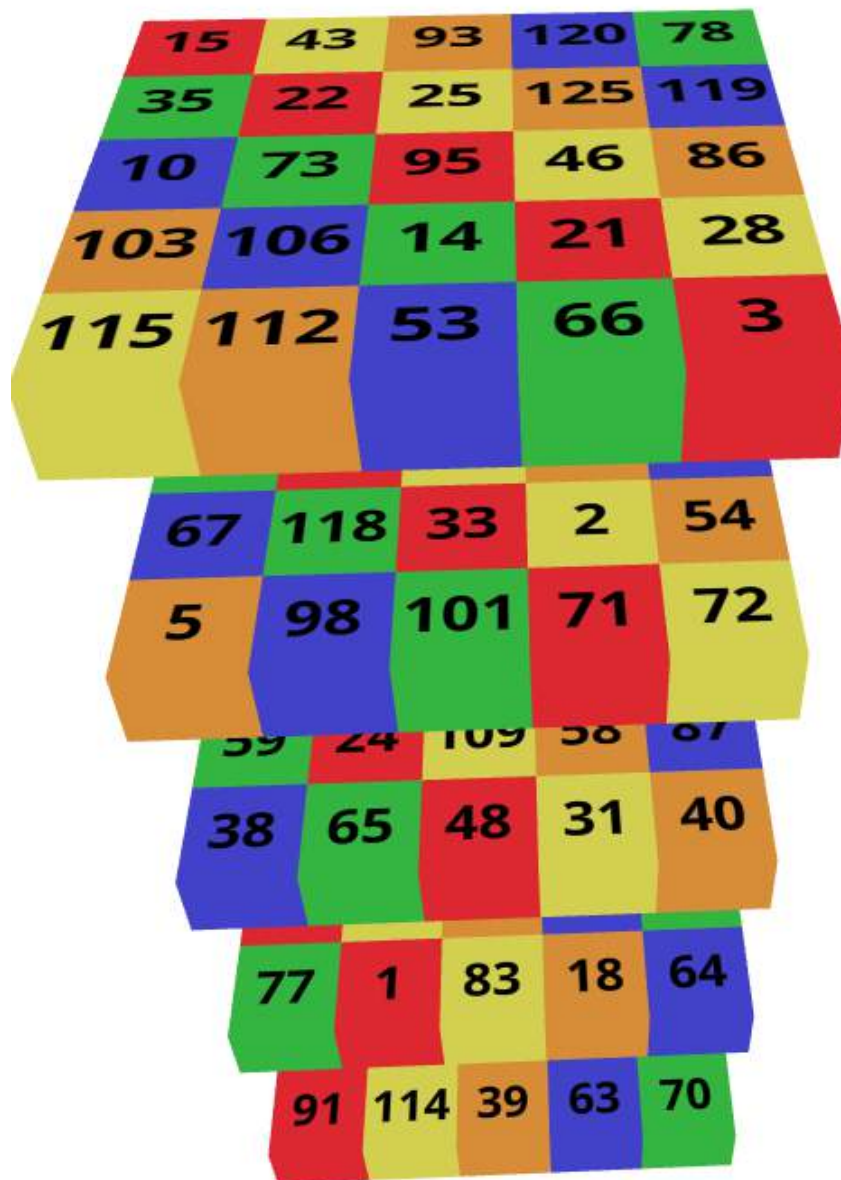
15	43	93	120	78
35	22	25	125	119
10	73	95	46	86
103	106	14	21	28
115	112	53	66	3

75	20	27	44	49
61	16	84	96	116
30	111	80	74	41
67	118	33	2	54
5	98	101	71	72

108	122	57	94	6
17	124	88	7	50
76	34	37	45	121
59	24	109	58	87
38	65	48	31	40

51	13	36	113	85
92	110	4	29	89
19	90	100	55	97
102	9	56	32	26
77	1	83	18	64

42	68	23	107	105
104	117	82	99	60
47	69	11	123	81
52	12	79	62	8
91	114	39	63	70



State Akhir :

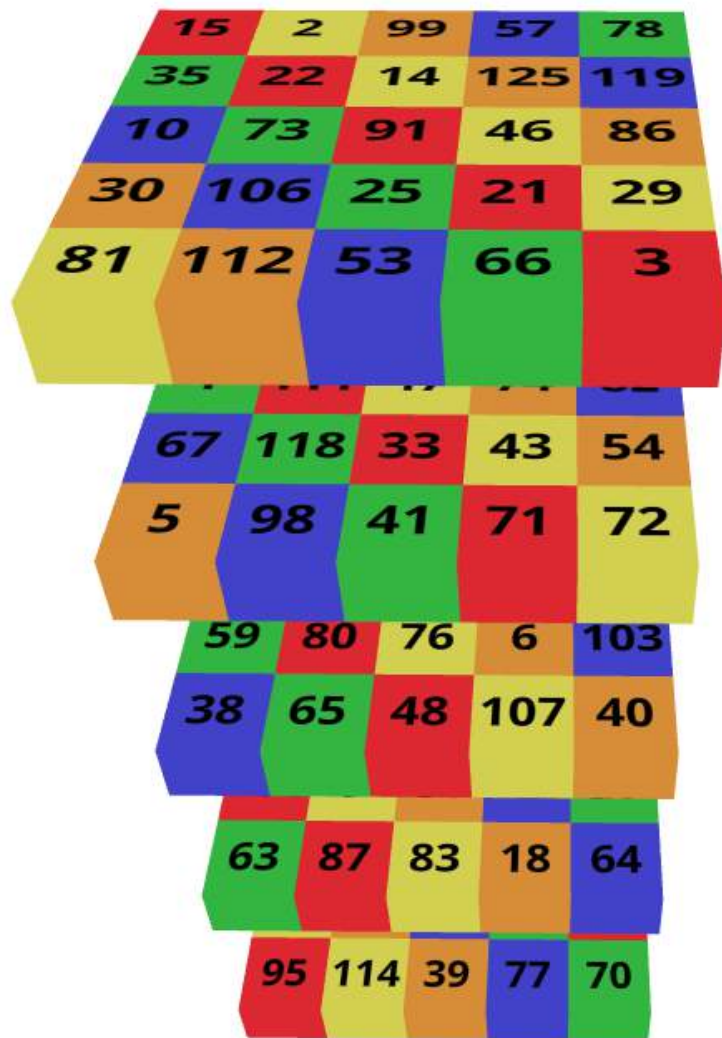
15	2	99	57	78
35	22	14	125	119
10	73	91	46	86
30	106	25	21	29
81	112	53	66	3

105 20 110 31 49
61 16 84 96 58
1 111 47 74 82
67 118 33 43 54
5 98 41 71 72

108 122 92 121 28
17 124 88 36 50
93 34 37 45 94
59 80 76 6 103
38 65 48 107 40

97 13 7 113 85
32 27 69 116 89
19 90 100 55 51
104 9 56 120 26
63 87 83 18 64

42 68 123 44 75
115 117 101 109 60
11 4 24 23 102
52 12 79 62 8
95 114 39 77 70



Keterangan :

Algoritma : Sideways Algorithm

Duration : 89.2048ms

Jumlah Iterasi : 44

Max Sideways Iteration : 10

Objective Function : 41

2.4.3 Random Restart Hill-Climbing

a. Percobaan 1

State Awal :

```
93 95 1 123 50
41 37 74 21 5
105 4 102 13 28
20 107 111 62 60
97 38 90 115 44

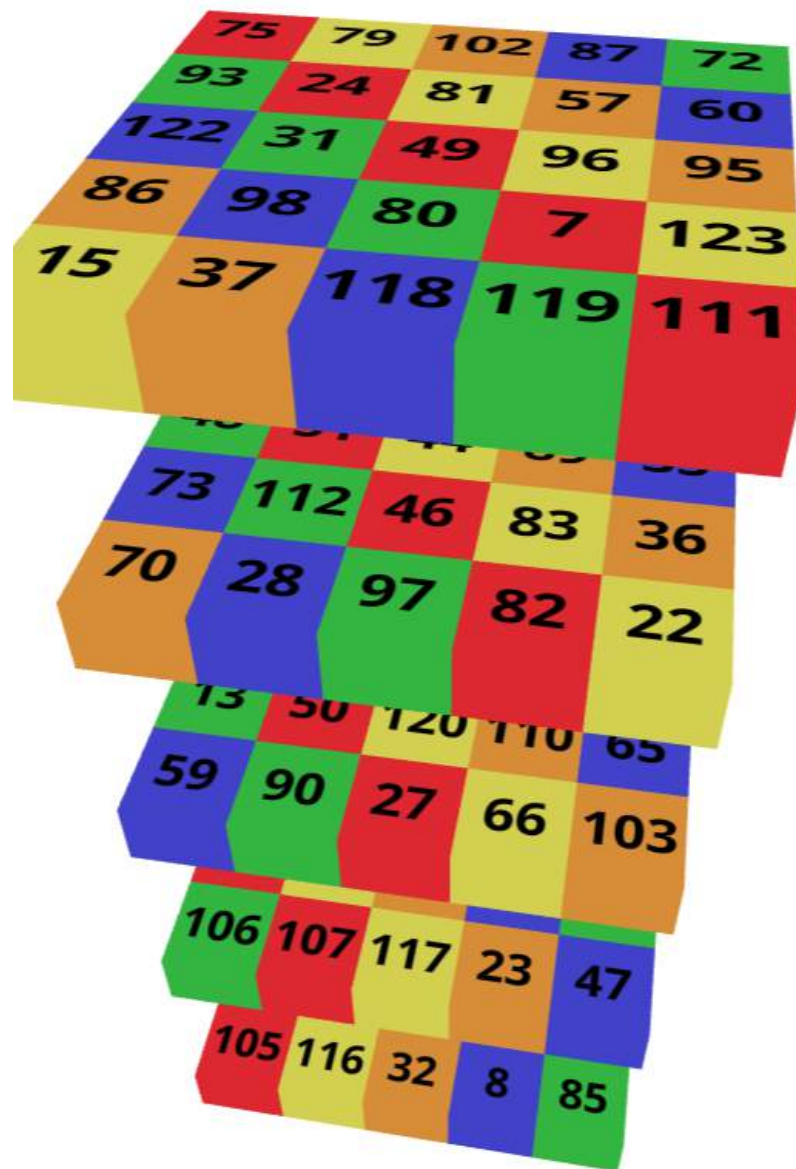
86 46 94 83 31
100 80 121 118 108
26 57 68 122 109
101 25 59 69 79
125 51 33 85 82

116 96 58 117 34
88 61 32 14 39
66 15 22 40 98
103 84 24 64 113
29 23 19 3 110

89 10 67 52 112
47 124 91 92 16
```

120 35 48 106 104
99 73 43 71 17
12 2 42 30 81

49 54 119 36 72
55 87 114 6 53
76 63 11 18 7
65 9 8 75 56
70 27 45 78 77



State Akhir :

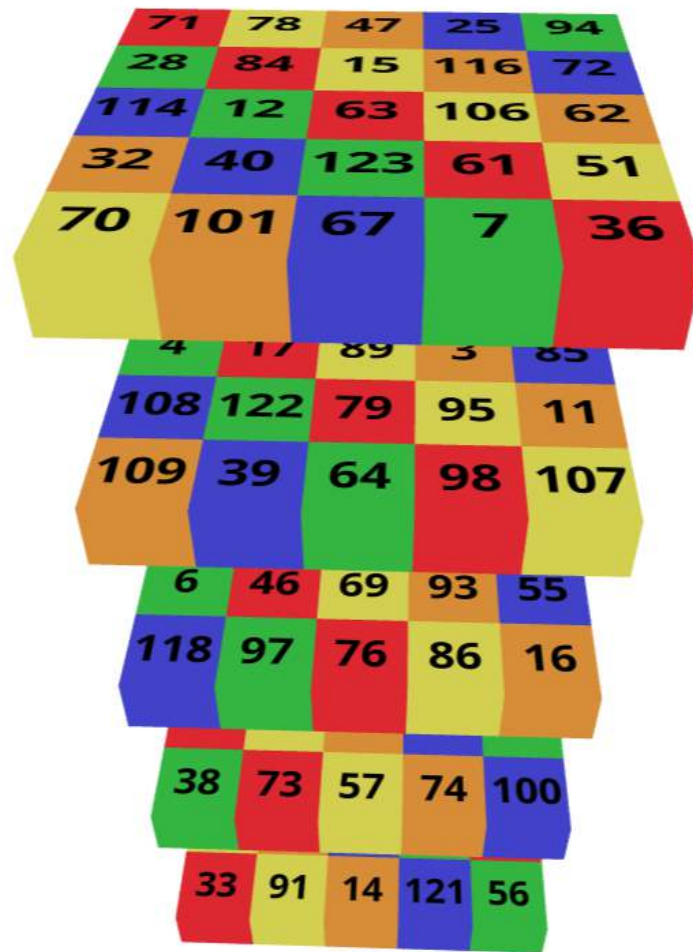
71 78 47 25 94
28 84 15 116 72
114 12 63 106 62
32 40 123 61 51
70 101 67 7 36

19 60 24 42 13
75 5 59 77 99
4 17 89 3 85
108 122 79 95 11
109 39 64 98 107

112 110 58 1 49
31 44 35 52 20
48 18 50 83 10
6 46 69 93 55
118 97 76 86 16

43 53 2 92 125
124 66 29 23 103
22 96 102 54 41
88 27 80 9 111
38 73 57 74 100

68 8 90 115 34
65 120 81 113 21
30 105 26 37 117
119 82 104 45 87
33 91 14 121 56



Keterangan :

Deskripsi

Algoritma : **Random Restart Algorithm**

Duration : **1m6.2026716s**

Jumlah Iterasi : **30817**

Jumlah Restart : **1000**

Objective Function : **46**

b. Percobaan 2

State Awal :

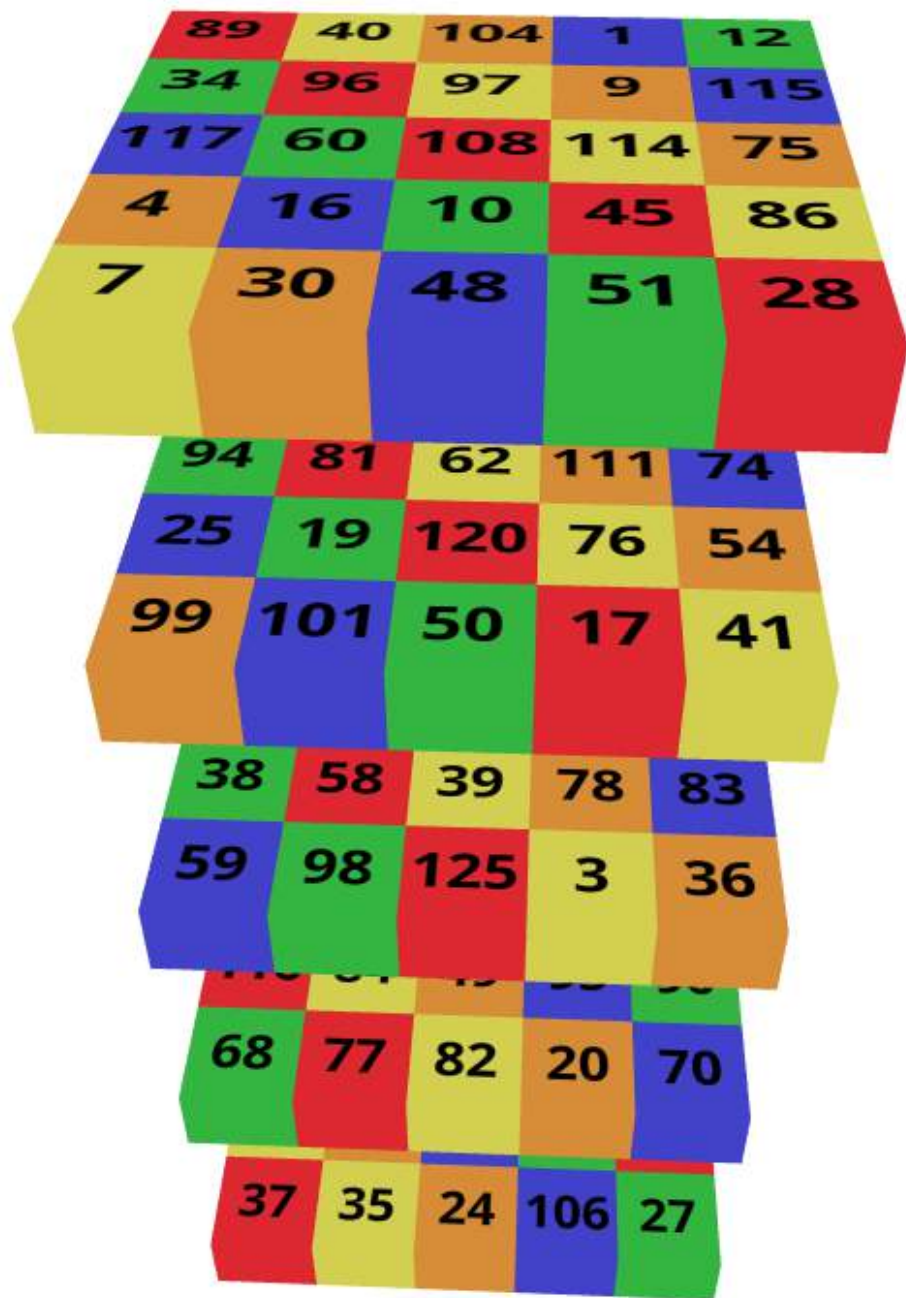
```
89 40 104 1 12
34 96 97 9 115
117 60 108 114 75
4 16 10 45 86
7 30 48 51 28

31 26 47 22 32
109 63 105 119 53
94 81 62 111 74
25 19 120 76 54
99 101 50 17 41

67 107 66 5 52
64 21 61 87 92
42 57 15 29 56
38 58 39 78 83
59 98 125 3 36

100 80 122 72 88
13 93 11 85 102
124 118 123 55 18
116 84 49 95 90
68 77 82 20 70

91 33 110 113 44
23 103 14 46 2
121 65 6 71 73
112 43 8 69 79
37 35 24 106 27
```

State Akhir :

93 51 67 22 55
27 78 90 87 33
106 44 23 123 19
56 34 125 2 121
116 108 10 81 69

25 84 110 61 35
50 115 1 74 75
89 26 70 4 107
53 32 36 77 13
62 58 95 72 28

20 80 57 92 66
6 100 68 38 96
91 16 119 49 73
113 7 40 52 103
85 112 94 120 24

76 60 54 3 122
47 42 104 11 39
99 48 9 118 41
64 102 114 82 83
71 63 86 65 30

8 117 46 97 37
98 45 12 59 101
109 18 21 124 43
5 15 79 111 105
17 31 14 88 29

93	51	67	22	55
27	78	90	87	33
106	44	23	123	19
56	34	125	2	121
116	108	10	81	69
53	32	36	77	13
62	58	95	72	28
113	7	40	52	103
85	112	94	120	24
71	63	86	65	30
17	31	14	88	29

Keterangan :

Algoritma : **Random Restart Algorithm**

Duration : **6.3582937s**

Jumlah Iterasi : **3103**

Jumlah Restart : **100**

Objective Function : **42**

c. Percobaan 3

State Awal :

```
59 20 122 4 28
117 84 100 30 115
53 43 34 33 51
124 111 95 87 99
82 113 85 8 9

70 112 46 67 90
120 57 118 58 75
98 19 72 42 93
89 55 11 125 49
64 60 17 63 7

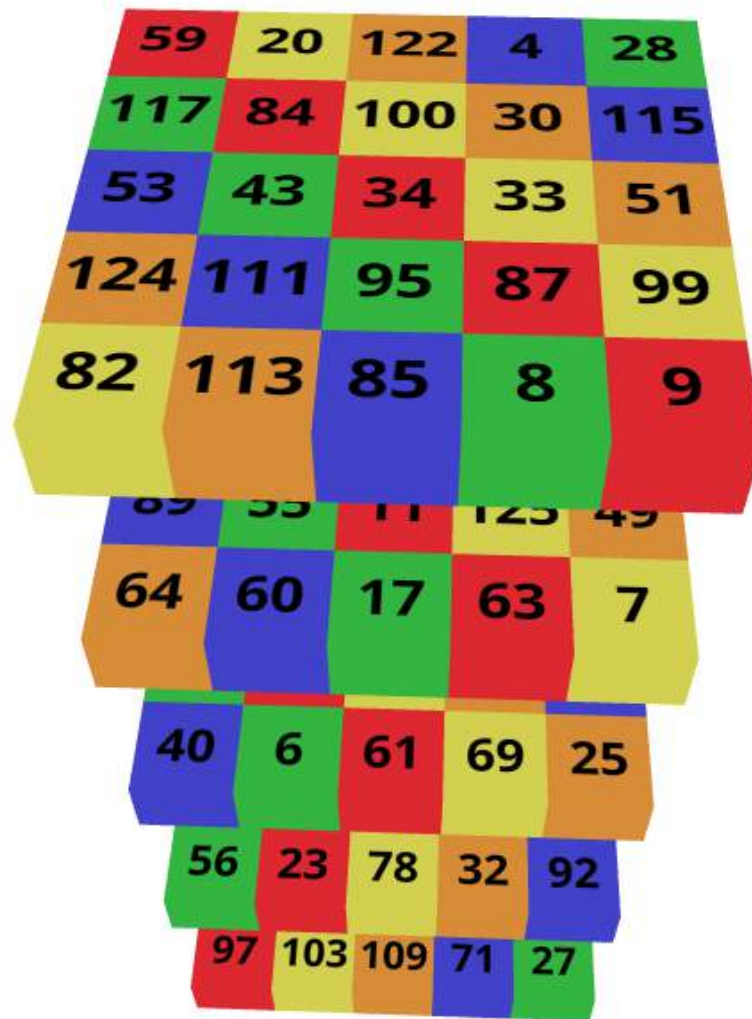
10 106 16 83 41
86 123 114 54 77
76 119 65 45 47
31 74 36 105 29
40 6 61 69 25

108 107 3 13 88
2 22 1 44 38
116 68 35 96 79
91 104 14 52 73
56 23 78 32 92
```

```

12 62 5 110 15
94 66 39 50 24
26 37 81 102 48
80 18 101 121 21
97 103 109 71 27

```



State Akhir :

```

121 80 120 83 12
117 7 69 25 97
125 96 5 114 30
67 32 78 72 66
102 100 92 9 110

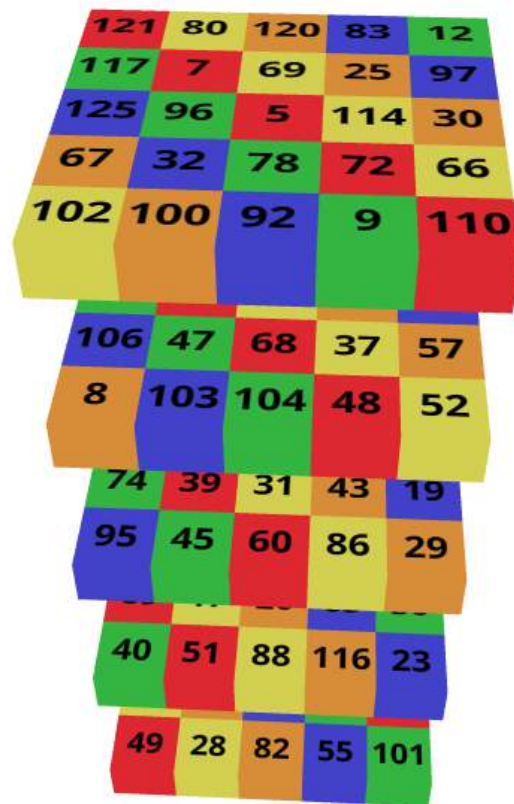
```

93 21 108 58 35
18 71 34 20 81
90 98 1 122 4
106 47 68 37 57
8 103 104 48 52

105 65 13 59 73
3 85 124 33 70
38 119 53 94 11
74 39 31 43 19
95 45 60 86 29

27 123 2 76 87
115 118 61 14 56
44 6 84 46 113
89 17 26 63 36
40 51 88 116 23

109 24 41 42 99
62 112 10 77 54
16 22 75 91 111
79 15 64 50 107
49 28 82 55 101



Keterangan :

Algoritma : Random Restart Algorithm

Duration : 713.5364ms

Jumlah Iterasi : 320

Jumlah Restart : 10

Objective Function : 41

2.4.4 Stochastic Hill-Climbing

a. Percobaan 1

State Awal :

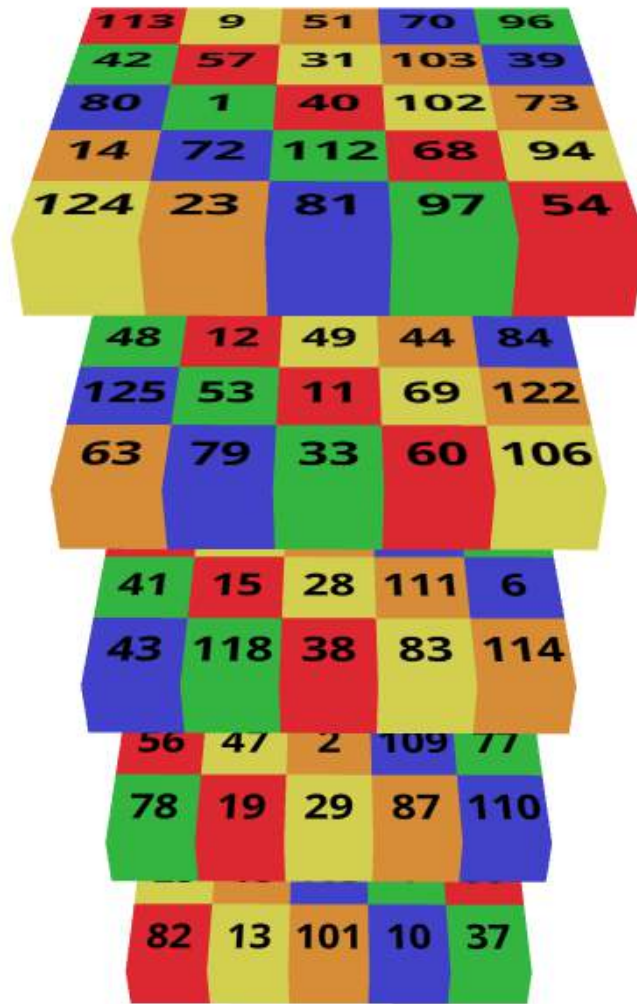
```
113 9 51 70 96
42 57 31 103 39
80 1 40 102 73
14 72 112 68 94
124 23 81 97 54

71 34 117 67 55
93 17 5 95 108
48 12 49 44 84
125 53 11 69 122
63 79 33 60 106

74 62 121 91 88
8 90 3 92 58
22 65 24 50 120
41 15 28 111 6
43 118 38 83 114

89 64 66 20 116
52 32 45 115 119
100 30 46 85 104
56 47 2 109 77
78 19 29 87 110

75 98 21 16 61
99 26 123 76 107
35 36 59 27 7
25 18 105 4 86
82 13 101 10 37
```

State Akhir :

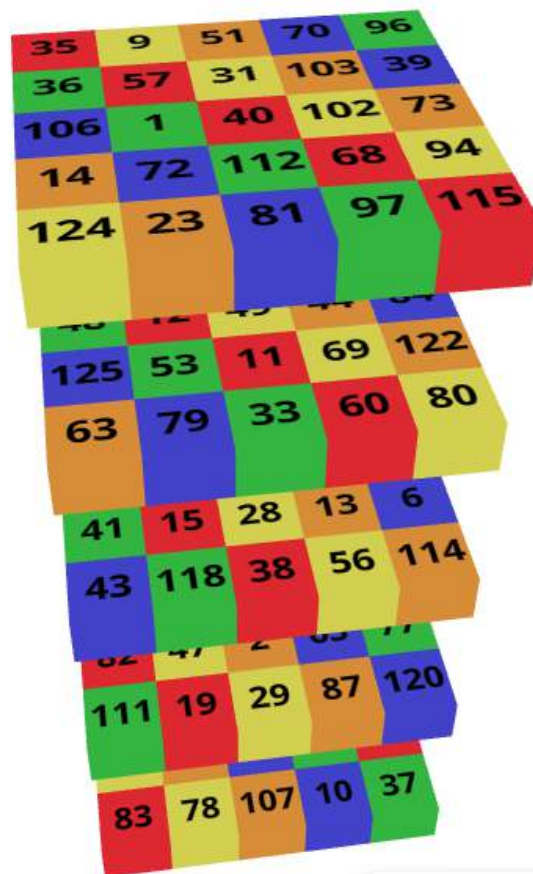
```
35 9 51 70 96
36 57 31 103 39
106 1 40 102 73
14 72 112 68 94
124 23 81 97 115
```

```
71 34 117 67 55
93 17 105 95 108
48 12 49 44 84
125 53 11 69 122
63 79 33 60 80
```

74 62 121 91 88
8 90 3 92 58
22 109 24 50 110
41 15 28 13 6
43 118 38 56 114

89 64 66 20 116
52 32 45 54 119
100 30 46 85 104
82 47 2 65 77
111 19 29 87 120

75 98 21 16 61
99 26 123 76 101
113 42 59 27 7
25 18 5 4 86
83 78 107 10 37



Keterangan :

Algoritma : **Stochastic Hill Climbing**

Duration : **0 ms**

Jumlah Iterasi : **1001**

Objective Function : **14**

b. Percobaan 2

State Awal :

```
113 74 87 6 10
69 56 65 78 115
85 89 100 52 88
15 53 107 33 57
83 5 118 17 42

122 36 61 70 2
35 22 103 63 46
28 51 32 9 25
43 84 24 67 68
49 8 106 47 58

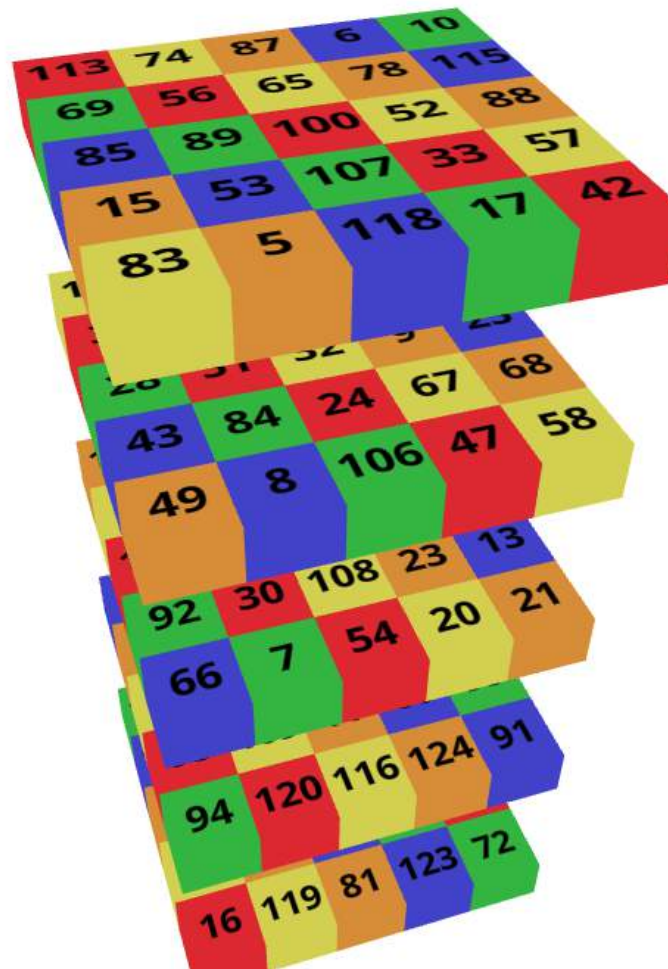
125 110 29 55 99
41 45 40 3 64
101 86 90 96 102
92 30 108 23 13
66 7 54 20 21

98 60 97 114 27
38 34 14 105 62
79 19 26 37 1
93 109 11 80 95
94 120 116 124 91
```

```

104 59 73 18 112
121 111 4 31 77
75 50 12 82 71
44 48 39 117 76
16 119 81 123 72

```



State Akhir :

```

113 12 87 93 10
109 56 47 69 34
85 66 100 52 88
15 53 107 4 57
83 55 118 17 42

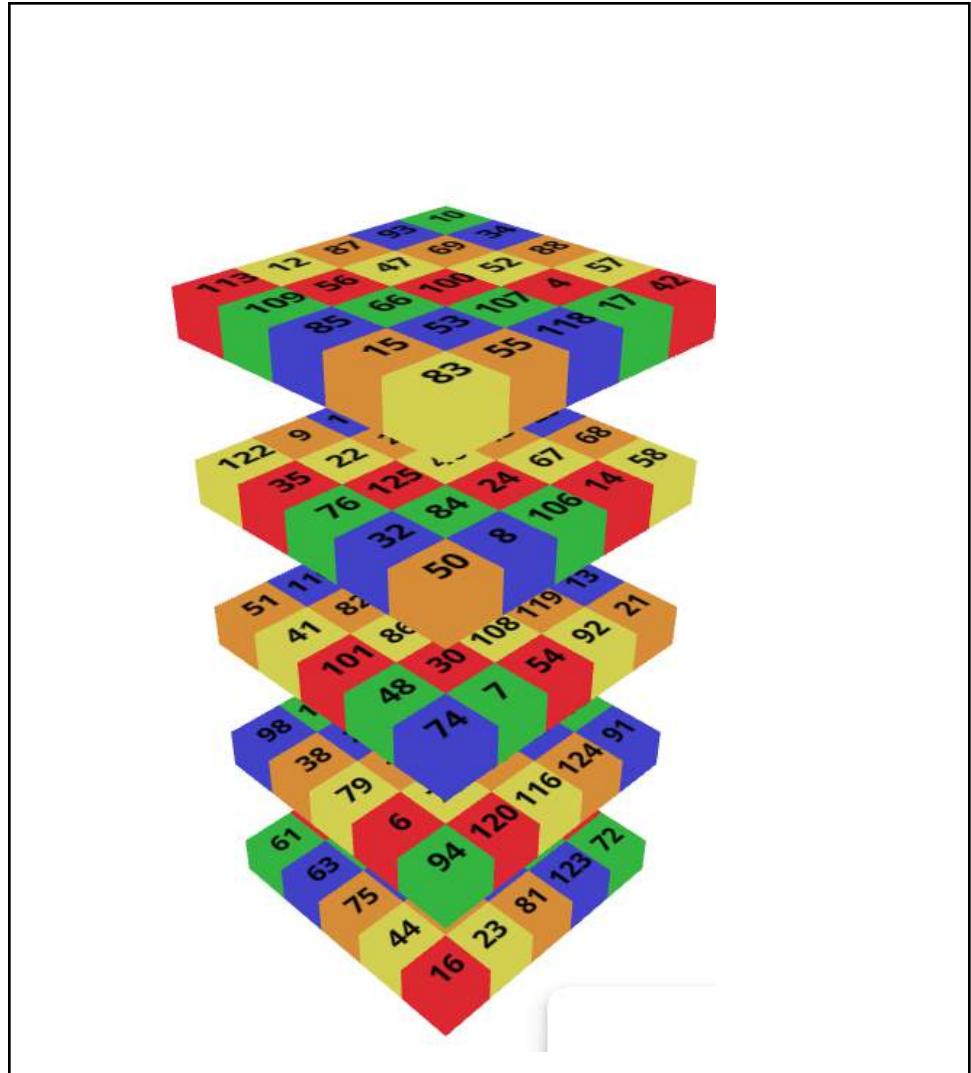
```

122 9 112 70 2
35 22 27 121 36
76 125 46 43 25
32 84 24 67 68
50 8 106 14 58

51 110 1 5 115
41 82 62 3 64
101 86 90 96 102
48 30 108 119 13
74 7 54 92 21

98 103 97 114 60
38 20 65 117 40
79 19 26 37 29
6 78 11 80 95
94 120 116 124 91

61 59 73 18 104
63 111 33 31 77
75 49 89 45 71
44 99 39 105 28
16 23 81 123 72



Keterangan :

Algoritma : **Stochastic Hill Climbing**

Duration : **16 ms**

Jumlah Iterasi : **50001**

Objective Function : **32**

c. Percobaan 3

State Awal :

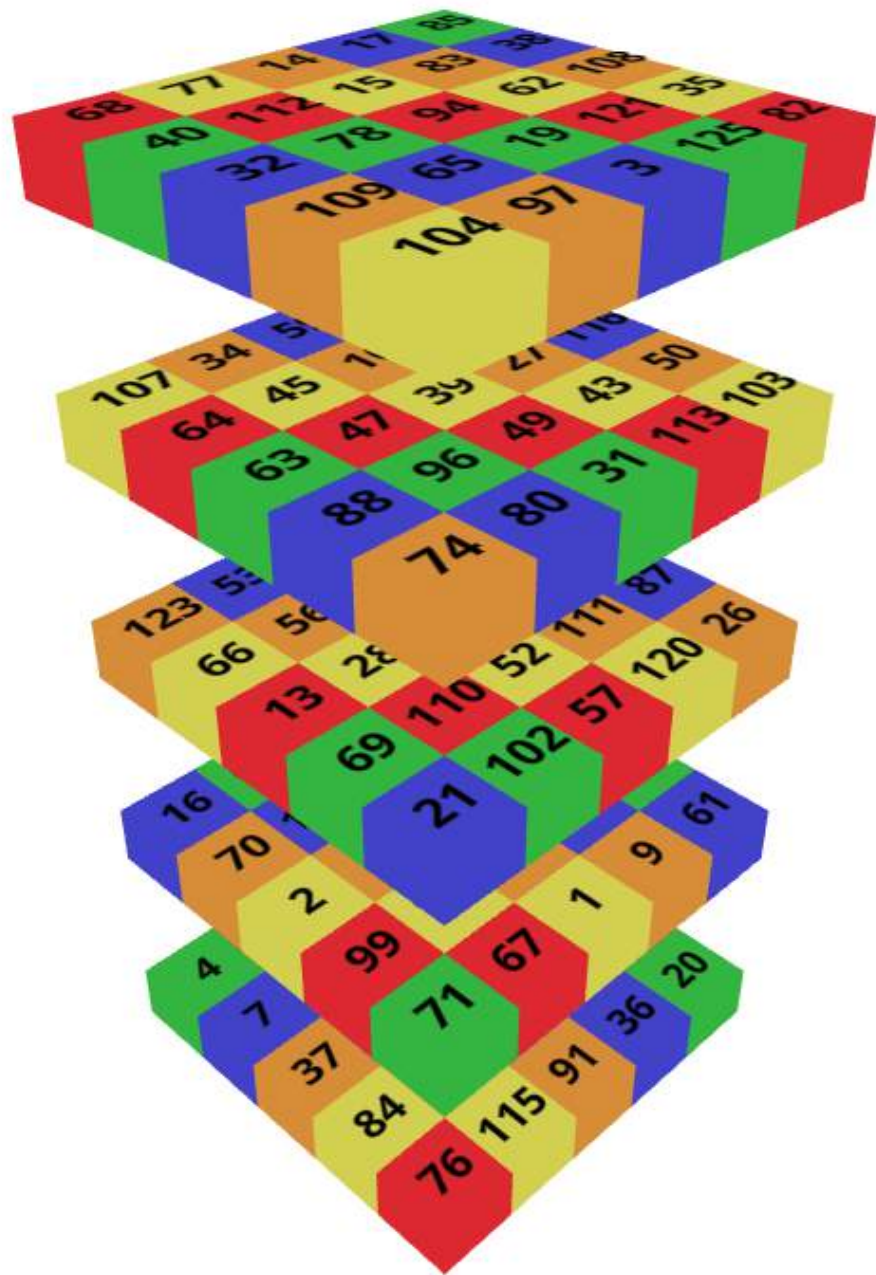
68 77 14 17 85
40 112 15 83 38
32 78 94 62 108
109 65 19 121 35
104 97 3 125 82

107 34 59 118 24
64 45 10 51 89
63 47 39 27 116
88 96 49 43 50
74 80 31 113 103

123 53 33 98 92
66 56 54 81 5
13 28 41 42 72
69 110 52 111 87
21 102 57 120 26

16 95 105 11 117
70 119 75 22 100
2 18 60 23 79
99 73 124 29 8
71 67 1 9 61

4 25 122 106 93
7 44 90 12 86
37 30 55 101 6
84 48 114 46 58
76 115 91 36 20



State Akhir :

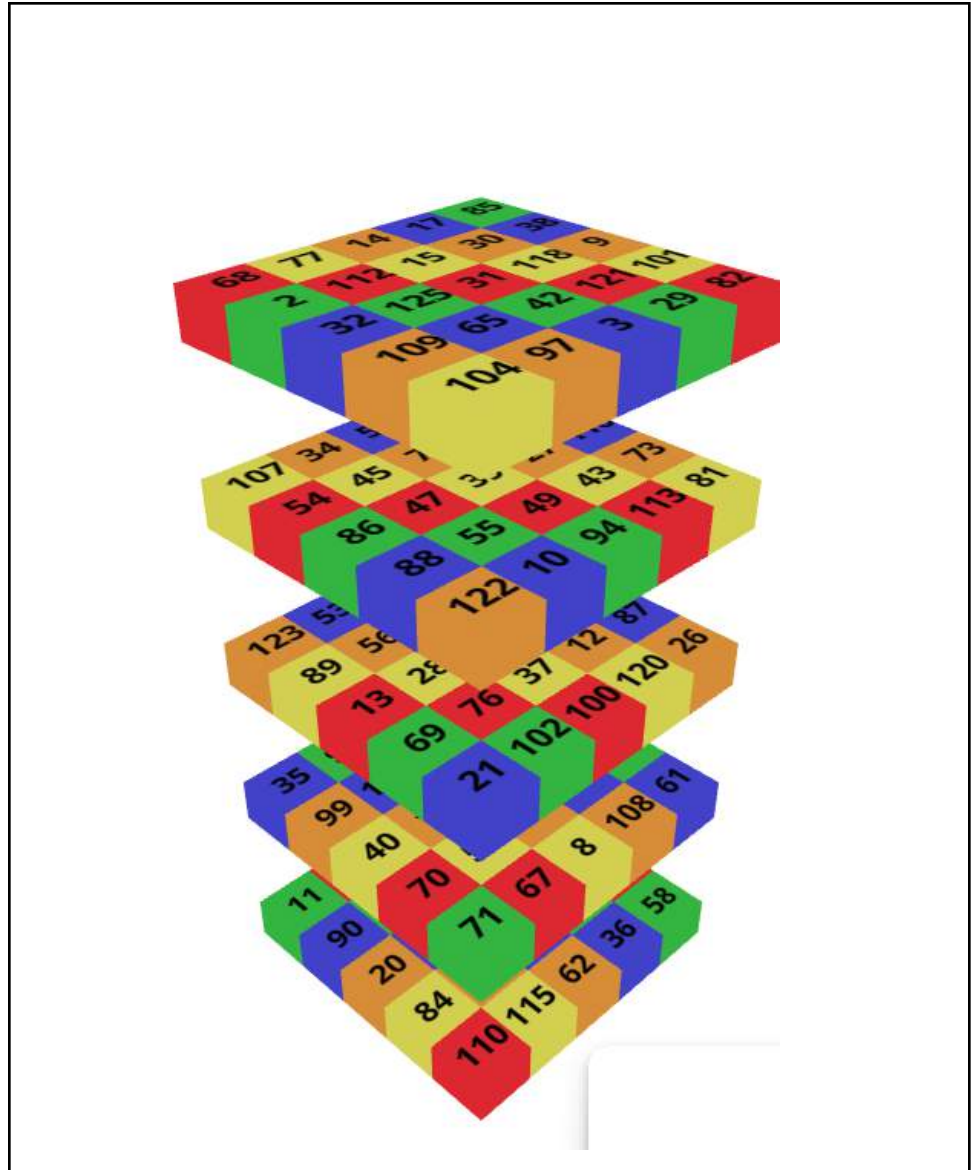
68 77 14 17 85
2 112 15 30 38
32 125 31 118 9
109 65 42 121 101
104 97 3 29 82

107 34 59 91 24
54 45 74 75 66
86 47 39 27 116
88 55 49 43 73
122 10 94 113 81

123 53 33 98 92
89 56 64 96 5
13 28 41 19 105
69 76 37 12 87
21 102 100 120 26

35 95 72 4 117
99 119 51 22 57
40 18 60 78 79
70 50 124 103 1
71 67 8 108 61

11 25 80 106 93
90 44 7 111 63
20 83 52 16 6
84 48 114 46 23
110 115 62 36 58



Keterangan :

Algoritma : **Stochastic Hill Climbing**

Duration : **373 ms**

Jumlah Iterasi : **1000001**

Objective Function : **36**

2.4.5 Simulated Annealing

a. Percobaan 1

State Awal :

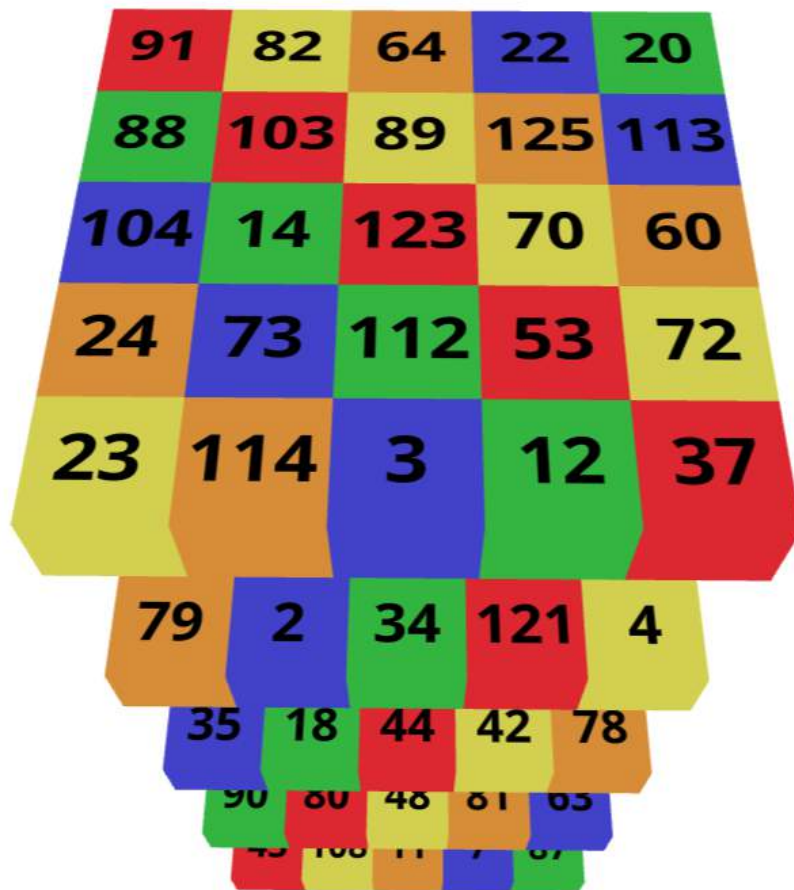
```
91 82 64 22 20
88 103 89 125 113
104 14 123 70 60
24 73 112 53 72
23 114 3 12 37

8 36 52 77 15
66 6 27 92 33
124 38 97 94 99
67 55 68 10 31
79 2 34 121 4

29 119 111 100 117
95 28 106 84 19
62 9 51 17 96
98 46 54 45 57
35 18 44 42 78

86 61 65 1 76
110 30 41 47 71
69 21 13 85 102
58 5 74 115 50
90 80 48 81 63

59 32 49 93 122
105 16 118 101 83
109 116 120 56 40
25 107 39 75 26
43 108 11 7 87
```



State Akhir :

```

42 56 13 120 84
25 83 121 27 59
92 24 90 51 58
116 124 94 29 67
40 80 96 88 6

32 115 86 91 1
69 110 47 8 81
26 18 98 71 102

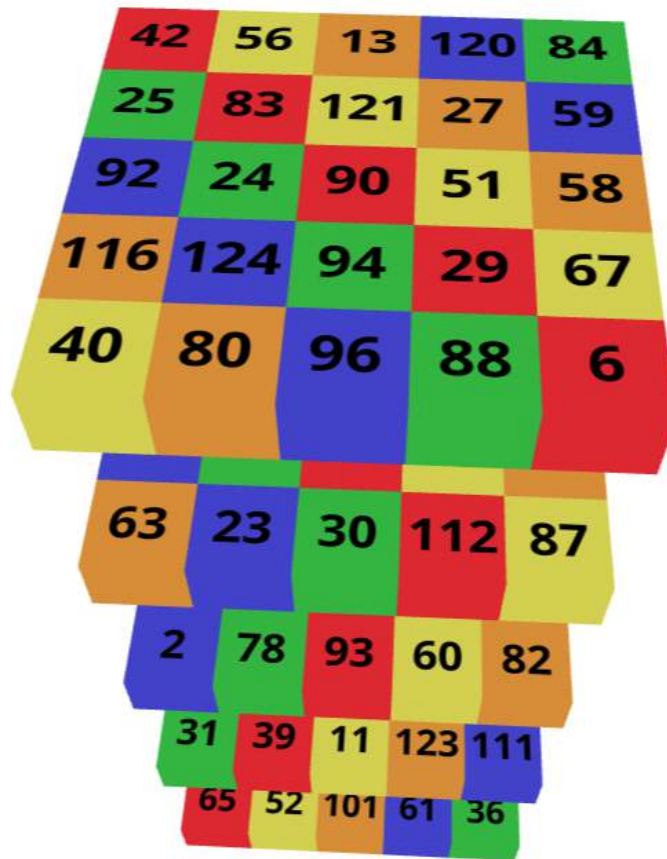
```

125 21 54 33 118
63 23 30 112 87

4 73 108 49 100
109 117 20 66 3
62 85 48 76 44
89 99 46 64 17
2 78 93 60 82

14 113 41 5 55
35 10 95 103 72
38 122 114 34 7
77 12 106 50 70
31 39 11 123 111

53 119 22 15 75
107 37 105 9 57
16 79 19 97 104
74 28 68 45 43
65 52 101 61 36



Keterangan :

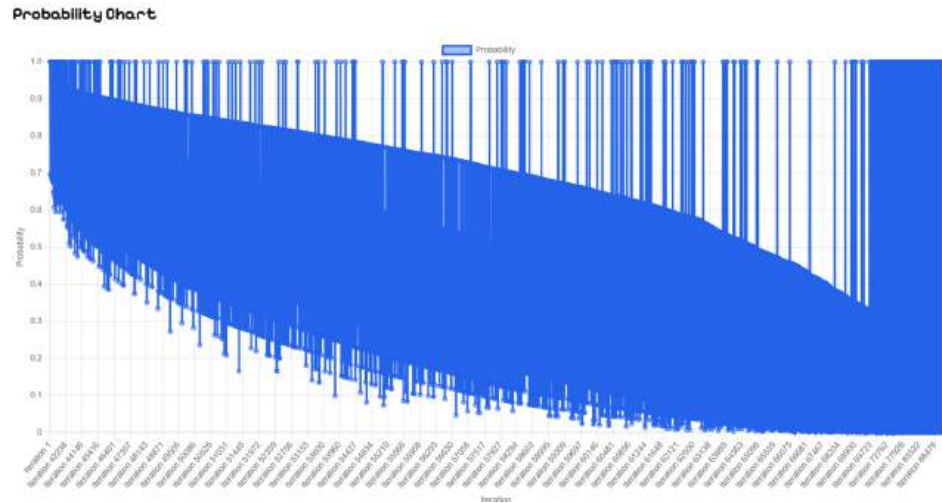
Algoritma : **Simulated Annealing**

Duration : **114ms**

Frekuensi Stuck : **91483**

Jumlah Iterasi : **91534**

Objective Function : **50**



b. Percobaan 2

State Awal :

```

71 57 7 43 11
68 39 87 5 124
65 81 12 92 85
110 52 13 34 67
6 82 40 122 30

35 38 58 89 88
60 48 9 80 1
114 36 93 108 100
46 15 26 20 32
17 51 84 42 64

111 117 86 90 16
63 120 77 94 27
78 103 4 66 125
55 21 119 69 115
95 47 121 83 96

74 116 79 44 41
101 29 73 91 107
10 2 123 37 112
113 14 54 22 53
24 28 76 23 98

49 109 72 25 105
8 31 59 70 45
102 18 106 61 3
75 62 99 104 33

```

118 97 19 56 50



State Akhir :

105 116 10 6 78
120 28 115 104 12
9 92 51 101 62
42 72 27 71 103
49 20 112 33 60

40 106 13 108 48
83 55 79 43 25
19 36 37 56 80
93 65 123 107 86
122 53 63 1 76

111 4 89 90 21
85 77 22 67 64
3 82 46 26 102
29 94 59 8 125
87 58 30 124 16

14 35 117 31 118
32 61 110 95 17
38 81 47 114 109
69 39 84 57 66

119 99 74 18 5

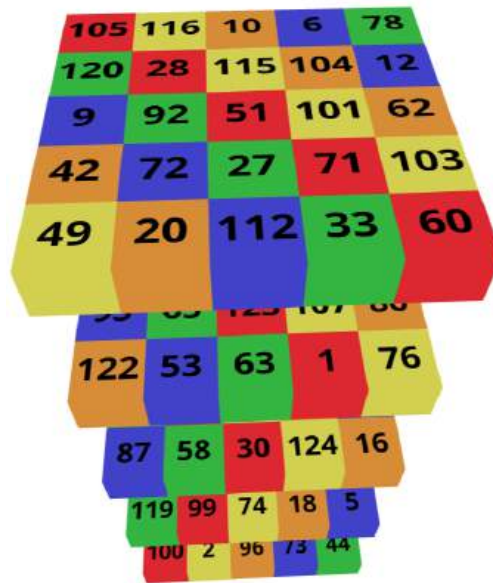
41 113 34 54 50

97 68 52 15 23

70 11 45 98 91

7 121 88 75 24

100 2 96 73 44



Keterangan :

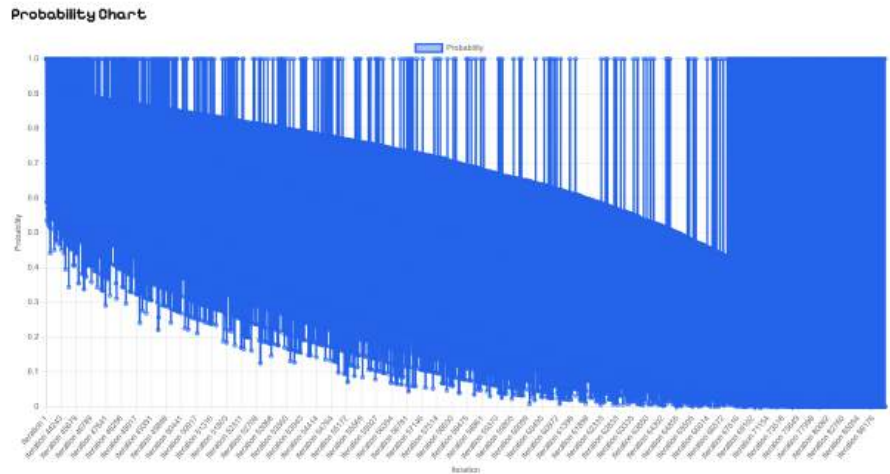
Algoritma : **Simulated Annealing**

Duration : **177ms**

Frekuensi Stuck : **91488**

Jumlah Iterasi : **91534**

Objective Function : **46**



c. Percobaan 3

State Awal :

```
19 30 45 38 31
104 50 15 99 101
9 89 37 84 49
77 82 78 41 52
123 29 92 86 111
```

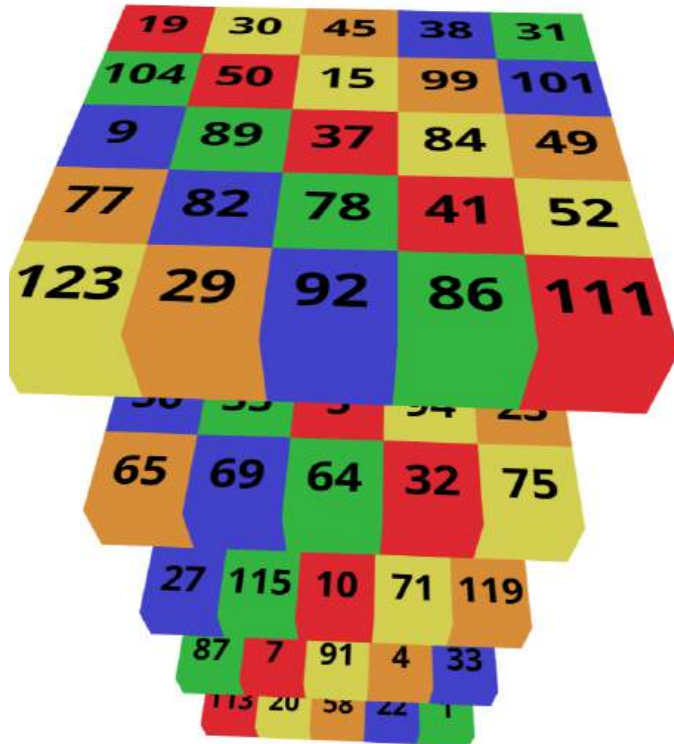
```
72 88 97 68 100
106 40 13 60 117
103 90 67 59 95
56 35 5 94 23
65 69 64 32 75
```

```
83 114 116 57 54
8 105 73 93 25
112 11 122 3 98
79 80 28 66 63
27 115 10 71 119
```

```
85 44 53 118 2
108 102 36 18 16
61 46 43 107 76
48 42 70 14 120
87 7 91 4 33
```

```
81 47 26 21 124
24 51 55 12 17
121 34 39 62 125
```

96 6 109 74 110
113 20 58 22 1



State Akhir :

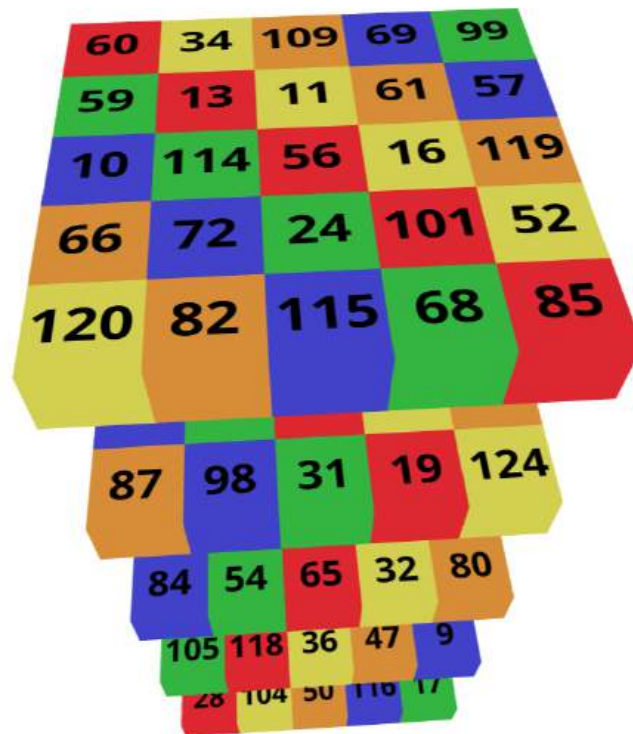
60 34 109 69 99
59 13 11 61 57
10 114 56 16 119
66 72 24 101 52
120 82 115 68 85

40 21 71 2 23
67 26 110 91 39
79 64 8 86 78
12 106 95 117 51
87 98 31 19 124

100 38 76 113 88
 70 49 92 123 4
 58 77 45 6 121
 3 97 37 41 22
 84 54 65 32 80

18 75 90 112 20
 62 5 7 30 107
 125 43 29 93 25
 102 74 1 33 111
 105 118 36 47 9

63 89 53 96 14
 122 27 83 46 108
 94 55 48 15 103
 35 44 81 42 73
 28 104 50 116 17



Keterangan :

Algoritma : **Simulated Annealing**

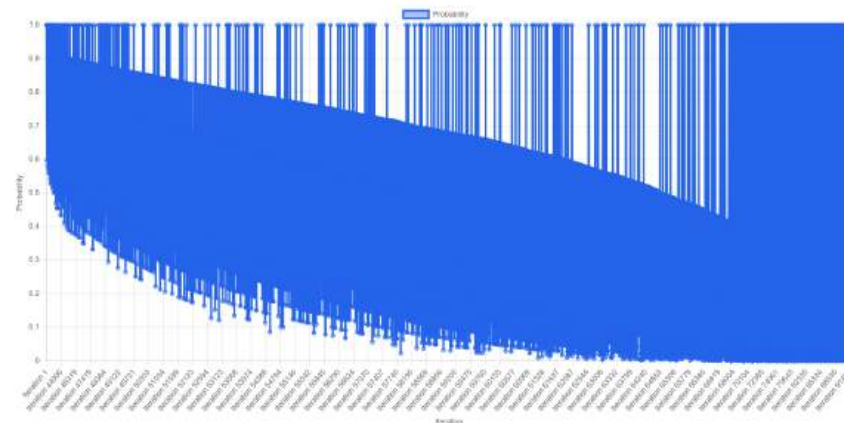
Duration : **89ms**

Frekuensi Stuck : **91485**

Jumlah Iterasi : **91534**

Objective Function : **48**

Probability Chart



2.4.6 Genetic Algorithm

a. Percobaan 1

State Awal :

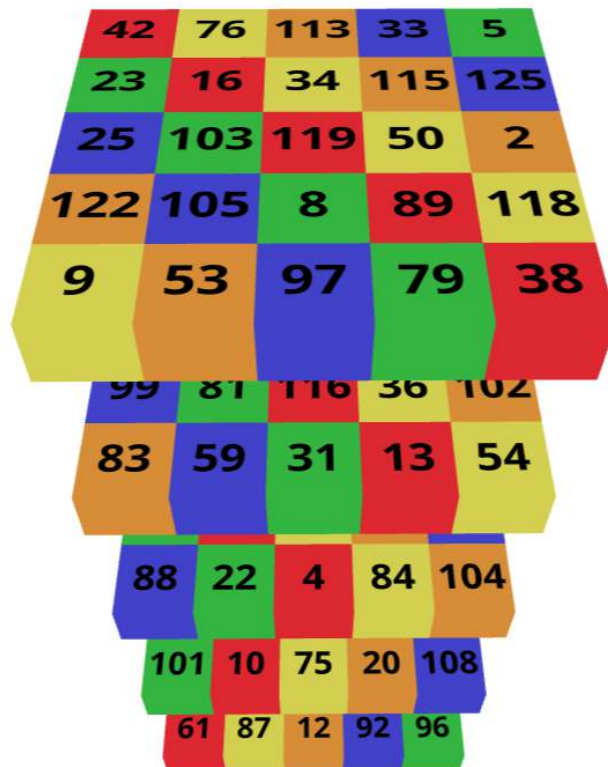
```
42 76 113 33 5
23 16 34 115 125
25 103 119 50 2
122 105 8 89 118
9 53 97 79 38
```

```
98 56 39 47 110
19 67 107 15 21
106 24 93 85 28
99 81 116 36 102
83 59 31 13 54
```

```
52 27 64 32 26
123 124 80 35 41
65 91 46 95 68
112 100 49 30 11
88 22 4 84 104
```

43 114 60 82 51
 71 7 74 44 70
 3 86 66 40 117
 6 18 37 69 120
 101 10 75 20 108

73 45 17 109 63
 121 111 48 90 58
 14 94 29 62 78
 57 77 1 55 72
 61 87 12 92 96



State Akhir :

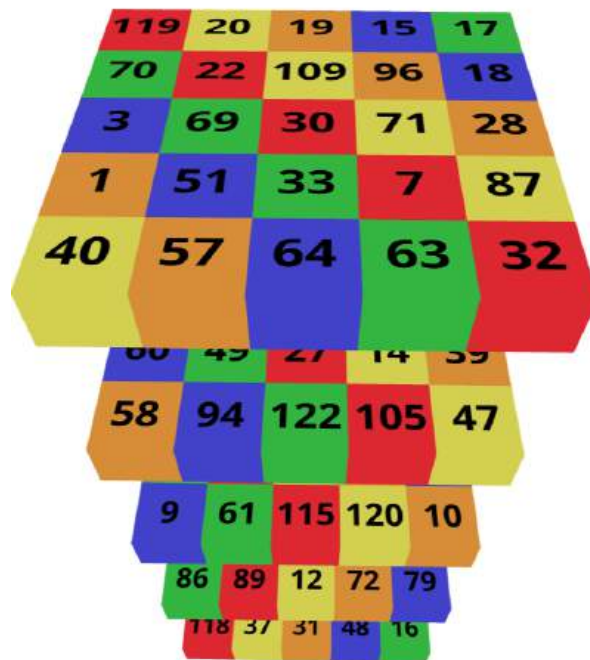
119 20 19 15 17
70 22 109 96 18
3 69 30 71 28
1 51 33 7 87
40 57 64 63 32

99 23 104 83 6
73 4 21 92 117
56 93 65 113 106
60 49 27 14 39
58 94 122 105 47

123 5 2 78 107
100 125 59 114 85
38 124 41 88 24
91 34 81 82 66
9 61 115 120 10

111 54 110 25 108
53 84 97 102 50
11 35 101 74 45
13 36 121 43 95
86 89 12 72 79

52 75 67 29 77
62 76 46 55 80
90 8 68 42 112
44 26 103 116 98
118 37 31 48 16



Keterangan :

Algoritma : **Genetic Algorithm**

Best Value : **11**

Duration : **3.016662s**

Jumlah Generasi : **500**

Jumlah Populasi Awal : **500**

b. Percobaan 2

State Awal :

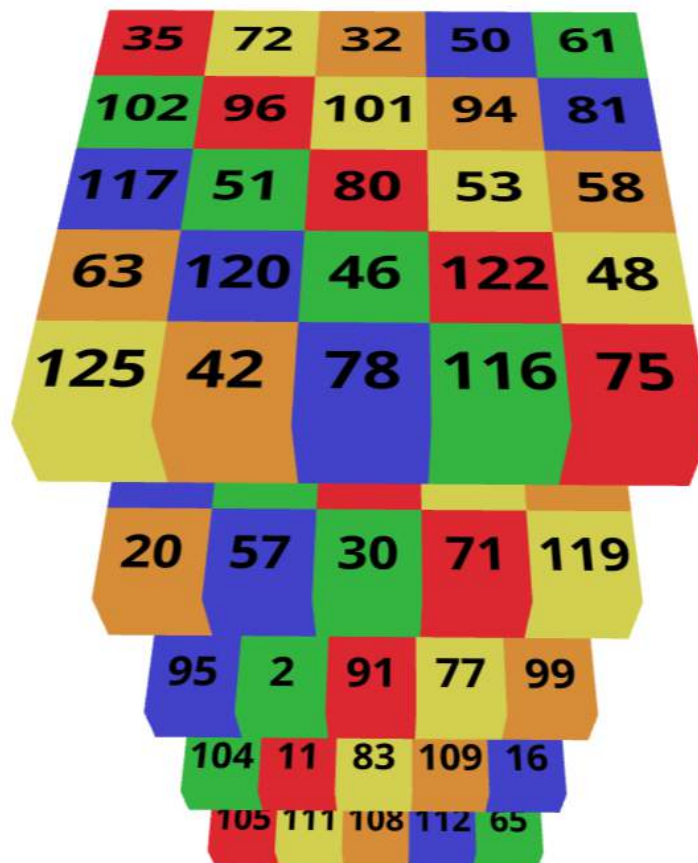
```
35 72 32 50 61
102 96 101 94 81
117 51 80 53 58
63 120 46 122 48
125 42 78 116 75

103 14 45 113 13
100 28 29 34 33
68 59 106 10 21
1 54 39 76 85
20 57 30 71 119

5 31 40 6 43
69 47 93 124 41
55 123 62 26 12
67 3 82 84 87
95 2 91 77 99

8 107 88 114 4
66 44 121 49 22
90 73 23 37 98
56 79 92 118 27
104 11 83 109 16

25 52 70 19 60
38 9 7 86 18
97 17 89 24 110
15 115 74 36 64
105 111 108 112 65
```



State Akhir :

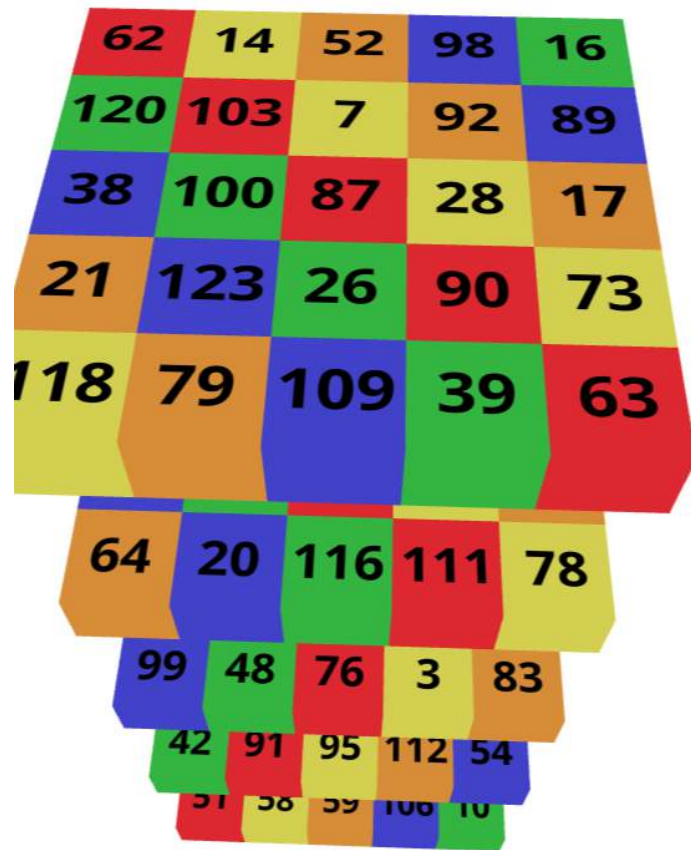
62 14 52 98 16
120 103 7 92 89
38 100 87 28 17
21 123 26 90 73
118 79 109 39 63

31 1 15 84 34
71 45 125 94 57
35 74 105 81 115
37 97 24 110 47
64 20 116 111 78

8 65 77 117 108
85 124 88 96 68
30 2 70 9 19
93 66 49 18 32
99 48 76 3 83

122 11 23 40 12
13 82 22 104 55
33 5 86 56 41
50 61 69 6 114
42 91 95 112 54

27 29 102 67 119
36 75 80 121 46
53 101 43 72 44
4 25 60 113 107
51 58 59 106 10



Keterangan :

Algoritma : **Genetic Algorithm**

Best Value : 3

Duration : **1.1904ms**

Jumlah Generasi : **10**

Jumlah Populasi Awal : **10**

c. Percobaan 3

State Awal :

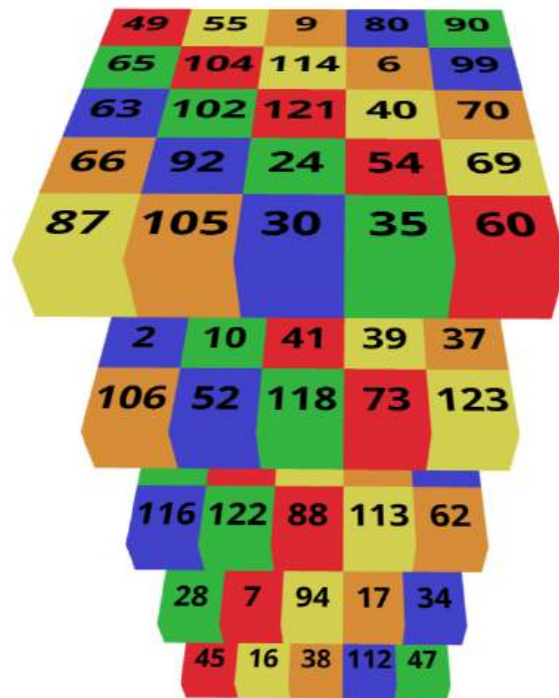
```
49 55 9 80 90
65 104 114 6 99
63 102 121 40 70
66 92 24 54 69
87 105 30 35 60

64 21 57 67 15
32 96 85 1 98
100 26 93 22 72
2 10 41 39 37
106 52 118 73 123

95 115 84 23 42
74 59 58 119 5
103 77 110 51 29
78 27 8 25 83
116 122 88 113 62

101 44 31 86 76
46 14 43 61 75
19 91 13 125 53
107 111 81 48 82
28 7 94 17 34

36 109 50 12 108
18 124 117 20 120
33 56 11 97 89
4 3 79 71 68
45 16 38 112 47
```



State Akhir :

```

58 99 87 24 20
1 106 51 10 68
81 64 88 21 3
97 66 29 18 107
92 23 34 11 95

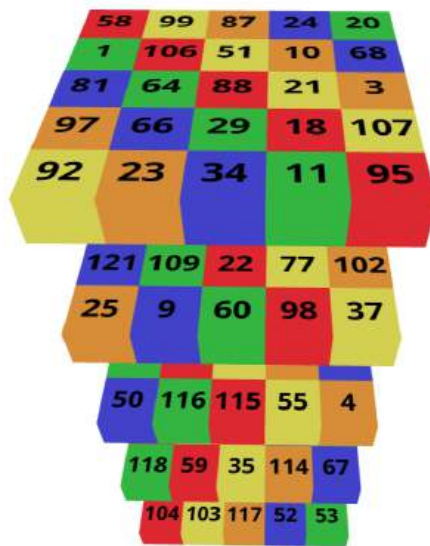
63 42 96 28 6
39 108 38 49 65
123 13 27 56 43
121 109 22 77 102
25 9 60 98 37

61 113 54 26 86
79 19 90 120 7
78 8 119 40 70
80 74 100 112 12
50 116 115 55 4

```

33 122 46 14 44
41 101 110 32 91
57 71 48 94 45
36 73 5 72 125
118 59 35 114 67

83 82 30 31 69
75 62 85 105 84
124 76 89 16 17
93 15 47 111 2
104 103 117 52 53



Keterangan :

Algoritma : **Genetic Algorithm**

Best Value : **10**

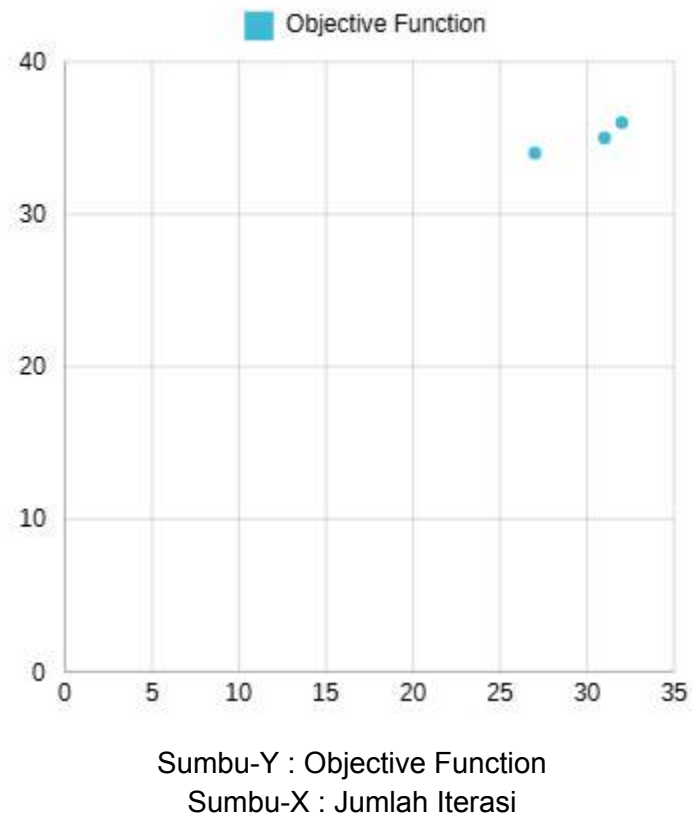
Duration : **632.5024ms**

Jumlah Generasi : **200**

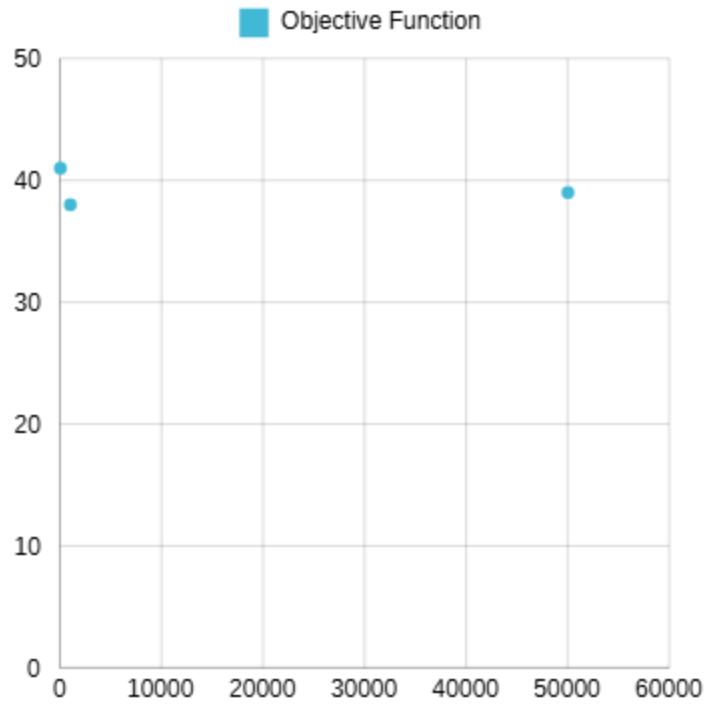
Jumlah Populasi Awal : **200**

2.4.7 Plot Objective Function

- a. Plot Objective Function pada Steepest Ascent Hill Climbing Algorithm terhadap banyak iterasi

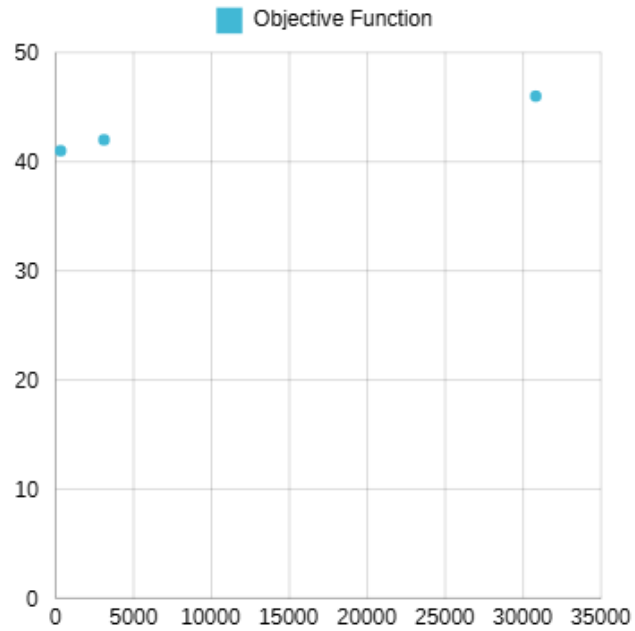


- b. Plot Objective Function pada Hill Climbing with Sideways Move Algorithm terhadap banyak iterasi



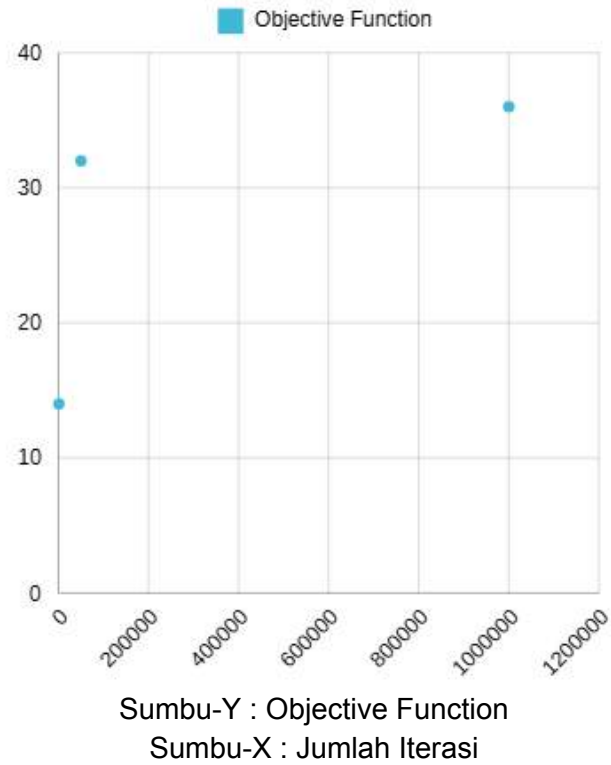
Sumbu-Y : Objective Function
Sumbu-X : Jumlah Iterasi

- c. Plot Objective Function pada Random Restart Hill Climbing Algorithm terhadap banyak iterasi

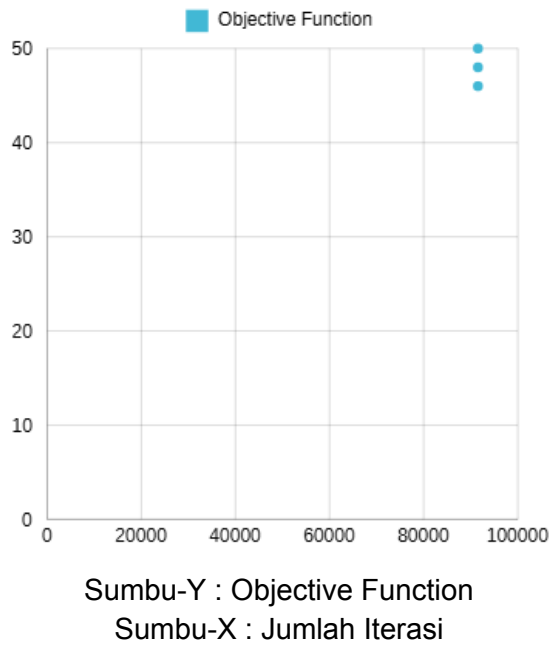


Sumbu-Y : Objective Function
Sumbu-X : Jumlah Iterasi

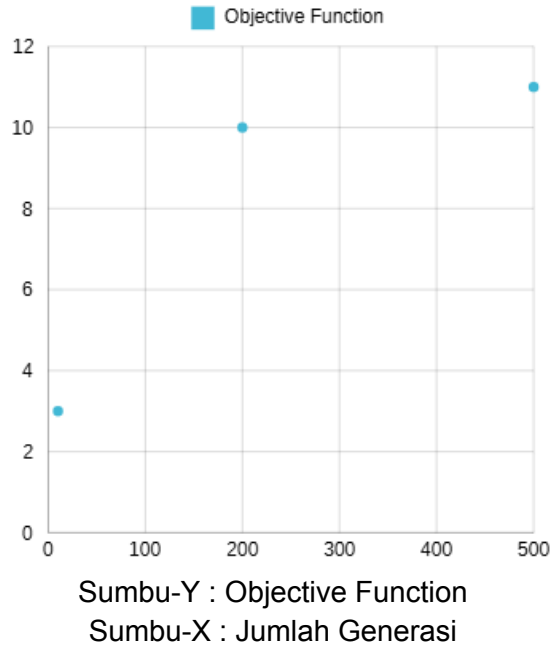
- d. Plot Objective Function pada Stochastic Hill Climbing Algorithm terhadap banyak iterasi



- e. Plot Objective Function pada Simulated Annealing terhadap banyak iterasi



f. Plot Objective Function pada Genetic Algorithm terhadap banyak iterasi



2.4.8 Analisis

Berikut adalah tabel hasil eksperimen dengan perbandingan eksekusi waktu dan nilai objective

No	Algoritma		Keterangan	Waktu Eksekusi	Rata-Rata	Nilai Objective Function		Rata-Rata
1	Steepest Ascent Hill-climbing	Percobaan 1	-	60.756ms	63.3695ms	Percobaan 1	34	35
		Percobaan 2	-	62.5736ms		Percobaan 2	35	
		Percobaan 3	-	66.7788ms		Percobaan 3	36	
2	Hill-climbing with Sideways Move	Percobaan 1	Max iterasi = 1000	2.1268242s	34.8953s	Percobaan 1	38	39.3
		Percobaan 2	Max iterasi = 50000	1m42.47s		Percobaan 2	39	
		Percobaan 3	Max iterasi = 10	89.2048ms		Percobaan 3	41	

3	Random Restart Hill-climbing	Percobaan 1	Max Restart = 1000	1m6.203s	24.245s	Percobaan 1	46	43
		Percobaan 2	Max Restart = 100	6.3582937 s		Percobaan 2	42	
		Percobaan 3	Max Restart = 10	713.5364 ms		Percobaan 3	41	
4	Stochastic Hill-climbing	Percobaan 1	Max iterasi = 1001	0ms	129.67ms	Percobaan 1	14	27.3
		Percobaan 2	Max iterasi = 50001	16ms		Percobaan 2	32	
		Percobaan 3	Max iterasi = 1000001	373ms		Percobaan 3	36	
5	Simulated Annealing	Percobaan 1	-	114ms	106.67ms	Percobaan 1	50	48
		Percobaan 2	-	117ms		Percobaan 2	46	
		Percobaan 3	-	89ms		Percobaan 3	48	
6	Genetic Algorithm	Percobaan 1	Jumlah Generasi = 500 Jumlah Populasi Awal = 500	3.016662s	1216.785 ms	Percobaan 1	11	8
		Percobaan 2	Jumlah Generasi = 10 Jumlah Populasi Awal = 10	1.1904ms		Percobaan 2	3	
		Percobaan 3	Jumlah Generasi = 200 Jumlah Populasi Awal = 200	632.5024 ms		Percobaan 3	10	

Setelah melakukan tiga kali percobaan pada algoritma Steepest Ascent Hill-Climbing, hasil yang diperoleh masih jauh dari global optimum. Hal ini terjadi karena initial state algoritma ditentukan secara acak, dan probabilitas untuk mendapatkan initial state dengan nilai objective function yang baik sangat kecil. Selain itu, jarak antara setiap state ke global optimum sangatlah jauh sehingga diperlukan sangat banyak langkah tanpa henti untuk bisa mendapatkan hasil berupa global optimum. Algoritma ini mampu memberikan hasil yang cukup optimal (tidak terlalu buruk dan cukup baik) jika dibandingkan dengan algoritma lainnya. Waktu eksekusi Steepest Ascent Hill-Climbing dapat dikatakan relatif singkat, karena sering kali nilai neighbor memiliki nilai yang sama atau lebih kecil daripada current state, sehingga algoritma berhenti lebih cepat. Hasil akhir algoritma ini konsisten di sekitar angka 35 dari total nilai optimal 109. Konsistensi ini terjaga karena algoritma tidak memiliki parameter yang membatasi jumlah iterasi atau lamanya proses berjalan.

Setelah melakukan 3 kali percobaan terhadap algoritma Hill-Climbing with Sideways Move, hasil pemrosesan dari algoritma tersebut masih sangat jauh dari global optima. Hal tersebut dapat terjadi karena initial state pada algoritma ini ditentukan secara random, dan probabilitas untuk mendapatkan initial state dengan nilai objective function yang bagus sangat kecil. Selain itu, jarak antara setiap state ke global optimum sangatlah jauh, sehingga diperlukan sangat banyak langkah tanpa henti untuk bisa mendapatkan hasil berupa global optimum. Algoritma ini mendapatkan hasil pencarian yang cukup optimal (tidak terlalu buruk dan cukup baik) jika dibandingkan dengan algoritma lainnya dan juga lebih baik daripada algoritma Steepest Ascent Hill-Climbing karena algoritma ini mampu mentoleransi flat area, sehingga kemungkinan untuk algoritma ini berhenti karena local maksimum di flat area bisa dikurangi. Waktu eksekusi algoritma ini bisa dibilang sangat singkat, karena kasus dimana neighbor value lebih kecil daripada current state muncul sangat sering. Namun, algoritma ini masih lebih lama jika dibandingkan dengan Steepest Ascent Hill-Climbing karena algoritma ini masih menerima atau mentoleransi adanya flat area. Hasil akhir dari algoritma ini sangat konsisten disekitar angka 39 dari total 109. Konsistensi algoritma ini kurang terjaga karena adanya parameter max-iterasi. Parameter ini akan mempengaruhi waktu eksekusi algoritma pada kasus yang berbeda (dengan max-iterasi yang berbeda pula), terutama saat algoritma terjebak di flat area untuk waktu atau jumlah iterasi yang cukup besar. Nilai parameter max-iterasi yang kecil mampu membatasi eksekusi dari algoritma ini agar bisa selesai lebih cepat. Namun, algoritma ini memiliki trade-off yaitu kemungkinan algoritma ini

untuk bisa mencapai nilai yang lebih baik akan mengecil. Hal sebaliknya terjadi jika algoritma ini memiliki nilai parameter yang besar, maka algoritma ini akan memakan waktu yang lebih banyak namun akan memiliki kemungkinan lebih besar untuk mencapai nilai objective function yang lebih baik.

Setelah melakukan 3 kali percobaan terhadap algoritma Random Restart Hill-Climbing, hasil pemrosesan dari algoritma tersebut masih cukup jauh terhadap global optima. Hal tersebut dapat terjadi karena initial state pada algoritma ini ditentukan secara random, dan probabilitas untuk mendapatkan initial state dengan nilai objective function yang bagus sangat kecil. Selain itu, jarak antara setiap state ke global optimum sangatlah jauh, sehingga diperlukan sangat banyak langkah tanpa henti untuk bisa mendapatkan hasil berupa global optimum. Algoritma ini mendapatkan hasil pencarian yang lebih optimal jika dibandingkan dengan algoritma lainnya. Waktu eksekusi algoritma ini bisa dibilang singkat namun ada beberapa kasus yang membuat waktu eksekusi algoritma ini menjadi cukup lama. Hal tersebut dapat terjadi karena algoritma ini kurang lebih sama seperti mengulang-ulang algoritma Steepest Ascent Hill-Climbing. Hasil akhir dari algoritma ini cukup konsisten disekitar angka 43 dari total 109. Konsistensi algoritma ini tergolong cukup buruk karena sangat dipengaruhi oleh jumlah restart yang diperbolehkan, jika jumlah restart yang diperbolehkan sangat banyak maka nilai yang dihasilkan pun bisa menjadi sangat baik dengan trade-off waktu eksekusi akan menjadi sangat lama. Namun, jika jumlah restart yang diperbolehkan sangat sedikit, maka waktu yang dibutuhkan akan sedikit namun hasil yang didapat belum tentu baik.

Setelah melakukan 3 kali percobaan terhadap algoritma Stochastic Hill-Climbing, hasil pemrosesan dari algoritma tersebut masih sangat jauh terhadap global optima. Hal tersebut dapat terjadi karena initial state pada algoritma ini ditentukan secara random, dan probabilitas untuk mendapatkan initial state dengan nilai objective function yang bagus sangat kecil. Selain itu, jarak antara setiap state ke global optimum sangatlah jauh, sehingga diperlukan sangat banyak langkah tanpa henti untuk bisa mendapatkan hasil berupa global optimum dan juga hasil dari algoritma ini sangat bergantung pada parameter max-iterasi. Algoritma ini mendapatkan hasil pencarian yang kurang optimal jika dibandingkan dengan algoritma lainnya. Waktu eksekusi algoritma ini tidak dapat ditentukan dengan pasti karena algoritma ini sangat bergantung pada nilai parameter max-iterasi, jika max-iterasi kecil maka waktu eksekusi akan sangat cepat dan sebaliknya. Algoritma ini menghasilkan hasil akhir yang sangat tidak

konsisten. Ketidakkonsistenan ini terjadi karena algoritma sangat bergantung pada nilai parameter max-iterasi. Semakin besar nilai max-iterasi, hasil yang diperoleh cenderung lebih baik, namun waktu eksekusinya lebih lama. Sebaliknya, jika max-iterasi bernilai kecil, eksekusi menjadi lebih cepat, tetapi hasil yang didapat cenderung kurang optimal.

Setelah melakukan tiga kali percobaan pada algoritma Simulated Annealing, hasil yang diperoleh masih cukup jauh dari global optimum namun masih lebih baik dibanding algoritma lainnya. Hal ini dapat terjadi karena algoritma ini memiliki metode untuk mengatasi kekurangan algoritma lainnya, seperti untuk mengurangi peluang algoritma terjebak pada lokal optimum dengan memadukan algoritma Stochastic Hill-Climbing dengan free walk algorithm. Algoritma ini memberikan hasil terbaik jika dibandingkan algoritma lainnya. Waktu eksekusi algoritma ini cukup cepat jika dibandingkan dengan algoritma lainnya. Hasil akhir dari algoritma ini ada di angka 48 dari total 109. Hasil ini tergolong sangat bagus, karena algoritma ini mampu recover dari terjebak di lokal optimum. Algoritma ini memiliki konsistensi yang cukup tinggi, hal ini dapat terjadi karena temperatur awal dan peluruhan yang konsisten. Dengan pengaturan suhu yang konsisten, hasil akhir algoritma tidak terlalu dipengaruhi oleh keacakan dalam pemilihan *state*, sehingga memungkinkan algoritma untuk lebih andal dalam mencapai hasil yang mendekati optimal.

Terakhir, setelah melakukan tiga kali percobaan pada algoritma Genetic Algorithm, hasil yang diperoleh sangat jauh dari global optimum. Hal ini terjadi karena initial populasi algoritma ditentukan secara acak dan seluruh proses dalam algoritma ini dilakukan secara random tanpa adanya heuristik apapun. Algoritma ini memberikan hasil terburuk jika dibandingkan dengan algoritma lainnya. Waktu eksekusi algoritma ini sangat tergantung kepada parameter jumlah generasi dan jumlah populasi. Semakin besar jumlah generasi dan jumlah populasi, maka akan semakin lama juga waktu eksekusinya dan begitupun sebaliknya. Sehingga, waktu eksekusi algoritma ini jika dibandingkan dengan algoritma lainnya akan sangat buruk. Hasil akhir algoritma ini ada di sekitar angka 8 dari total nilai optimal 109. Ketidakkonsistenan algoritma ini sangat dipengaruhi oleh nilai parameter jumlah generasi dan jumlah populasi. Kedua parameter tersebut paling berdampak pada waktu eksekusi algoritma dan tidak terlalu mempengaruhi nilai objective function terbaik yang dihasilkan.

BAB 3 KESIMPULAN DAN SARAN

3.1 Kesimpulan

Dari berbagai algoritma local search yang diuji untuk menyelesaikan masalah Magic Cube, Simulated Annealing terbukti sebagai algoritma paling optimal dan andal. Algoritma ini mampu mencapai hasil yang mendekati global optimum dan memiliki kemampuan untuk menghindari stuck di local optimum melalui mekanisme probabilitas yang memungkinkan penerimaan solusi yang lebih buruk. Konsistensi hasil Simulated Annealing cukup tinggi berkat pengaturan suhu awal dan laju peluruhan yang stabil,

yang mengurangi ketergantungan pada keacakan initial state. Random Restart Hill-Climbing juga efektif dalam menghindari stuck, meskipun hasil akhirnya masih lebih rendah dari Simulated Annealing.

Di sisi lain, Genetic Algorithm menunjukkan hasil yang paling tidak konsisten, dengan hasil akhir sering jauh dari optimal meski memakan waktu eksekusi yang panjang, karena ketergantungannya pada parameter jumlah generasi dan populasi. Stochastic Hill-Climbing juga cenderung kurang konsisten karena sangat bergantung pada max-iterasi. Secara keseluruhan, Simulated Annealing adalah pilihan terbaik untuk mencapai hasil optimal dan konsisten dalam waktu yang efisien, sementara Genetic Algorithm adalah yang terburuk dalam hal hasil dan konsistensi.

3.1 Saran

Untuk meningkatkan pengalaman pengguna dalam platform penyelesaian Magic Cube, berikut beberapa saran agar platform lebih mudah dipahami dan menarik.

1. Visualisasi Pembentukan Populasi. Penambahan visualisasi dalam proses pembangkitan populasi di setiap generasi dalam algoritma genetika akan membantu pengguna memahami langkah-langkah yang ditempuh algoritma menuju solusi optimal. Tampilan ini akan memberikan gambaran lebih jelas tentang perubahan populasi di tiap generasi.
2. Visualisasi Interaktif dengan Video Player. Menyediakan visualisasi interaktif menggunakan video player yang menampilkan perubahan kondisi kubus pada tiap iterasi atau langkah algoritma, bahkan setiap detik. Fitur ini memungkinkan pengguna melihat perkembangan solusi secara bertahap menuju hasil akhir.
3. Tampilan Kubus yang Lebih Realistis. Meningkatkan kualitas visual kubus dengan penggunaan PBR (Physically Based Rendering) material pada shader akan menambah kesan realistis dan estetika, sehingga tampilan kubus terlihat lebih hidup dan menarik.

Dengan saran-saran ini, diharapkan platform penyelesaian Magic Cube dapat menjadi lebih interaktif, informatif, dan ramah bagi pengguna.

Pembagian Tugas

Tugas	Anggota
Visualisasi (Front-End)	13522157 - Muhammad Davis Adhipramana
Objective Function	13522164 - Valentino Chryslie Triadi
Algoritma Steepest Ascent Hill-Climbing	13522134 - Shabrina Maharani
Algoritma Hill-Climbing with Sideways Move	13522157 - Muhammad Davis Adhipramana
Algoritma Random Restart Hill-Climbing	13522134 - Shabrina Maharani
Algoritma Stochastic Hill-Climbing	13522153 - Muhammad Fauzan Azhim
Algoritma Simulated Annealing	13522153 - Muhammad Fauzan Azhim

Genetic Algorithm	13522164 - Valentino Chryslie Triadi
-------------------	--------------------------------------

Referensi

Russell, S., & Norvig, P. (2021). *Artificial intelligence: A modern approach (Global ed.)*. Pearson Higher Ed.

PPT Pembelajaran Mata Kuliah Inteligensi Artifisial Tahun 2024

Redi, A. A. N. P., Maula, F. R., Kumari, F., Syayevenda, N. U., Ruswandi, N., Khasanah, A. U., & Kurniawan, A. C. (2020). Simulated annealing algorithm for solving the capacitated vehicle routing problem: A case study of pharmaceutical distribution. *Jurnal Sistem dan Manajemen Industri*, 4(1), 41-49.