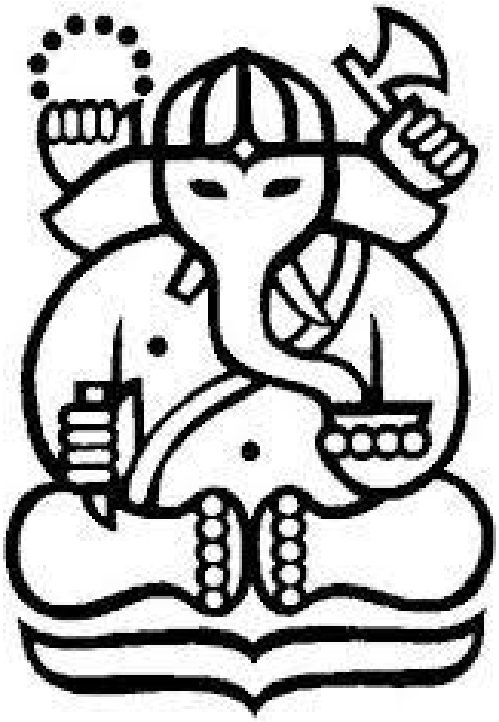
APLIKASI LOCAL SEARCH DALAM PENYELESAIAN PERMASALAHAN MAGIC CUBE



IF3170 - Inteligensi Artifisial

Oleh Kelompok 43:

1. Ignatius Jhon Hezkiel Chan (13522029)
2. Daniel Mulia Putra Manurung (13522043)
3. Benjamin Sihombing (13522054)
4. Marvin Scifo Y. Hutahaean (13522110)

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

2024

Daftar Isi

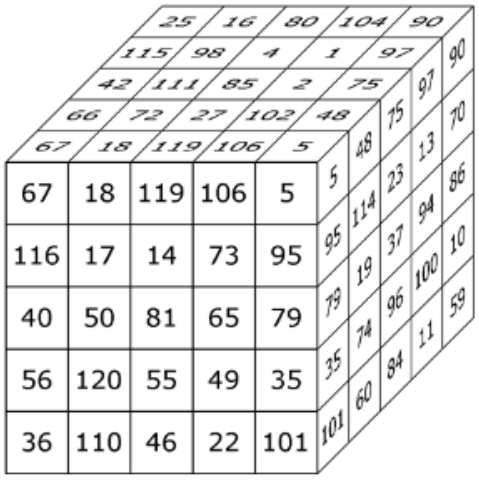
Daftar Isi 2
Bab 1 - Deskripsi Persoalan 3
Bab 2 - Pembahasan 5
- I. Objective Function dan Pemilihannya 5
- II. Implementasi State Functions 5
- III. Hill Climbing 13
- IV. Cuplikan Kode Local Search 15
- V. Simulated Annealing 19
- VI. Cuplikan Kode Simulated Annealing 20
- VII. Genetic Algorithm 21
- VIII. Cuplikan Kode Genetic Algorithm 21

Bab 3 - Eksperimen 23
- I. Steepest Ascent Hill-Climbing 23
- II. Sideways Move Hill-Climbing 30
- III. Random Restart Hill Climbing 37
- IV. Stochastic Hill-Climbing 47
- V. Simulated Annealing 57
- VI. Genetic Algorithm 66

Bab 4 - Analisis dan Pembahasan 94
Bab 5 - Kesimpulan dan Saran 97
Bab 6 - Pembagian Kerja 98
Bab 7 - Referensi 99

Bab 1 - Deskripsi Persoalan

Diagonal magic cube merupakan kubus yang tersusun dari angka 1 hingga n^3 tanpa pengulangan dengan n adalah panjang sisi dari magic cube tersebut. Sebuah magic cube yang berukuran n memiliki n^2 baris, n^2 kolom, n^2 tiang, 6n diagonal bidang, dan 4 diagonal ruang. Untuk persoalan ini, magic cube nya berukuran 5x5x5. Oleh karena itu, magic cube ini memiliki 25 baris, 25 kolom, 25 tiang, 30 diagonal bidang, dan 4 diagonal ruang.

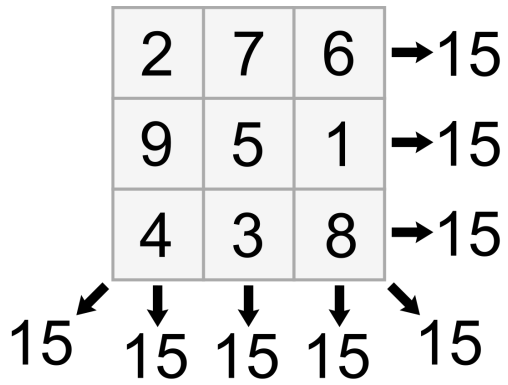


Gambar 1 : Magic Cube 5x5x5

Angka-angka pada kubus tersusun sedemikian rupa sehingga komponen-komponen seperti berikut terpenuhi:

1. Terdapat satu angka yang merupakan magic number dari kubus tersebut (Magic number tidak harus termasuk dalam rentang 1 hingga n^3, magic number juga bukan termasuk ke dalam angka yang harus dimasukkan ke dalam kubus
2. Jumlah angka-angka untuk setiap baris harus sama dengan magic number
3. Jumlah angka-angka untuk setiap kolom harus sama dengan magic number
4. Jumlah angka-angka untuk setiap tiang harus sama dengan magic number
5. Jumlah angka-angka untuk seluruh diagonal ruang pada kubus harus sama dengan magic number
6. Jumlah angka-angka untuk seluruh diagonal pada suatu potongan bidang dari kubus sama dengan magic number

Intinya, Angka-angka yang pada setiap kubus harus memenuhi konfigurasi pada gambar dibawah ini:



Gambar 2 : Konfigurasi Magic Square

Bab 2 - *Pembahasan*

I. Objective Function dan Pemilihannya

Objective Function adalah fungsi yang digunakan untuk mencari nilai dari suatu state. Nilai ini akan dihitung saat state di saat itu sudah lengkap konfigurasinya. Pada permasalahan Magic Cube, nilai dari state bisa dijelaskan sebagai berikut.

a. Nilai state pada awalnya bernilai acak.
b. Nilai ini didapat dengan mengurangi 315 (Jumlah Angka dari Magic Cube yang sudah di state yang benar) dengan jumlah segmen dari salah satu komponen yang memenuhi sebuah Magic Cube (Misalkan salah satu tinggi dari kubus) lalu dikuadratkan dan dijumlahkan dengan operasi yang sama dengan jumlah segmen dari komponen yang berbeda. Rumusnya adalah sebagai berikut:

$$H = \sum_{i=1}^{109} \left(315 - \sum_{j=1}^{5} f_i(j) \right)^2$$

Gambar 1 - Rumus Heuristik

c. State akan disebut sebagai state yang lebih mendekati maksimum jika nilai statenya lebih dekat ke angka 0.
d. Jika tidak ada state dengan nilai yang lebih besar, state akan mencapai daerah optimal.
e. Jika state optimal memiliki nilai lebih dari 0, artinya state mencapai daerah optimum lokal yang artinya algoritma gagal mencapai tujuan.
f. Sebaliknya, jika state optimal memiliki nilai 0, artinya state mencapai daerah optimum global yang artinya algoritma berhasil mencapai tujuan dan Magic Cube berhasil terbentuk.

II. Implementasi State Functions

A. Representasi Class State

Implementasi State berbentuk array satu dimensi berukuran 125, yang merupakan bentuk *flatten* Matrix tiga dimensi berukuran 5x5x5, yang berisi nilai bilangan pada tiap block. Matrix akan berbentuk Matriks[x][y][z], dengan x adalah nomor baris, y adalah nomor kolom, dan z adalah nomor layer. Representasi segmen (baris, kolom, tiang, diagonal bidang, diagonal ruang) pada matriks ini adalah sebagai berikut.

1. Segmen Baris
   Baris pada matriks ini adalah deretan bilangan pada Matrix[0-4][m][m]

dengan m berupa bilangan bulat dari 0 sampai 4.

2. Segmen Kolom

   Kolom pada matriks ini adalah deretan bilangan pada Matrix[m][0-4][m] dengan m berupa suatu bilangan bulat antara 0 sampai 5.

3. Segmen DIagonal Bidang

   Diagonal bidang pada matriks ini adalah deretan bilangan pada

   1. Diagonal bidang pada Satu Layer

      Segmen diagonal bidang pada layer m adalah deretan bilangan pada Matriks[n][n][m] dan Matrix[4-n][4-n][m] dengan m berupa suatu bilangan bulat antara 0 dan 4 dan n adalah bilangan bulat dari 0 sampai 4.

   2. Diagonal bidang pada Satu Baris

      Segmen diagonal bidang pada layer m adalah deretan bilangan pada Matrix[m][n][n] dan Matrix[m][4-n][4-n] dengan m berupa suatu bilangan bulat antara 0 dan 4 dan n adalah bilangan bulat dari 0 sampai 4.

   3. Diagonal bidang pada Satu Kolom

      Segmen diagonal bidang pada layer m adalah deretan bilangan pada Matrix[n][m][n] dan Matrix[4-n][m][4-n] dengan m berupa suatu bilangan bulat antara 0 dan 4 dan n adalah bilangan bulat dari 0 sampai 4.

4. Segmen Diagonal Ruang

   Diagonal ruang pada matriks ini adalah deretan bilangan pada Matriks[n][n][n], Matriks[4-n][n][n], Matriks[n][4-n][n], dan Matriks[n][n][4-n] dengan n merupakan bilangan bulat dari 0 sampai 4.

Berikut adalah implementasi representasi State pada kode kami.

```go
type State struct {
    matriks [125]int
    buffer  [5][]int
    val     int
}

func NewState() State {
    var state State
    for i := 0; i < 125; i++ {
        state.matriks[i] = i + 1
    }

    for i := 0; i < 3; i++ {
        state.buffer[i] = make([]int, 25)
    }
    state.buffer[3] = make([]int, 30)
    state.buffer[4] = make([]int, 4)

    rand.Shuffle(len(state.matriks), func(i, j int) {
        state.matriks[i], state.matriks[j] = state.matriks[j], state.matriks[i]
    })

    state.calculateBuffer()
    state.val = state.objectiveFunction()

    return state
}
```

NewState sebagai constructor dari class State, dimana akan menghasilkan konfigurasi matriks Cube yang random. Disini akan dilakukan inisialisasi matriks, perhitungan buffer (akan dibahas berikutnya), dan perhitungan objective function.

Penambahan buffer di sini hanya sebagai bentuk optimasi aja. Buffer disini adalah ide penulis untuk mengurangi kompleksitas waktu dari perhitungan objective function. Dengan menggunakan buffer, kita tidak perlu melakukan iterasi tiap bilangan cube untuk menghitung jumlah bilangan pada semua segmen (baris, kolom, tiang, diagonal bidang, diagonal ruang).

Buffer disini menyimpan nilai total bilangan untuk tiap segmen (baris, kolom, tiang, diagonal bidang, diagonal ruang) dengan ketentuan sebagai berikut.

1. buffer[0] menyimpan jumlah untuk tiap baris pada Cube. Karena banyak baris adalah 25, maka buffer[0] memiliki elemen dari buffer[0][0] hingga buffer[0][24]
2. buffer[1] menyimpan jumlah untuk tiap kolom pada Cube. Karena banyak

kolom adalah 25, maka buffer[1] memiliki elemen dari buffer[1][0] hingga buffer[1][24]

3. buffer[2] menyimpan jumlah untuk tiap tiang pada Cube. Karena banyak tiang adalah 25, maka buffer[2] memiliki elemen dari buffer[2][0] hingga buffer[2][24]
4. buffer[3] menyimpan jumlah untuk tiap diagonal bidang pada Cube. Karena banyak diagonal bidang adalah 30, maka buffer[3] memiliki elemen dari buffer[3][0] hingga buffer[3][24]
5. buffer[4] menyimpan jumlah untuk tiap diagonal ruang pada Cube. Karena banyak diagonal ruang adalah 4, maka buffer[4] memiliki elemen dari buffer[1][0] hingga buffer[1][3]

Sistem index dari buffer ini sendiri adalah sebagai berikut.

I. Index pada buffer baris (buffer[0]), mulai dari baris pertama pada layer pertama, dan bertahap dengan bertambahnya baris. Jika sudah mencapai akhir baris pada layer tersebut, maka layer akan bertambah 1. Maka rumus indexing nya adalah sebagai berikut.

$$row_index = i + 5 * k$$

II. Index pada buffer kolom (buffer[1]), mulai dari kolom pertama pada layer pertama, dan bertahap dengan bertambahnya kolom. Jika sudah mencapai akhir kolom pada layer tersebut, maka layer akan bertambah 1. Rumus indexing nya adalah sebagai berikut.

$$col_index = j + 5 * k$$

III. Index pada buffer tiang (buffer[2]), mulai dari baris pertama pada kolom pertama, dan bertahap dengan bertambahnya baris. Jika sudah mencapai akhir baris pada layer tersebut, maka kolom akan bertambah 1. Rumus indexing nya adalah sebagai berikut.

$$tiang_index = i + 5 * j$$

IV. Urutan indexing diagonal bidang adalah diagonal bidang yang berada pada baris sama dahulu, kemudian diagonal bidang pada kolom yang sama, dan kemudian diagonal bidang pada layer yang sama. Diagonal bidang pada tiap orientasi akan memiliki range index sebanyak 10. Contohnya yaitu buffer[3][0] sampai buffer[3][9] berisi jumlah diagonal bidang pada tiap orientasi baris, buffer[3][10] sampai buffer[3][19] berisi jumlah diagonal bidang pada tiap orientasi kolom, dan buffer[3][29] berisi jumlah diagonal bidang berdasarkan orientasi layer. Index yang genap menyimpan diagonal

bidang yang bergradien positif (naik), sedangkan index ganjil untuk menyimpan diagonal bidang bergradien negatif (turun).

Untuk indexing pada diagonal bidang(buffer[3]) agak berbeda dari index pada baris,kolom, tiang yang hanya berjumlah 1. Suatu bilangan cube dapat mempengaruhi lebih dari satu diagonal bidang, dan suatu cube bisa juga bukan merupakan bagian dari diagonal bidang apapun. Suatu cube dikatakan merupakan bagian dari diagonal bidang jika dua atau lebih dari nilai i, j, k adalah sama. Berikut adalah algoritma pencarian index diagonal bidang dari suatu posisi i,j,k.

```go
func (s *State) diaBidangIndexes(i, j, k int) []int {
    var bidangIndexes []int
    if j == k {
        bidangIndexes = append(bidangIndexes, 2*i)
    }
    if j+k == 4 {
        bidangIndexes = append(bidangIndexes, 2*i+1)
    }
    if i == k {
        bidangIndexes = append(bidangIndexes, (2*j)+10)
    }
    if i+k == 4 {
        bidangIndexes = append(bidangIndexes, (2*j)+11)
    }
    if i == j {
        bidangIndexes = append(bidangIndexes, 2*k+20)
    }
    if i+j == 4 {
        bidangIndexes = append(bidangIndexes, 2*k+21)
    }
    return bidangIndexes
}
```

V. Untuk indexing buffer diagonal ruang(buffer[4]), deretan diagonal pada Matriks[n][n][n] menempati index 0, deretan pada Matriks[4-n][n][n] menempati index 1, deretan pada Matriks[n][4-n][n] menempati index 2, dan Matriks[n][n][4-n] menempati index 3. Berikut adalah algoritma pencarian index diagonal ruang dari suatu tuple i,j,k.

```go
func (s *State) diaRuangIndexes(i, j, k int) []int {
    var ruangIndexes []int
    if i == j && j == k {
        ruangIndexes = append(ruangIndexes, 0)
    }
    if (4-i) == j && j == k {
        ruangIndexes = append(ruangIndexes, 1)
    }
    if i == (4-j) && i == k {
        ruangIndexes = append(ruangIndexes, 2)
    }
    if i == j && j == (4-k) {
        ruangIndexes = append(ruangIndexes, 3)
    }
    return ruangIndexes
}
```

B. Perhitungan Buffer dan Perhitungan Objective Function

Perhitungan Buffer dilakukan dengan mengupdate buffer jumlah segmen baris, kolom, tiang, diagonal bidang, ataupun diagonal ruang dari suatu index terkait. Penentuan segmen mana yang index i,j,k merupakan bagian dari segmen tersebut, dilakukan dengan melihat aturan index dari tiap segmen yang telah disebutkan sebelumnya.

Implementasi perhitungan buffer ini terdapat dua kondisi, dimana perhitungan inisialisasi buffer, dan perhitungan update buffer ketika pertukaran antara dua bilangan cube. Berikut adalah bentuk fungsinya

```go
func (s *State) updateBuffer(i1, j1, k1 int, i2, j2, k2 int) {
```

Ketika proses inisialisasi, nilai parameter i2, j2, dan k2 akan bernilai -1. Sedangkan ketika proses penukaran, kedua index tersebut akan bernilai sesuai index kedua bilangan yang akan ditukar.

Perhitungan buffer ketika proses inisialisasi buffer adalah sebagai berikut.

```go
row1Idx := i1 + 5*k1
s.buffer[0][row1Idx] += val1

col1Idx := j1 + 5*k1
s.buffer[1][col1Idx] += val1

tiang1Idx := i1 + 5*j1
s.buffer[2][tiang1Idx] += val1

bidang1Indexes := s.diaBidangIndexes(i1, j1, k1)
for _, bidang1Idx := range bidang1Indexes {
    s.buffer[3][bidang1Idx] += val1
}

ruang1Indexes := s.diaRuangIndexes(i1, j1, k1)
for _, ruang1Idx := range ruang1Indexes {
    s.buffer[4][ruang1Idx] += val1
}
```

Perhitungan Objective Function dilakukan dengan cukup mengiterasi buffer dari tiap segmen. Jumlah dari quadratic difference dari tiap segmen ini kemudian disimpan sebagai nilai state value.

```go
func (s *State) objectiveFunction() int {
    result := 0
    for i := 0; i < 5; i++ {
        for _, sum := range s.buffer[i] {
            result += int(math.Pow(float64(MAGIC_NUMBER-sum), 2))
        }
    }
    return result
}
```

Sedangkan untuk proses penukaran dua bilangan, akan dilakukan pengupdatean buffer terkait dari masing-masing bilangan. Akan dilakukan juga pengupdatean state value sekaligus saat pengupdatean buffer.

C. Pembangkitan Suksesor

Pembangkitan suksesor dapat dilakukan dalam dua cara, yakni Highest Valued

Successor dan Random Successor. Berikut adalah implementasi dari kedua algoritma tersebut.

```go
func (s *State) randomSucc() (State, int, int) {
    var arr1, arr2 [3]int
    for {
        for i := 0; i < 3; i++ {
            arr1[i] = rand.Intn(5)
            arr2[i] = rand.Intn(5)
        }
        if arr1 != arr2 {
            break
        }
    }

    firstSwapIdx := 25*arr1[0] + 5*arr1[1] + arr1[2]
    secondSwapIdx := 25*arr2[0] + 5*arr2[1] + arr2[2]

    return s.generateSucc(arr1[0], arr1[1], arr1[2], arr2[0], arr2[1], arr2[2]), firstSwapIdx, secondSwapIdx
```

```go
func (s *State) highestValuedSucc() (State, int, int) {
    bestState := s.copy()
    var FirstSwapIndex int
    var SecondSwapIndex int
    idx := 0
    for i := 0; i < 5; i++ {
        for j := 0; j < 5; j++ {
            for k := 0; k < 5; k++ {
                for l := (25*i + 5*j + k + 1); l < 125; l++ {
                    iSucc := l / 25
                    jSucc := (l % 25) / 5
                    kSucc := l % 5
                    succ := s.generateSucc(i, j, k, iSucc, jSucc, kSucc)

                    if succ.getStateValue() < bestState.getStateValue() {
                        bestState = succ
                        FirstSwapIndex = 25*i + 5*j + k
                        SecondSwapIndex = l
                    }
                }
                idx++
            }
        }
    }

    return bestState, FirstSwapIndex, SecondSwapIndex
}
```

Algoritma *Hill Climbing* adalah salah satu algoritma optimasi dari Local Search yang bertujuan untuk mencari solusi maksimum dalam ruang pencarian yang mungkin dibangkitkan. Tujuan dari algoritma ini adalah mencari titik dalam ruang pencarian yang memberikan nilai tertinggi atau terendah tergantung pada jenis masalah yang sedang diselesaikan.

Prinsip dasar algoritma ini adalah membandingkan solusi saat ini dengan solusi yang berada di sekitarnya. Algoritma ini akan memilih solusi yang lebih baik dari yang ada saat ini, seolah-olah sedang mencari puncak bukit. Algoritma ini menganggap bahwa titik adalah puncak lokal atau global ketika state selanjutnya tidak memiliki nilai yang lebih tinggi daripada state yang sekarang.

Algoritma *Hill Climbing* memiliki langkah-langkah sebagai berikut:

1. Inisialisasi: Solusi awal dipilih secara acak berdasarkan jenis algoritma *Hill Climbing* yang dipilih
2. Evaluasi: Hitung nilai atau skor dari solusi saat ini menggunakan fungsi evaluasi atau fungsi tujuan yang telah ditentukan sebelumnya. Nilai ini menunjukkan seberapa baik solusi tersebut dalam konteks masalah yang sedang dihadapi.
3. Generasi Solusi Baru: Buat variasi kecil dari solusi saat ini. Variasi ini dapat diperoleh dengan melakukan perubahan kecil pada solusi, misalnya, mengubah nilai beberapa parameter atau komponen.
4. Evaluasi Solusi Baru: Hitung nilai atau skor dari solusi baru yang dihasilkan dalam langkah sebelumnya.
5. Pemilihan Solusi: Jika solusi baru memiliki nilai yang lebih baik daripada solusi sebelumnya, maka solusi baru tersebut menjadi solusi saat ini. Jika tidak, algoritma ini berhenti atau mencoba variasi solusi lain.
6. Iterasi: Langkah 3 hingga 5 diulang secara berulang sampai kondisi berhenti tercapai, seperti jumlah iterasi yang telah ditentukan, solusi

III. *Hill Climbing*

Algoritma *Hill Climbing* adalah salah satu algoritma optimasi dari Local Search yang bertujuan untuk mencari solusi maksimum dalam ruang pencarian yang mungkin dibangkitkan. Tujuan dari algoritma ini adalah mencari titik dalam ruang pencarian yang memberikan nilai tertinggi atau terendah tergantung pada jenis masalah yang sedang diselesaikan.

Prinsip dasar algoritma ini adalah membandingkan solusi saat ini dengan solusi yang berada di sekitarnya. Algoritma ini akan memilih solusi yang lebih baik dari yang ada saat ini, seolah-olah sedang mencari puncak bukit. Algoritma ini menganggap bahwa titik adalah puncak lokal atau global ketika state selanjutnya tidak memiliki nilai yang lebih tinggi daripada state yang sekarang.

Algoritma *Hill Climbing* memiliki langkah-langkah sebagai berikut:

1. Inisialisasi: Solusi awal dipilih secara acak berdasarkan jenis algoritma *Hill Climbing* yang dipilih
2. Evaluasi: Hitung nilai atau skor dari solusi saat ini menggunakan fungsi evaluasi atau fungsi tujuan yang telah ditentukan sebelumnya. Nilai ini menunjukkan seberapa baik solusi tersebut dalam konteks masalah yang sedang dihadapi.
3. Generasi Solusi Baru: Buat variasi kecil dari solusi saat ini. Variasi ini dapat diperoleh dengan melakukan perubahan kecil pada solusi, misalnya, mengubah nilai beberapa parameter atau komponen.
4. Evaluasi Solusi Baru: Hitung nilai atau skor dari solusi baru yang dihasilkan dalam langkah sebelumnya.
5. Pemilihan Solusi: Jika solusi baru memiliki nilai yang lebih baik daripada solusi sebelumnya, maka solusi baru tersebut menjadi solusi saat ini. Jika tidak, algoritma ini berhenti atau mencoba variasi solusi lain.
6. Iterasi: Langkah 3 hingga 5 diulang secara berulang sampai kondisi berhenti tercapai, seperti jumlah iterasi yang telah ditentukan, solusi yang cukup memuaskan telah ditemukan atau tidak ada solusi yang lebih baik yang dapat ditemukan.

Terdapat berbagai jenis-jenis algoritma *Hill Climbing* dan berikut cara menggunakannya pada permasalahan Magic Cube, yaitu:

1. Steepest Ascent Hill Climbing
    a. Pencarian dimulai dengan current state berupa State dengan konfigurasi matrix random
    b. Cari neighbor state dengan mengambil suksesor state yang memiliki nilai objective tertinggi, yakni state dengan banyak segmen yang valid (berjumlah sama dengan magic number), terbanyak
    c. Current state yaitu jumlah segmen yang sama dengan magic number dianggap sudah mencapai peak jika jumlahnya lebih banyak daripada neighbor state yang dipilih (nilai objective function tertinggi). Jika mencapai kondisi tersebut, kembalikan nilai current state
    d. Sebaliknya, neighbor state dijadikan current state. Ulangi langkah 2 hingga current state mencapai lokal atau global maksimum
2. Sideways Move Hill Climbing
    a. Pencarian dimulai dengan current state berupa State dengan konfigurasi matrix random
    b. Cari neighbor state dengan mengambil suksesor state yang memiliki nilai objective function tertinggi, yakni state dengan banyak segmen yang valid (berjumlah sama dengan magic number), terbanyak.
    c. Jika jumlah segmen yang valid dari neighbor, lebih kecil dari jumlah segmen valid current state, maka current state dipandang sebagai solusi optimum. Kembalikan current state

d. Jika tidak, current state diassign dengan neighbor. Ulangi dari langkah 2, hingga state value neighbor lebih kecil dari current state value.

3. Random Restart Hill Climbing
    a. Pencarian dimulai dengan current state berupa State dengan konfigurasi matrix random
    b. Cari neighbor state dengan mengambil suksesor state yang memiliki nilai objective function tertinggi, yakni state dengan banyak segmen yang valid (berjumlah sama dengan magic number), terbanyak
    c. Dengan menggunakan jenis algoritma Hill Climbing, kemungkinan nilai maksimal lokal bisa tercapai. Jika yang dicapai adalah nilai maksimal global, kembalikan nilai tersebut
    d. Jika tidak, lakukan Hill Climbing kembali tetapi dengan konfigurasi matrix random lainnya
    e. Lakukan hal ini sampai nilai maksimal global (segmen dengan magic number terbanyak) dicapai agar bisa dikembalikan nilai tersebut.
4. Stochastic Hill Climbing
    a. Pencarian dimulai dengan current state berupa State dengan konfigurasi matrix random
    b. Cari suksesor state secara acak
    c. Current state diganti oleh neighbor state jika value neighbor lebih besar current
    d. Lakukan berulang-berulang sampai iterasi ke n
    e. Jika sudah selesai, kembalikan nilai current state

IV. Cuplikan Kode Local Search

1. Steepest Ascent Hill Climbing

Proses -> Mencari neighbor state dengan nilai heuristik yang lebih besar dari sebelumnya sampai mencapai daerah optimal
Hasil -> State Cube dari daerah optimal lokal atau global

```go
func SteepestAscentHC() DataFormat {
    var current State = NewState()
    df := DataFormat{
        InitialState: current.getMatrix(),
    }

    start := time.Now()
    for {
        neighbor, FirstSwapIndex, SecondSwapIndex := current.highestValuedSucc()
        if neighbor.getStateValue() >= current.getStateValue() {
            break
        }
        df.ObjEachStep = append(df.ObjEachStep, neighbor.getStateValue())
        df.FirstSwapIndex = append(df.FirstSwapIndex, FirstSwapIndex)
        df.SecondSwapIndex = append(df.SecondSwapIndex, SecondSwapIndex)

        current = neighbor
    }
    elapsed := time.Since(start)

    df.LastState = current.getMatrix()
    df.Duration = elapsed.Seconds()
    df.BestObj = current.val
    return df
}
```

2. Sideways Move Hill Climbing

Proses -> Mencari neighbor state dengan nilai heuristik yang lebih besar atau sama dari sebelumnya sampai mencapai daerah optimal dengan syarat jumlah iterasi yang diizinkan masih memenuhi
Hasil -> State Cube dari daerah optimal lokal atau global

```go
func SideWaysMoveHC(maxSideways int) DataFormat {
    var current State = NewState()
    df := DataFormat{
        InitialState: current.getMatrix(),
    }

    start := time.Now()
    sidewaysMove := 0
    for {
        neighbor := current.highestValuedSucc()

        df.ObjEachStep = append(df.ObjEachStep, neighbor.getStateValue())

        if neighbor.getStateValue() > current.getStateValue() {
            break
        } else if neighbor.getStateValue() == current.getStateValue() {
            sidewaysMove++
            if sidewaysMove == maxSideways {
                break
            }
        } else {
            sidewaysMove = 0
        }

        current = neighbor
    }
    elapsed := time.Since(start)

    df.LastState = current.getMatrix()
    df.Duration = elapsed.Seconds()
    df.BestObj = current.val
    return df
}
```

3. Random Restart Hill Climbing

Proses -> Mencari neighbor state dengan nilai heuristik yang lebih besar dari sebelumnya sampai mencapai daerah optimal dan dilakukan berkali-kali sampai jumlah iterasi restart habis
Hasil -> State Cube dari daerah optimal lokal atau global

```go
func RandomRestartHC(maxRestart int) DataFormat {
    var current State
    var df DataFormat
    var bestState = State{val: math.MaxInt}
    start := time.Now()
    numRestart := 0
    for{
        current = NewState()
        if(numRestart == 0){
            df = DataFormat{
                InitialState: current.getMatrix(),
            }
        }

        df.RestartObj = append(df.RestartObj, []int{})
        iteration := 0
        for {
            iteration++
            neighbor := current.highestValuedSucc()
            df.RestartObj[numRestart] = append(df.RestartObj[numRestart], neighbor.getStateValue())
            if neighbor.getStateValue() >= current.getStateValue() {
                break
            }

            current = neighbor
        }
        if(current.val < bestState.val){
            bestState = current
        }

        df.IterationPerRestart = append(df.IterationPerRestart, iteration)

        numRestart++
        if(numRestart == maxRestart || bestState.val == GOAL_OBJ ){
            break
        }
    }
    elapsed := time.Since(start)

    df.NumRestarts = numRestart;
    df.LastState = bestState.matriks
    df.Duration = elapsed.Seconds()
    df.BestObj = bestState.val
    return df
}
```

4. Stochastic Hill Climbing

Proses -> Mencari neighbor state dengan nilai heuristik yang lebih besar dari sebelumnya dengan menerima state acak yang nilainya lebih tinggi dan menolak state acak yang nilainya lebih rendah
Hasil -> State Cube dari daerah optimal lokal atau gl

```go
func Stochastic() DataFormat {
    var current State = NewState()
    df := DataFormat{
        InitialState: current.getMatrix(),
    }

    start := time.Now()
    for i := 0; i < NMAX; i++ {
        neighbor := current.randomSucc()
        df.ObjEachStep = append(df.ObjEachStep, neighbor.getStateValue())

        if neighbor.getStateValue() < current.getStateValue() {
            current = neighbor
        }
    }
    elapsed := time.Since(start)

    df.LastState = current.getMatrix()
    df.Duration = elapsed.Seconds()
    df.BestObj = current.val
    return df
}
```

V. Simulated Annealing

Simulated Annealing adalah teknik probabilistik untuk mereka nilai optimal global dari sebuah fungsi yang diberikan. Simulated Annealing adalah salah satu jenis algoritma dari Local Search. Algoritma ini diharapkan juga bisa menemukan nilai lokal atau global maksimal. Namun, caranya berbeda dengan *Hill Climbing* dan memiliki variabel baru yang perlu diperhatikan yaitu threshold dan suhu. Beginilah tahapan dari Simulated Annealing pada permasalahan Magic Cube.

a. Pencarian dimulai dengan current state berupa State dengan konfigurasi matrix random
b. Cari suksesor state secara acak
c. Bangkitkan nilai T dengan ekspektasi T positif dan berkurang setiap iterasi
d. Jika $\Delta E > 0$, current state diganti dengan suksesor state
e. Jika $\Delta E < 0$, current state kemungkinan diganti dengan suksesor state dengan probabilitas $e^{\Delta E/T}$. Semakin besar perhitungan probabilitas, semakin besar kemungkinan pindah ke suksesor state
f. Hal ini dilakukan sampai T = 0 yang artinya nilai dikembalikan

VI. Cuplikan Kode Simulated Annealing

Proses -> Mencari neighbor state dengan nilai heuristik yang lebih besar dari sebelumnya dengan menerima state acak yang nilainya lebih tinggi dan menerima state acak yang nilainya lebih rendah dengan syarat nilai probabilitas yaitu $e^{\Delta E/T}$ lebih tinggi dari threshold

Hasil -> State Cube dari daerah optimal lokal atau global

```go
func SimulatedAnnealing() DataFormat {
    var current State = NewState()
    df := DataFormat{
        InitialState: current.getMatrix(),
    }

    start := time.Now()
    t := 0
    for {
        T := Schedule(t)
        if T <= 0.01 {
            break
        }
        neighbor := current.randomSucc()
        df.ObjEachStep = append(df.ObjEachStep, neighbor.getStateValue())

        deltaE := neighbor.getStateValue() - current.getStateValue()
        if deltaE < 0 {
            current = neighbor
        } else {
            eulerVal := math.Exp(float64(-deltaE) / T)
            df.PlotE = append(df.PlotE, eulerVal)
            if rand.Float64() <= eulerVal {
                current = neighbor
            }
        }
        t++
    }
    elapsed := time.Since(start)

    df.LastState = current.getMatrix()
    df.Duration = elapsed.Seconds()
    df.BestObj = current.val
    return df
}
```

VII. Genetic Algorithm

Genetic Algorithm adalah teknik yang terakhir dalam menyelesaikan permasalahan optimasi seperti Magic Cube. Algoritma ini memperbaharui populasi dari calon solusi. Dalam setiap iterasi, algoritma ini memilih beberapa state lalu dicampurkan satu sama lain berdasarkan titik mutasi lalu terdapat pencampuran kembali dengan angka-angka lain di titik mutasi juga. Proses ini dilakukan sampai lokal atau global maksimum tercapai. Berikut adalah tahapan dari Genetic Algorithm pada permasalahan Magic Cube:

a. Pencarian dimulai dengan parameter yang berupa suatu populasi dan objective functionnya. Disini, kami menggunakan populasi awal yaitu random generated matriks dan objective function sesuai yang telah kami definisikan sebelumnya.
b. Mendefinisikan set kosong
c. Memilih dua state dari populasi, kita sebut X dan Y (selection) dengan menggunakan cara roulette wheel. Persenan dari tiap individu state pada populasi didapat dari individu state value dibagi total state value populasi
d. Menghasilkan child state dari parent state X dan Y, dengan melakukan crossover.
e. Proses crossover menggunakan Order Crossover (OX1). Dalam proses cross over nya, bentuk matriks state akan diubah terlebih dahulu menjadi array satu dimensi berisi 125 elemen. Kita akan mengambil setengah pertama konfigurasi dari state X, dan menggunakan OX1 untuk menentukan setengah lain dari state Y.
f. Setelah mendapatkan child state, dalam probability yang kecil, dilakukan mutasi kepada child dengan menukar dua bilangan pada child.
g. Tambahkan child state kepada state kosong
h. Ulangi proses c - g sebanyak jumlah populasi (dalam konteks Magic Cube, sebanyak 125 kali).
i. Setelah set kosong sudah terisi, assign set yang terisi tersebut kepada variabel.
j. Cek apakah ada individual state dalam population yang memiliki state value maximum. Jika ada, return individual state tersebut sebagai solusi
k. Jika belum ada, ulangi proses pencarian dari (b)

VIII. Cuplikan Kode Genetic Algorithm

Proses -> Membuat state baru yang merupakan hasil dari 2 neighbor state yang dicampur dan dimutasi
Hasil -> State Cube dari daerah optimal lokal atau global

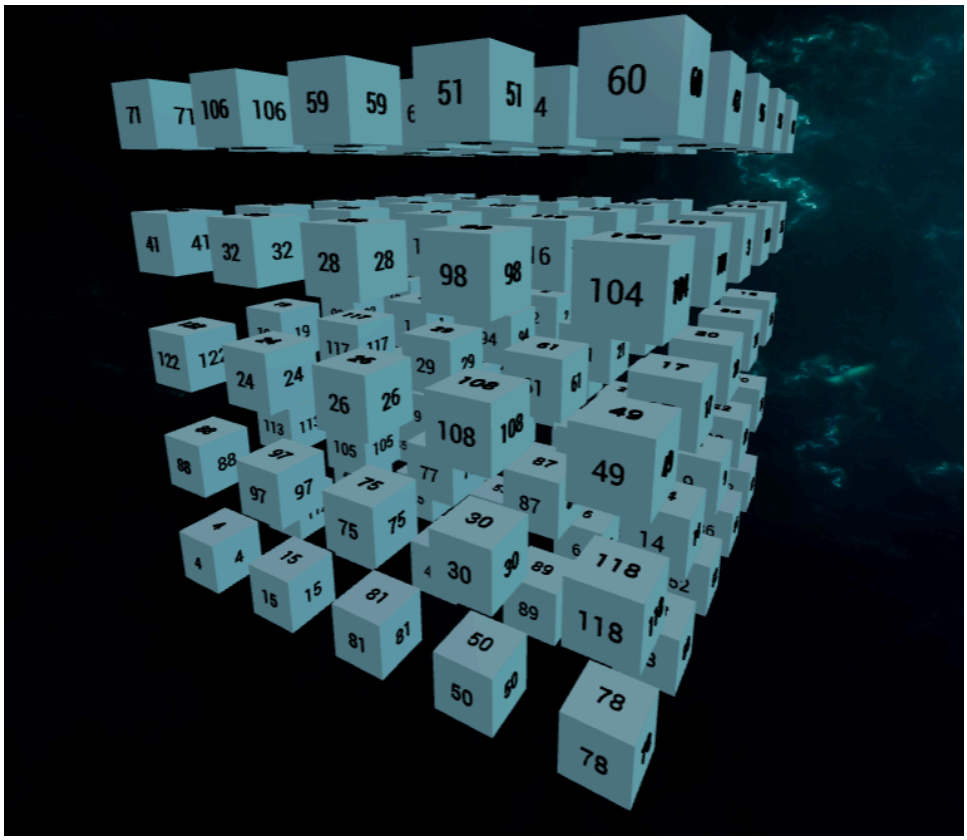
```go
func GeneticAlgorithm(nPopulation, nIteration int) GeneticFormat {
    var gf GeneticFormat

    var population []State
    totalValue := 0
    for i := 0; i < nPopulation; i++ {
        state := NewState()
        population = append(population, state)
        gf.InitialPopulation = append(gf.InitialPopulation, state.matriks)
        totalValue += state.val
    }

    start := time.Now()

    var bestOverall = State{val: math.MaxInt}
    iteration := 0
    for iteration < nIteration && bestOverall.val > 0 {
        new_population := []State{}
        var bestChild = State{val: math.MaxInt}
        newTotalValue := 0

        for i := 0; i < nPopulation; i++ {
            x := randomSelectionGenetic(population, totalValue, nPopulation)
            y := randomSelectionGenetic(population, totalValue, nPopulation)
            child := OX1(population[x], population[y])
            if mutateProb() {
                child = child.randomSucc()
            }
            if child.val < bestChild.val {
                bestChild = child
            }
            new_population = append(new_population, child)
            newTotalValue += child.val
        }
        iteration++
        population = new_population
        totalValue = newTotalValue
        if bestChild.val < bestOverall.val {
            bestOverall = bestChild
        }

        gf.MaxPerIteration = append(gf.MaxPerIteration, bestChild.val)
        gf.AvgPerIteration = append(gf.AvgPerIteration, float64(newTotalValue)/float64(nPopulation))
    }

    elapsed := time.Since(start)

    gf.BestState = bestOverall.matriks
    gf.BestObj = bestOverall.val
    gf.Duration = elapsed.Seconds()
    gf.NumIterations = iteration
    for _, state := range population {
        gf.LastPopulation = append(gf.LastPopulation, state.matriks)
    }

    return gf
}
```

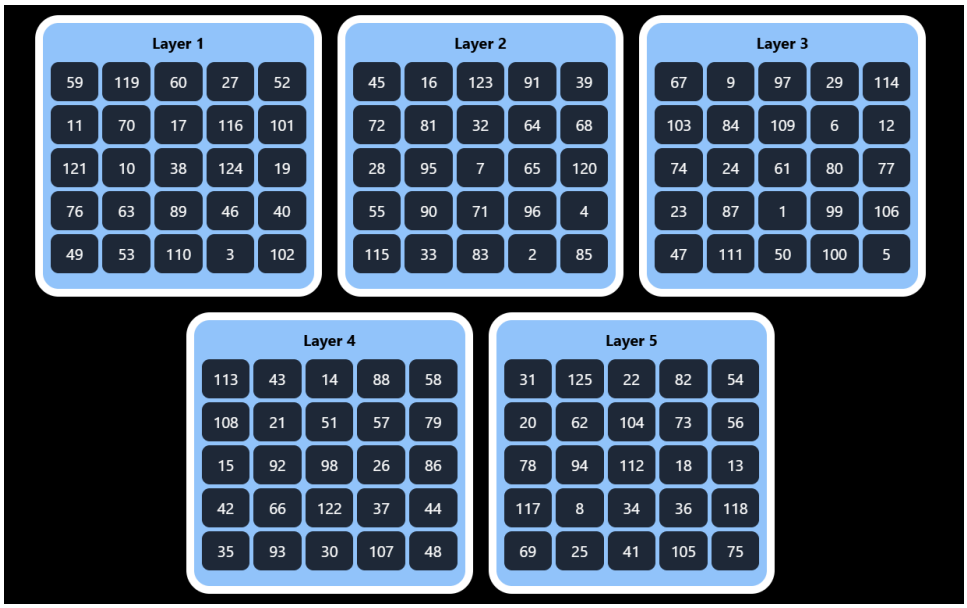
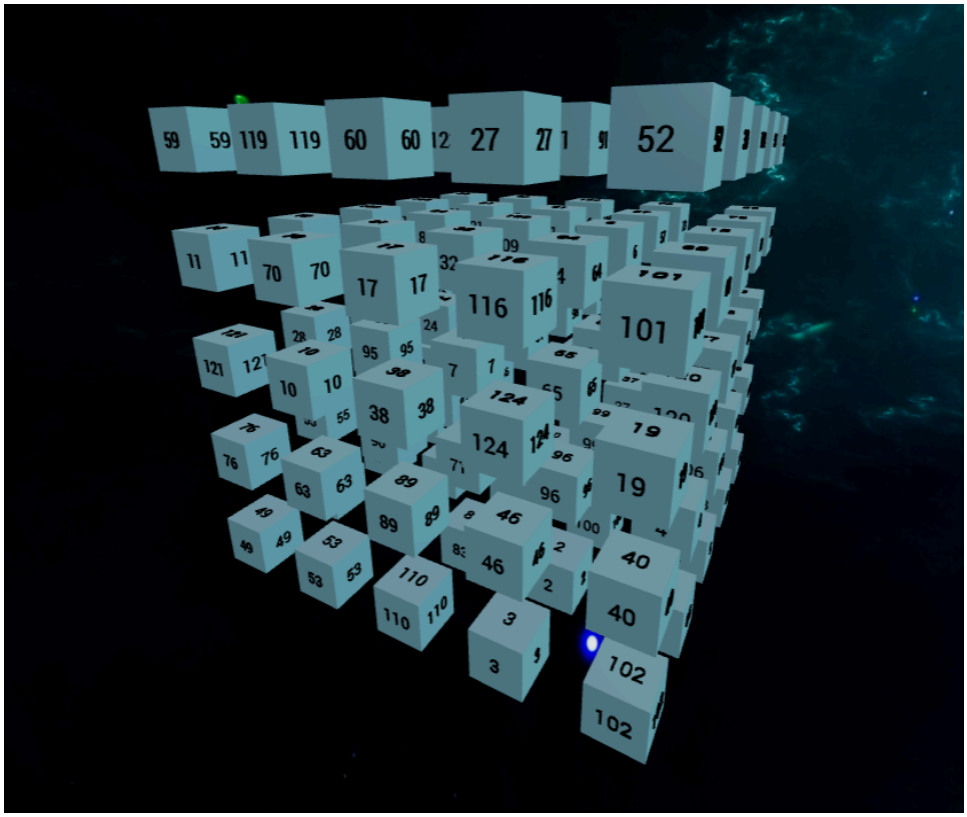
Bab 3 - Eksperimen

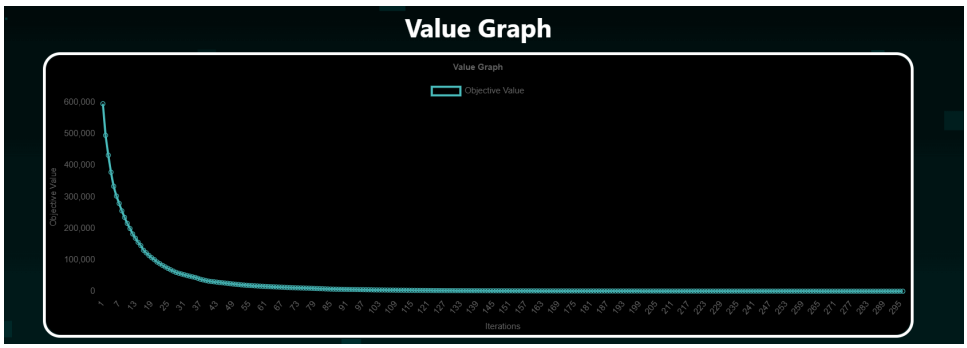
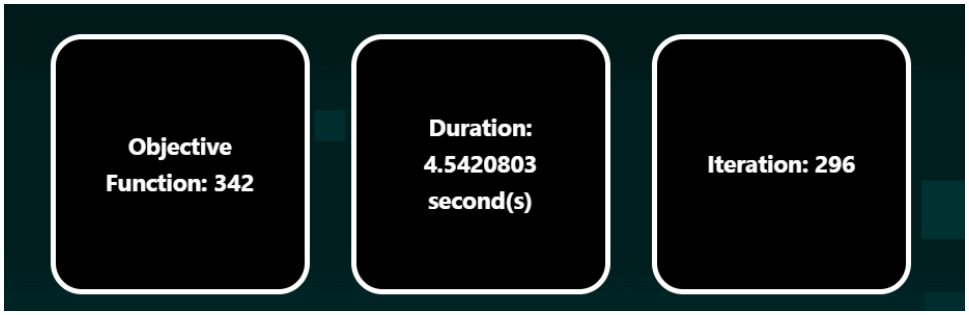
I. Steepest Ascent Hill-Climbing

a. Eksperimen 1

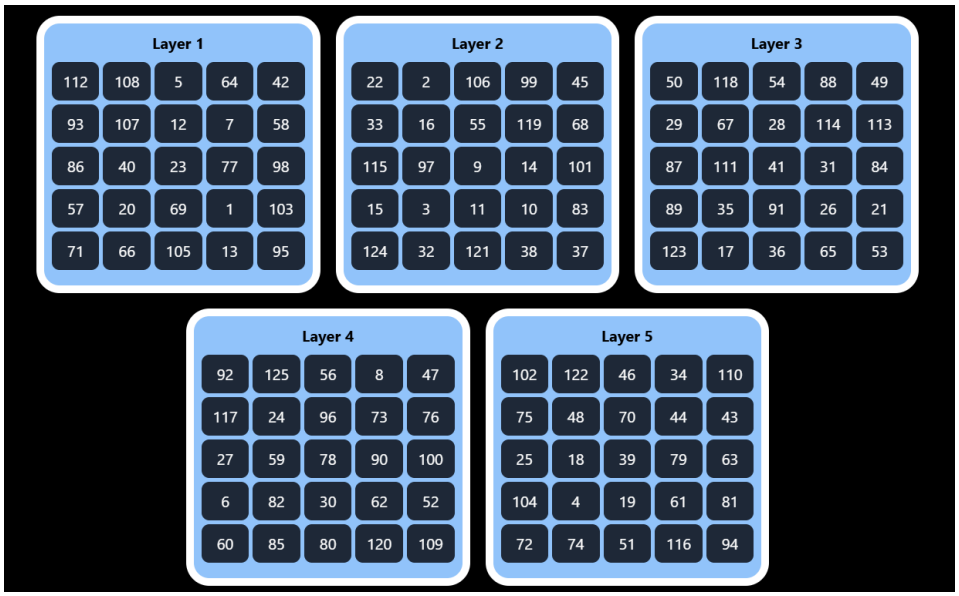
State Awal

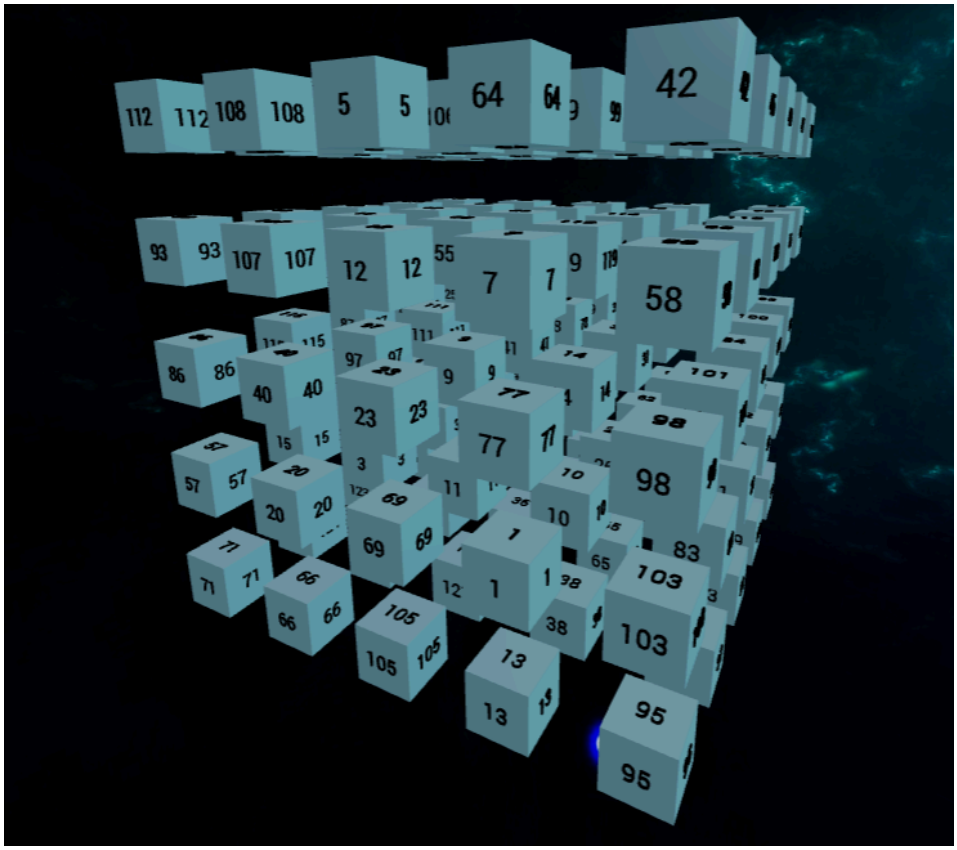


State Akhir	 **Layer 1** 59 119 60 27 52 11 70 17 116 101 121 10 38 124 19 76 63 89 46 40 49 53 110 3 102 **Layer 2** 45 16 123 91 39 72 81 32 64 68 28 95 7 65 120 55 90 71 96 4 115 33 83 2 85 **Layer 3** 67 9 97 29 114 103 84 109 6 12 74 24 61 80 77 23 87 1 99 106 47 111 50 100 5 **Layer 4** 113 43 14 88 58 108 21 51 57 79 15 92 98 26 86 42 66 122 37 44 35 93 30 107 48 **Layer 5** 31 125 22 82 54 20 62 104 73 56 78 94 112 18 13 117 8 34 36 118 69 25 41 105 75 
Objective Function	Objective Function: 342

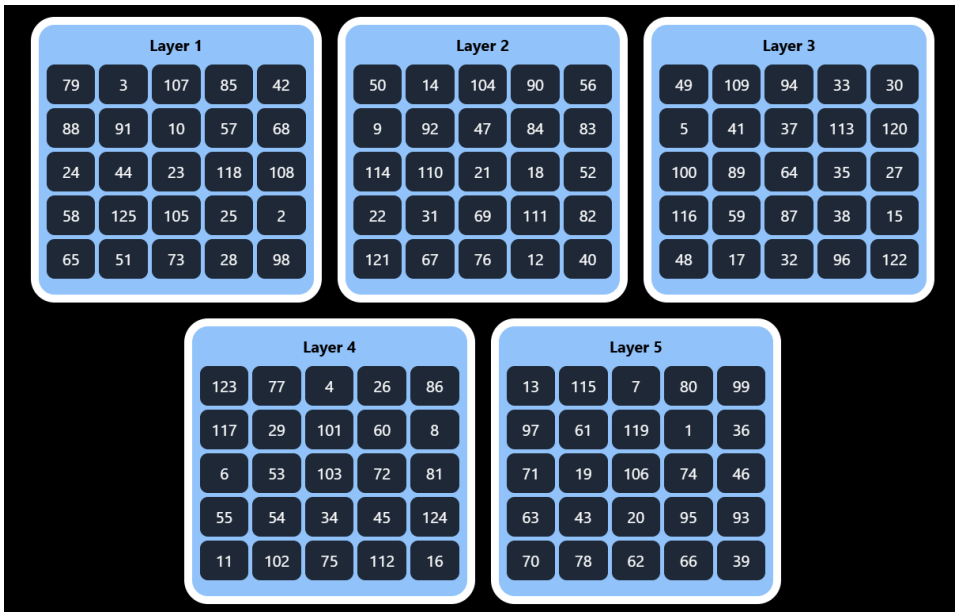
	
Durasi	Durasi: 4.5420803 seconds 

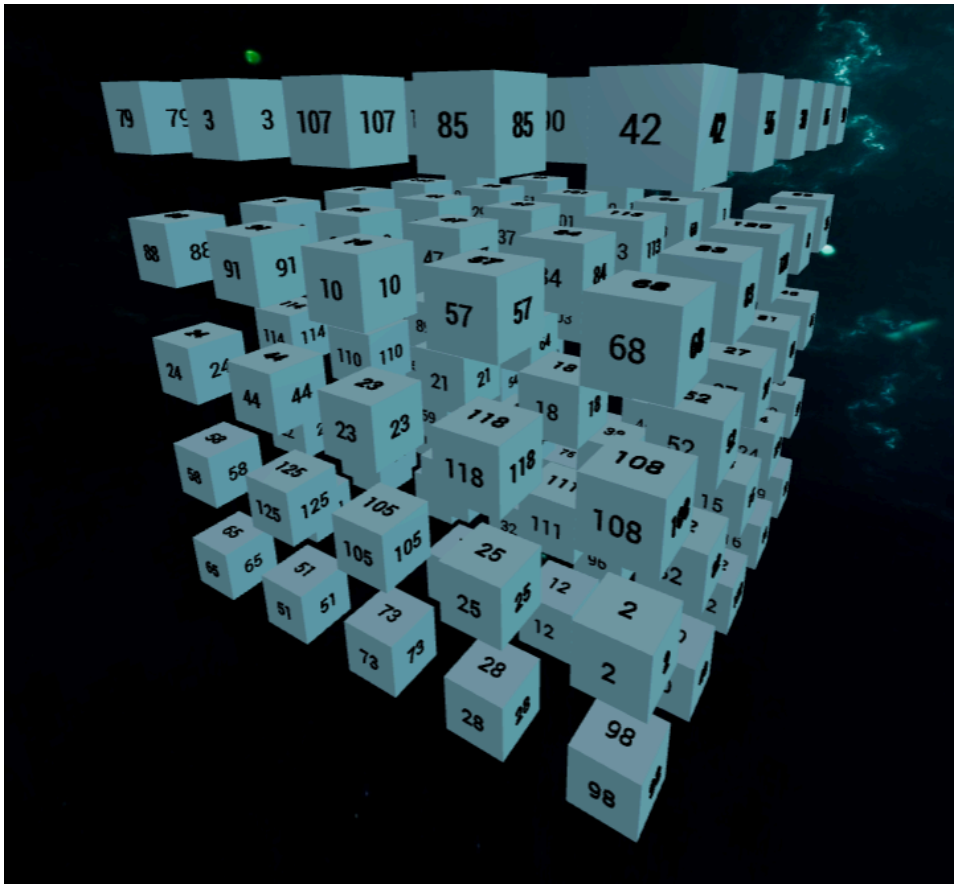
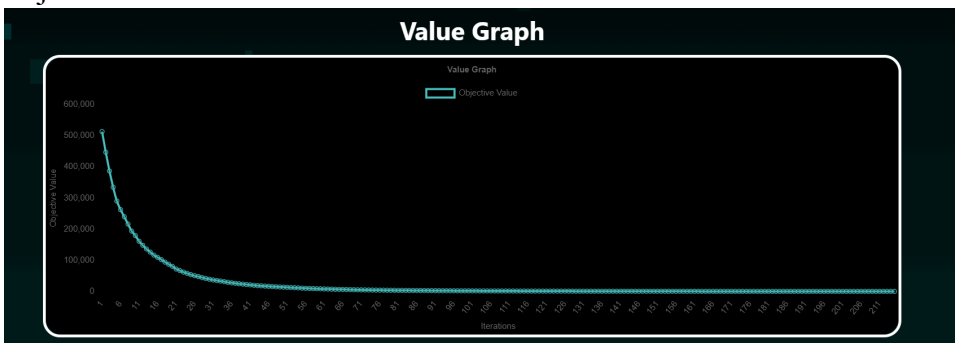
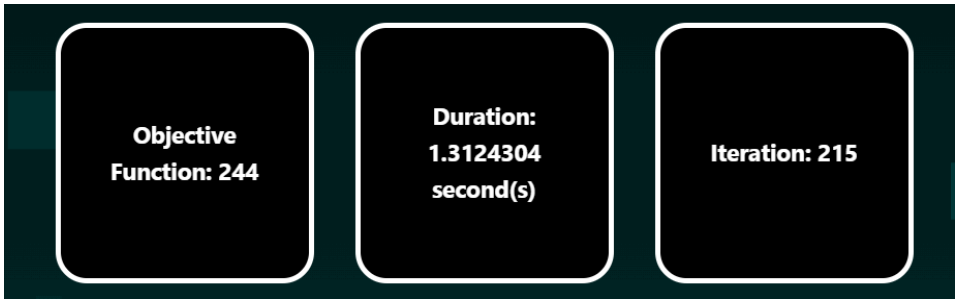
b. Eksperimen 2

State Awal	

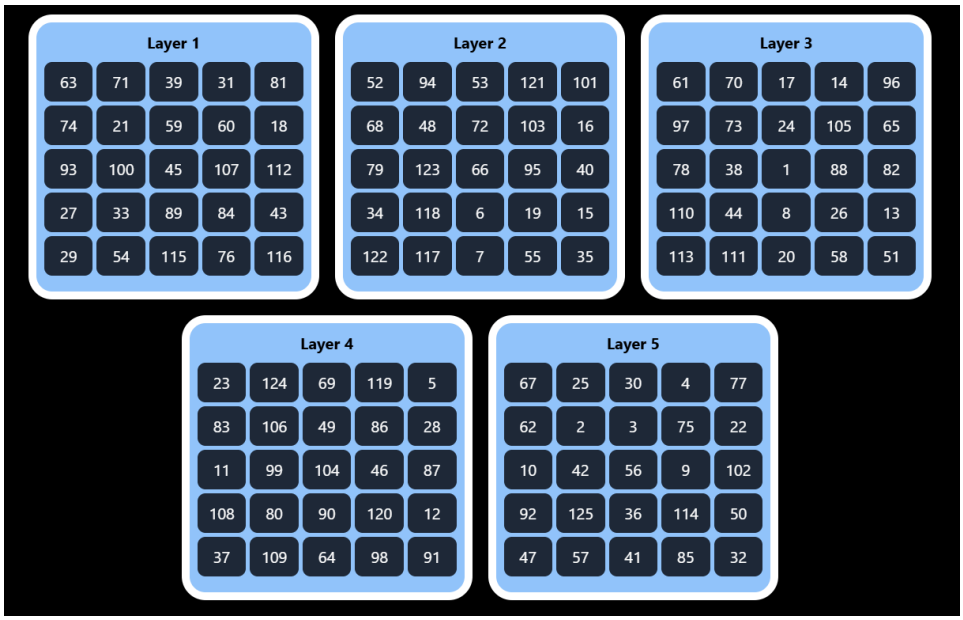
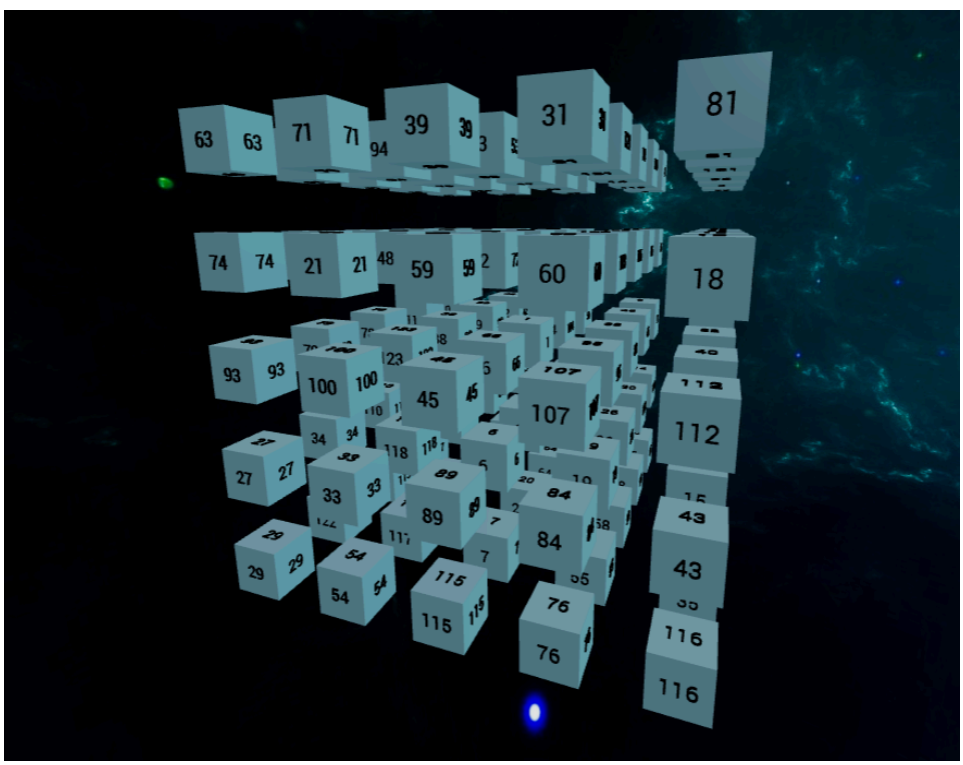


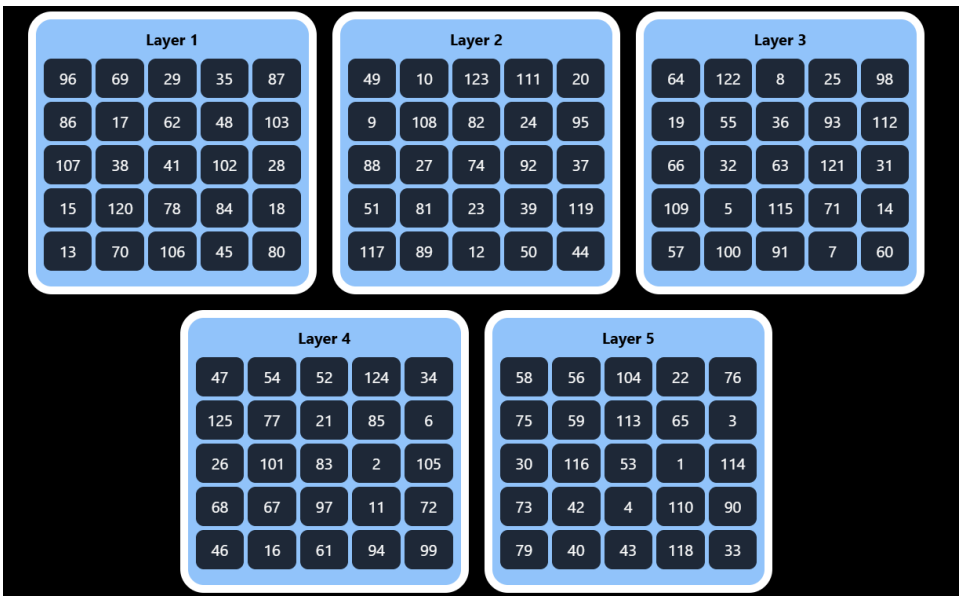
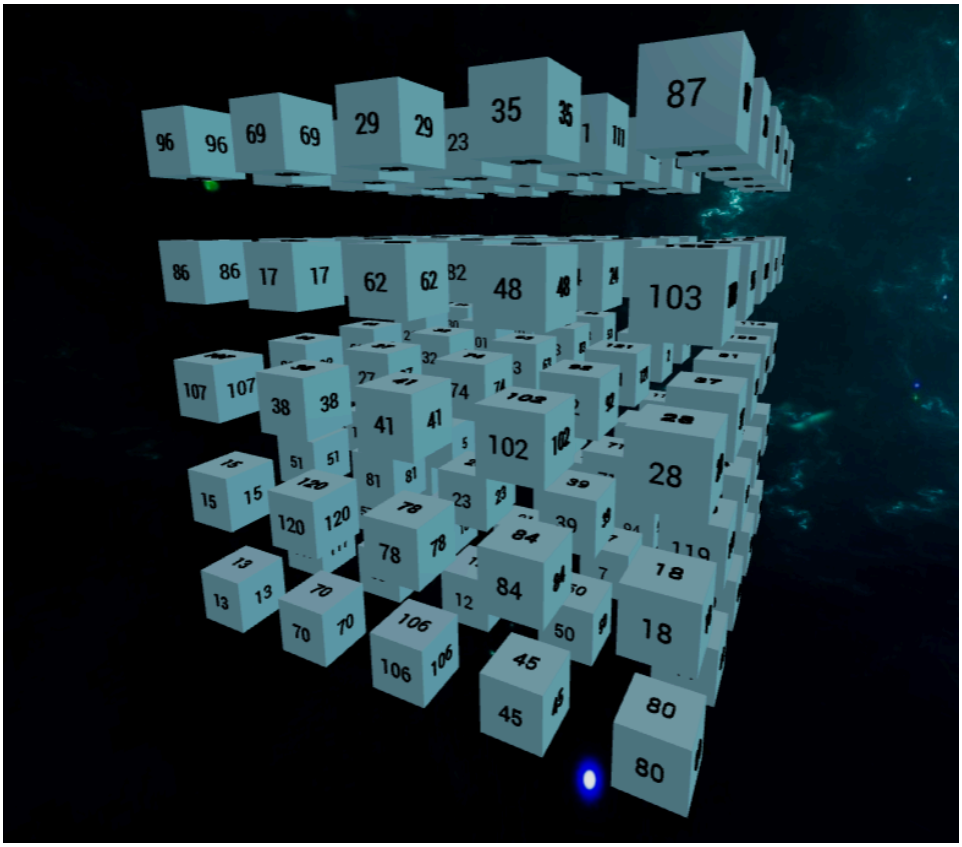
State Akhir

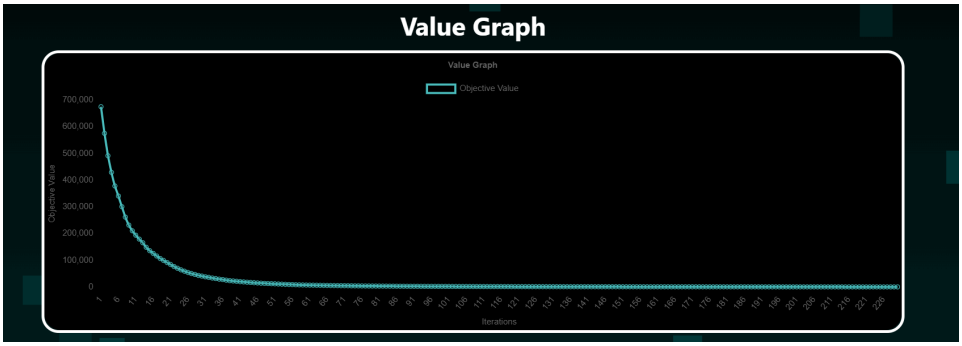
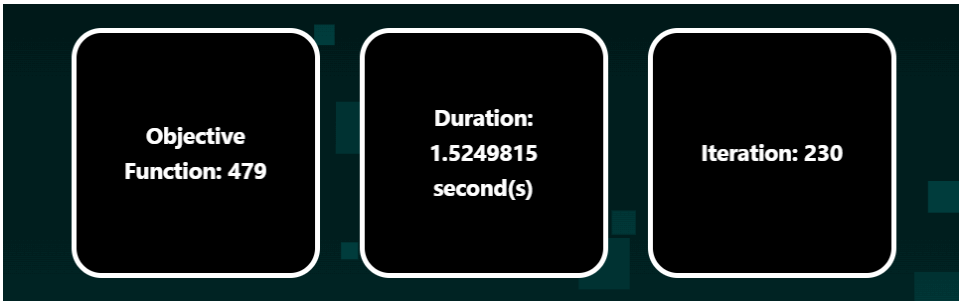

Layer 1
79 3 107 85 42
88 91 10 57 68
24 44 23 118 108
58 125 105 25 2
65 51 73 28 98
Layer 2
50 14 104 90 56
9 92 47 84 83
114 110 21 18 52
22 31 69 111 82
121 67 76 12 40
Layer 3
49 109 94 33 30
5 41 37 113 120
100 89 64 35 27
116 59 87 38 15
48 17 32 96 122
Layer 4
123 77 4 26 86
117 29 101 60 8
6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

	
Objective Function	Objective Function: 244 
Durasi	Durasi: 1.31240304 seconds 

c. Eksperimen 3

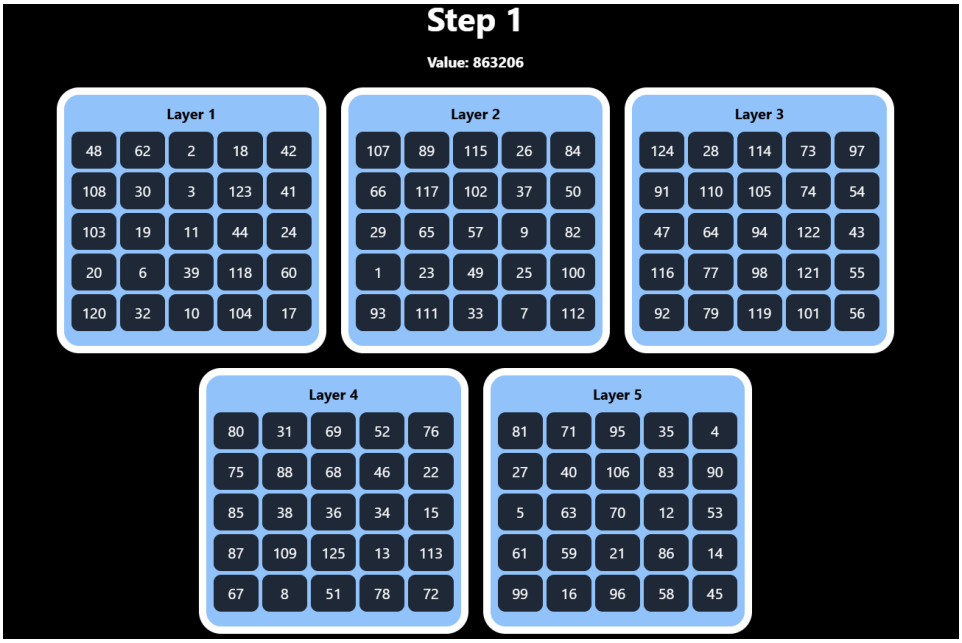
State Awal	
	

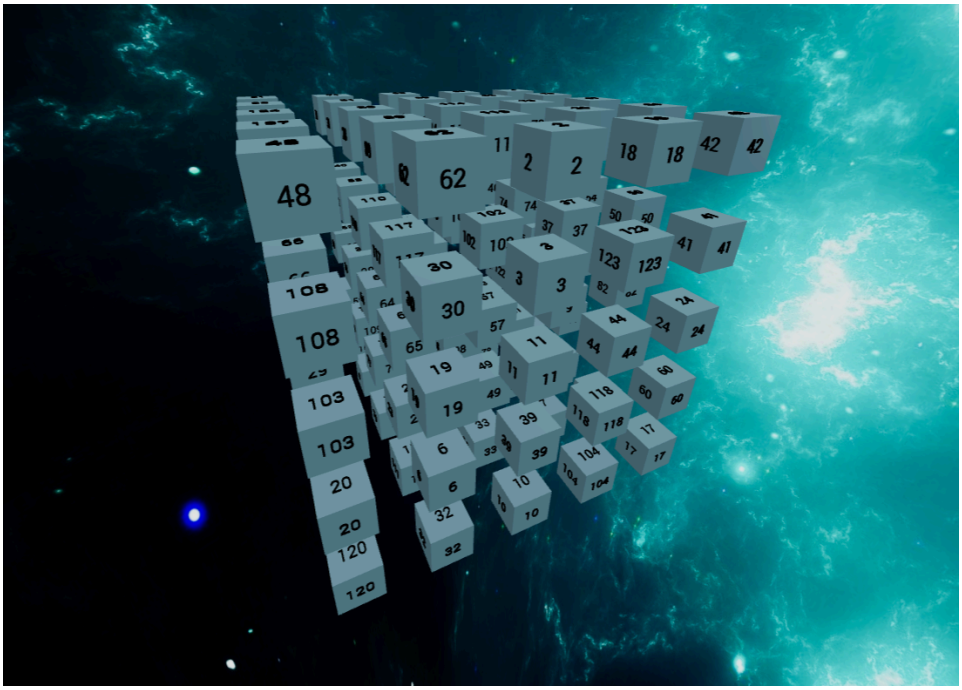
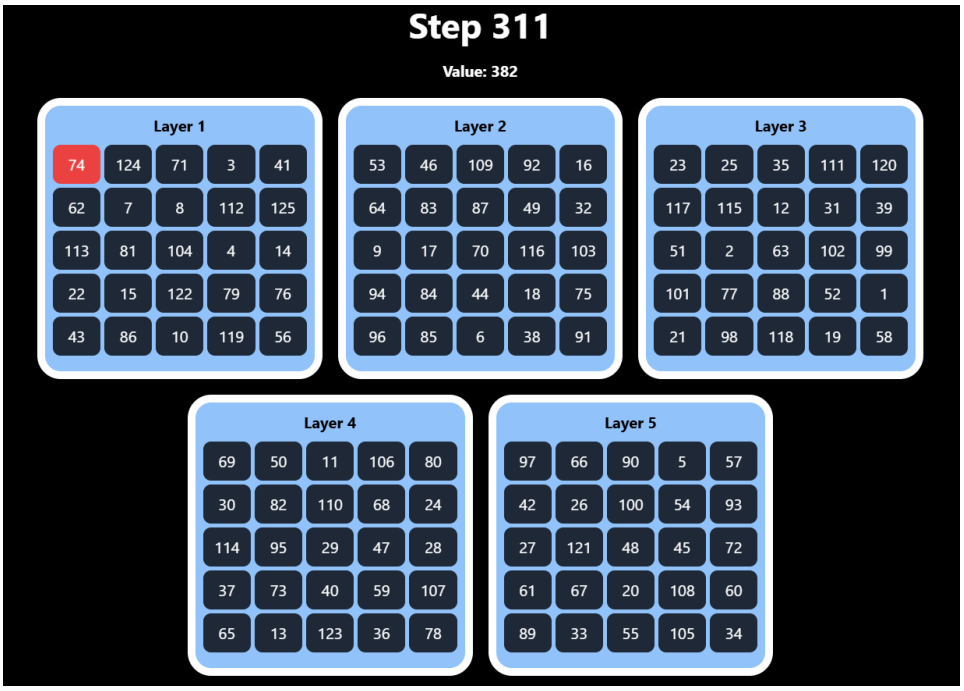
State Akhir	 
Objective Function	Objective Function: 479

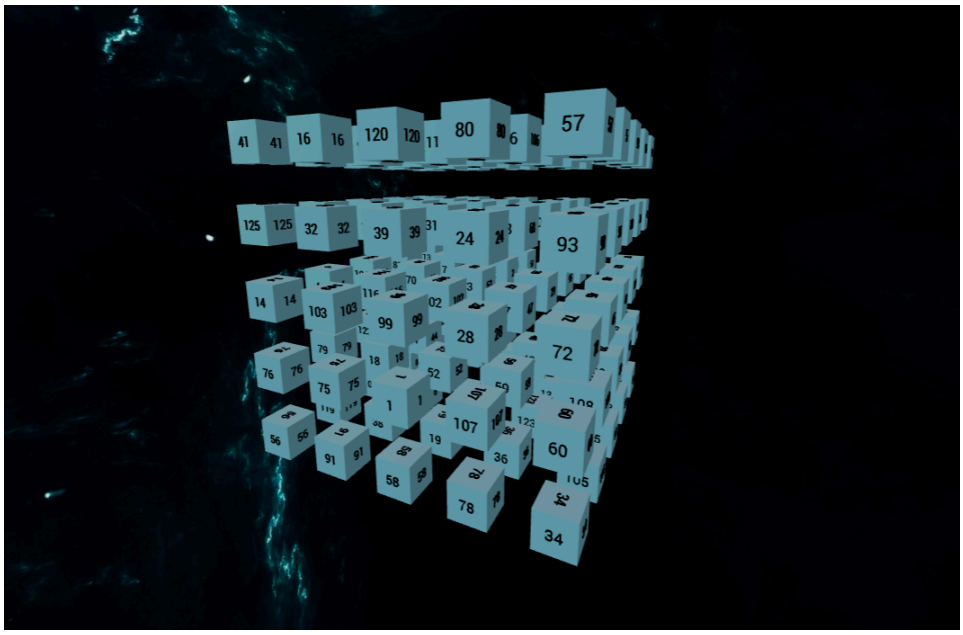
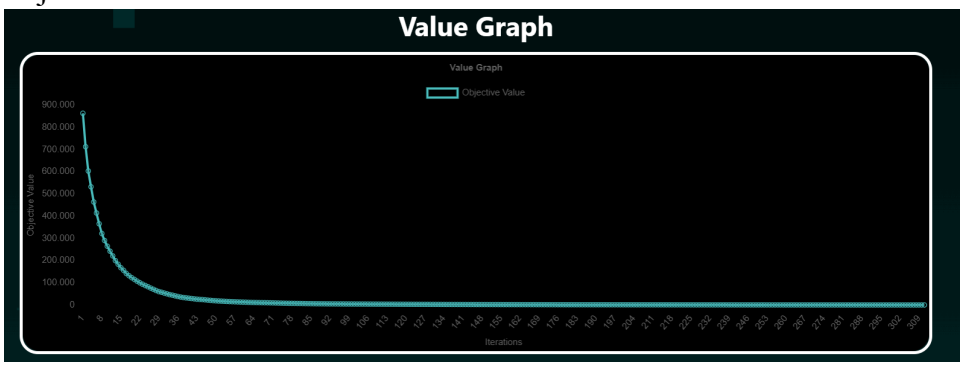
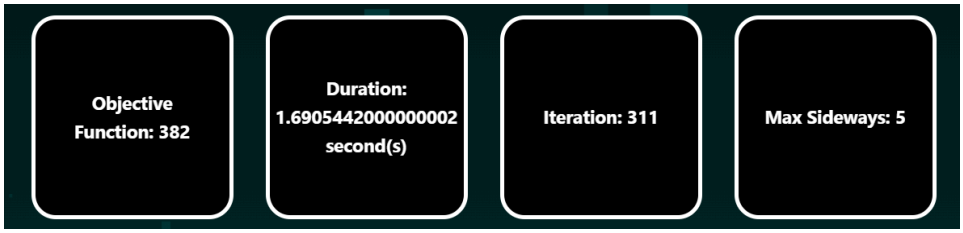
	
Durasi	Durasi: 1.5249815 seconds 

II. Sideways Move Hill-Climbing

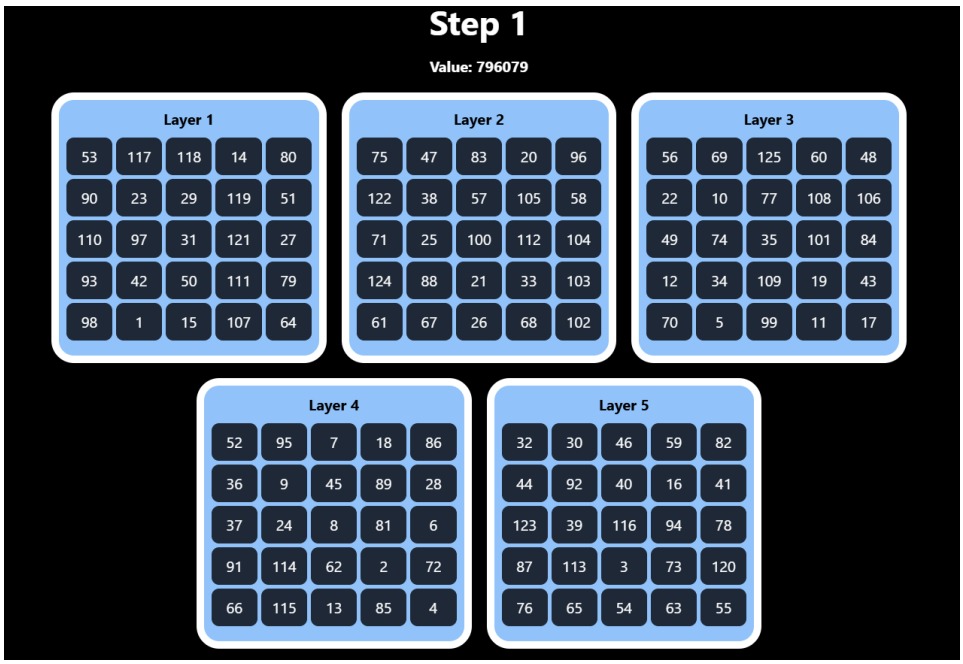
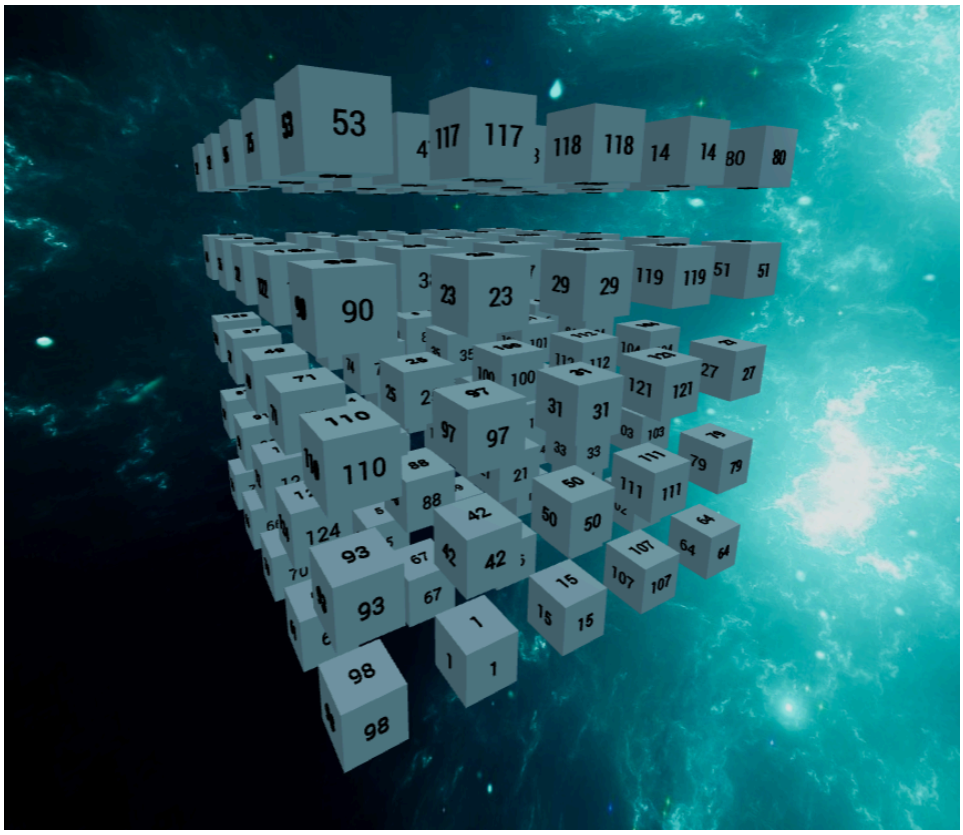
a. Eksperimen 1

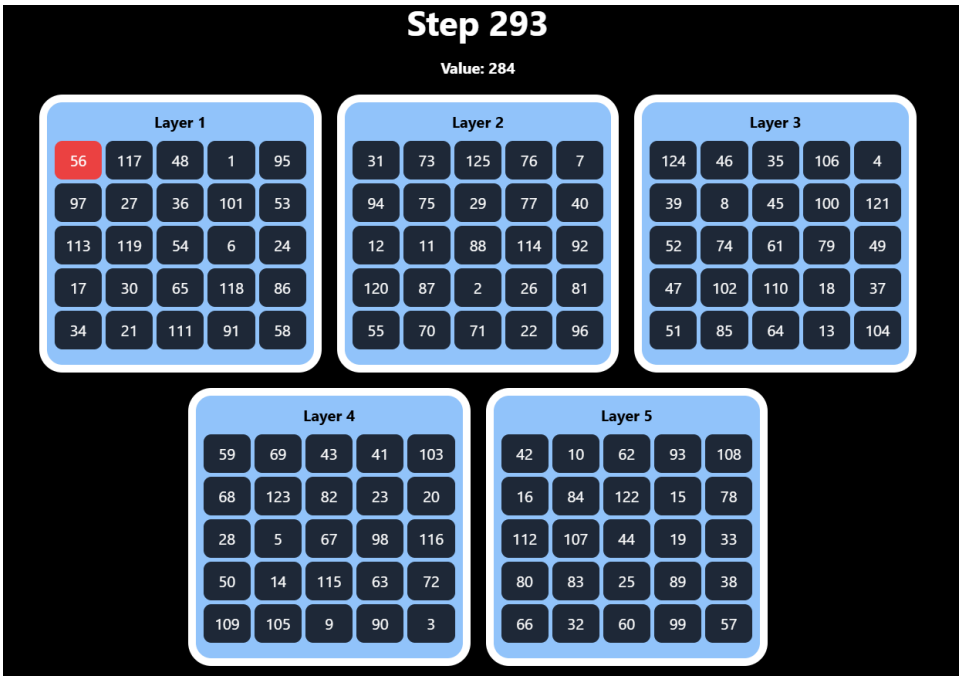
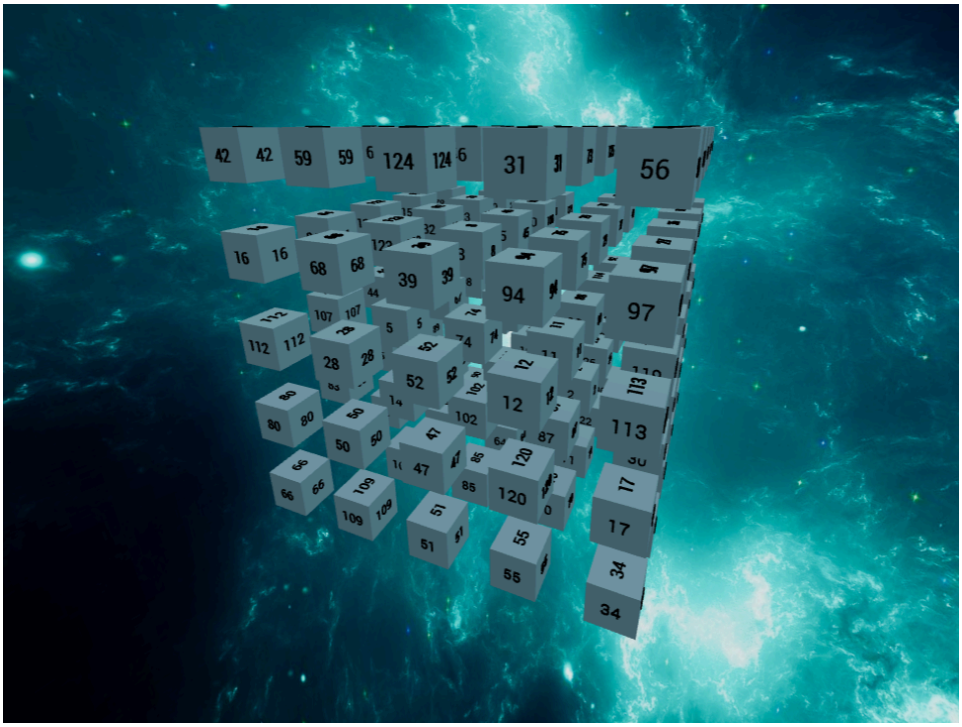
State Awal	

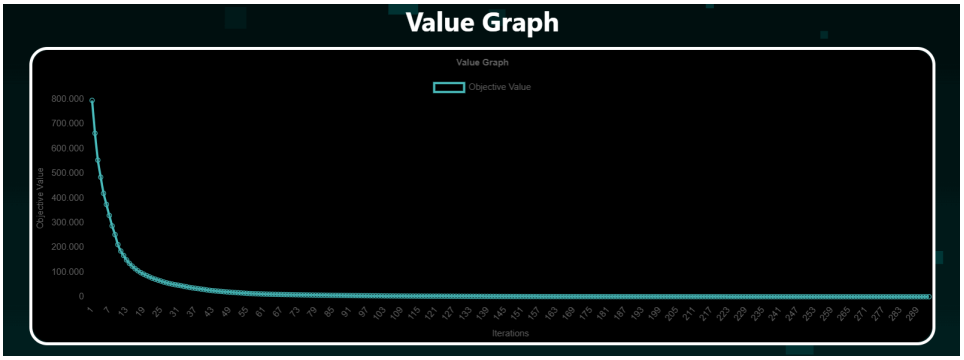
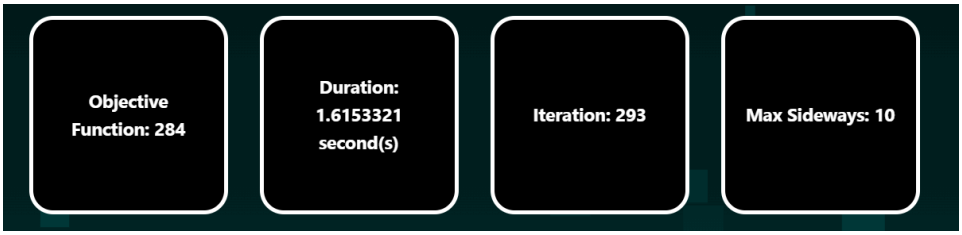
	
State Akhir	 Step 311 Value: 382 Layer 1 74 124 71 3 41 62 7 8 112 125 113 81 104 4 14 22 15 122 79 76 43 86 10 119 56 Layer 2 53 46 109 92 16 64 83 87 49 32 9 17 70 116 103 94 84 44 18 75 96 85 6 38 91 Layer 3 23 25 35 111 120 117 115 12 31 39 51 2 63 102 99 101 77 88 52 1 21 98 118 19 58 Layer 4 69 50 11 106 80 30 82 110 68 24 114 95 29 47 28 37 73 40 59 107 65 13 123 36 78 Layer 5 97 66 90 5 57 42 26 100 54 93 27 121 48 45 72 61 67 20 108 60 89 33 55 105 34

	
Objective Function	Objective Function: 382 
Durasi	Durasi: 1.69054 seconds 

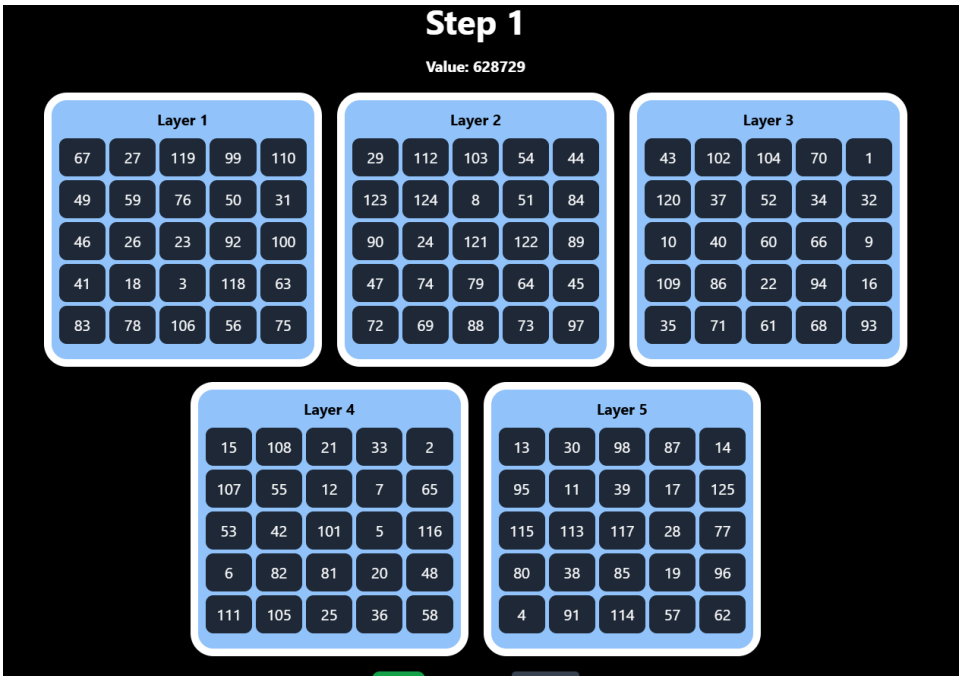
b. Eksperimen 2

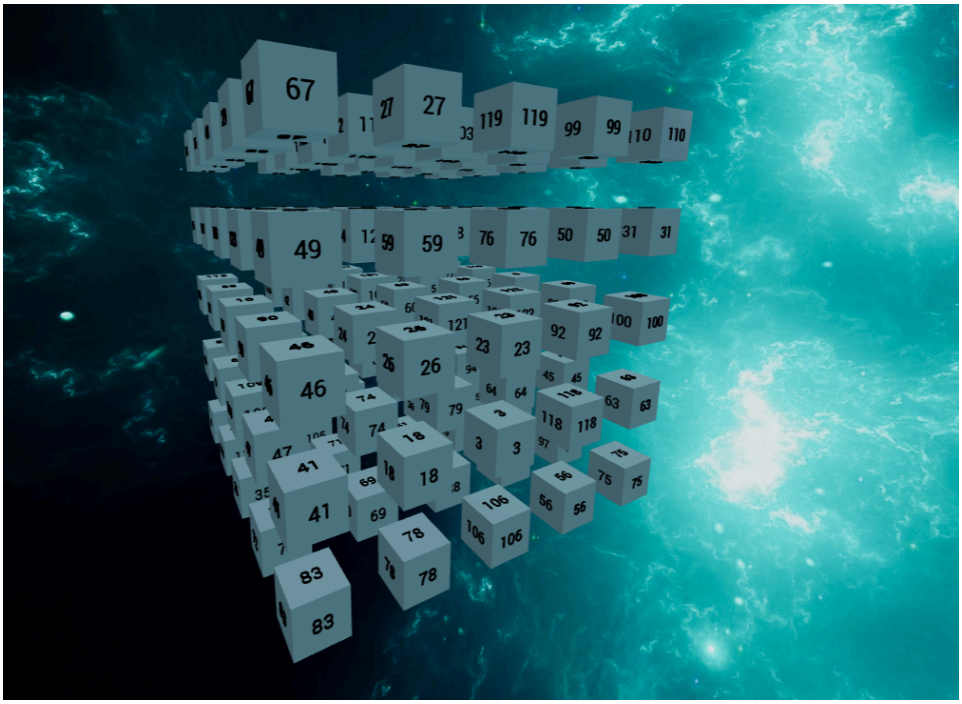
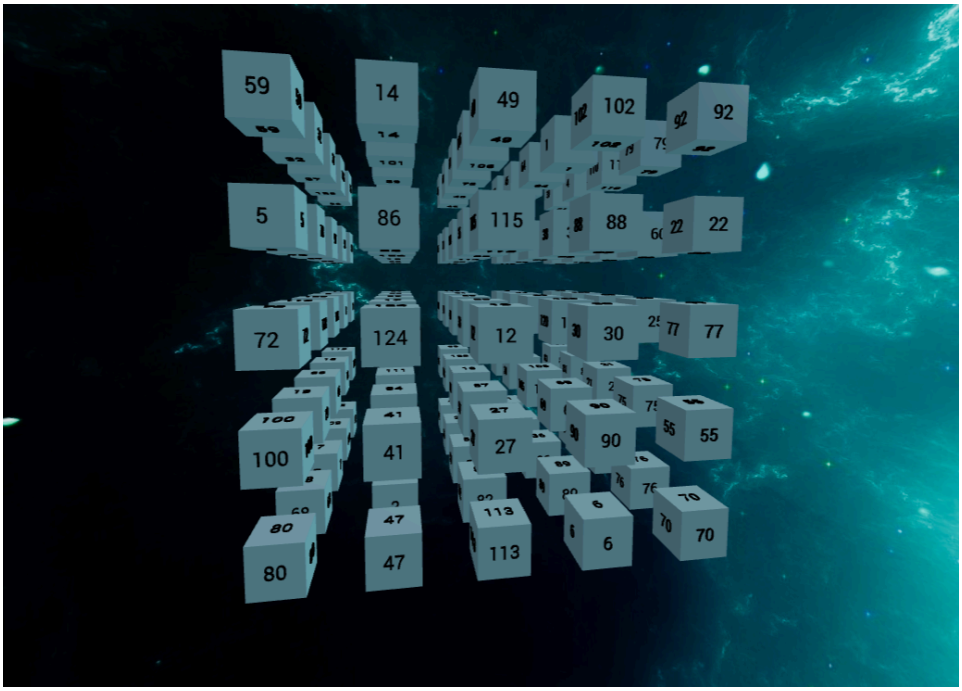
State Awal	 

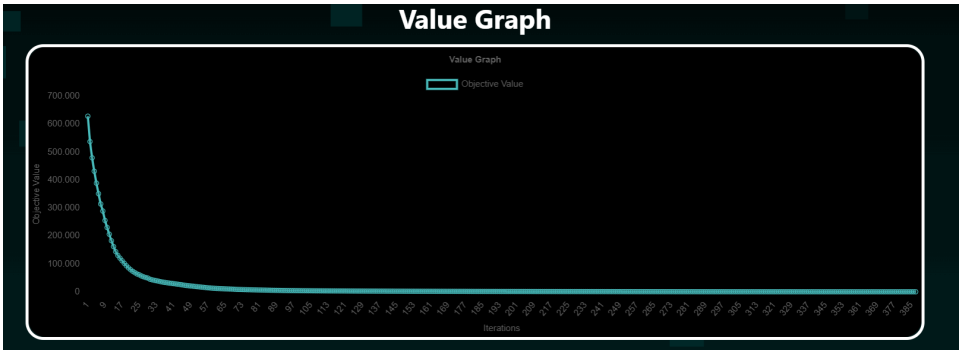
State Akhir	 
Objective Function	Objective Function: 284

	
Durasi	Durasi: 1.61533 seconds 

c. Eksperimen 3

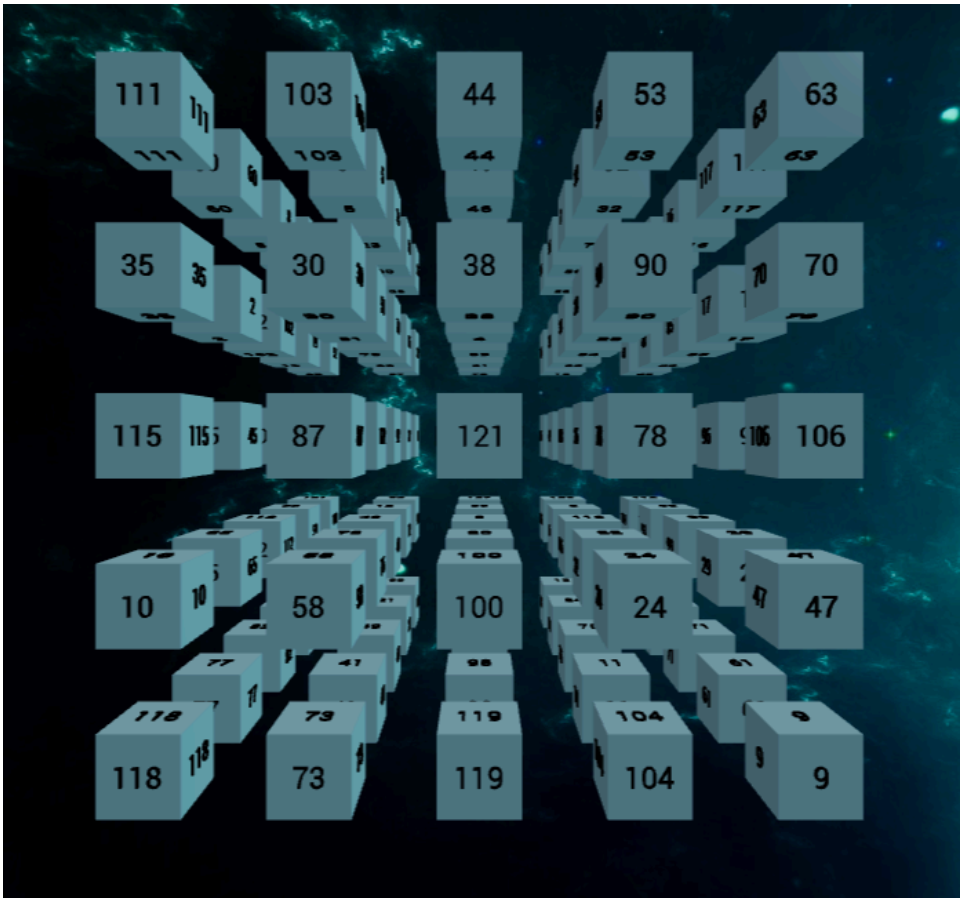
State Awal	

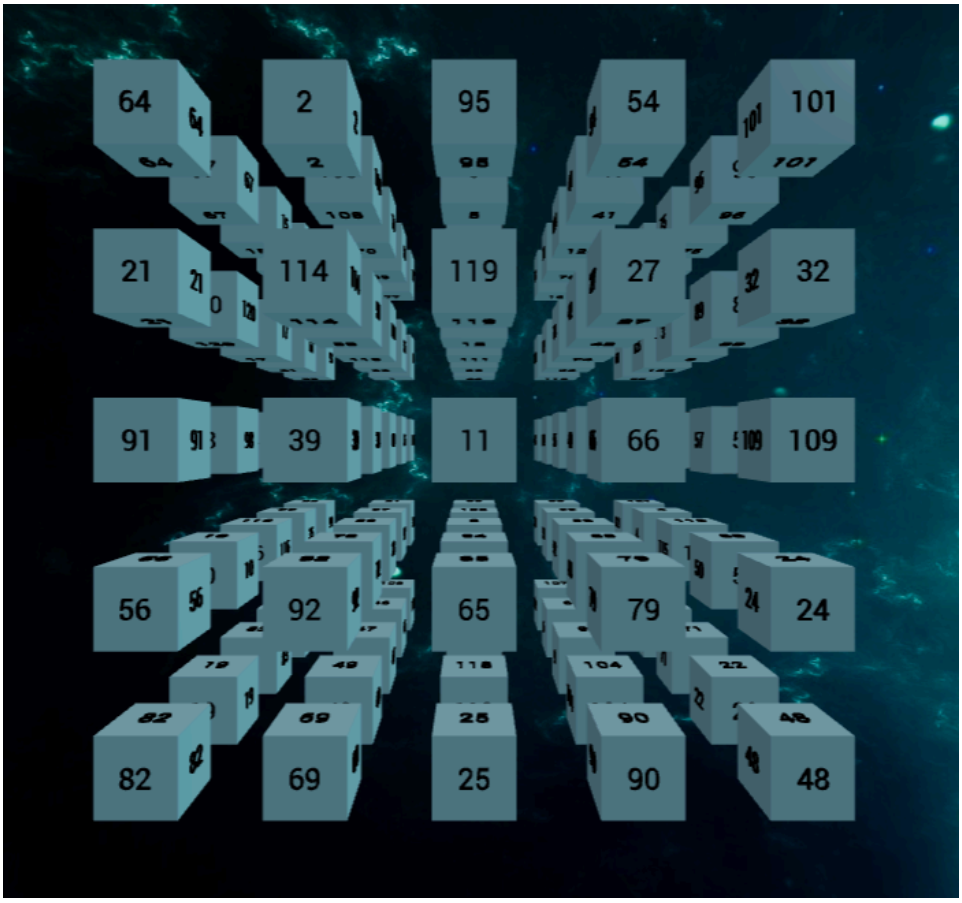
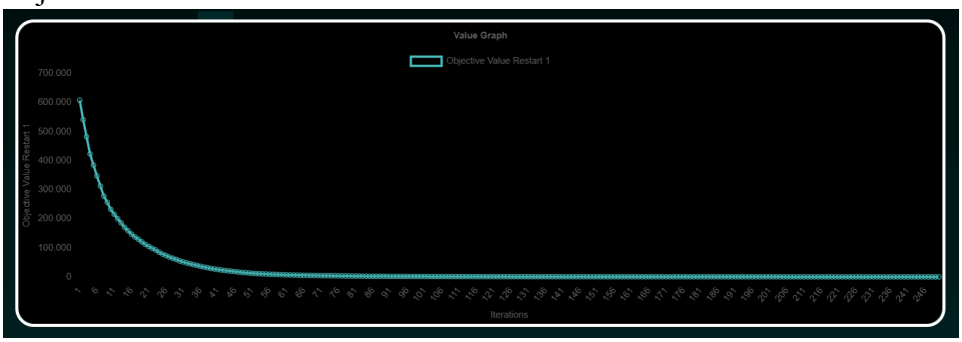
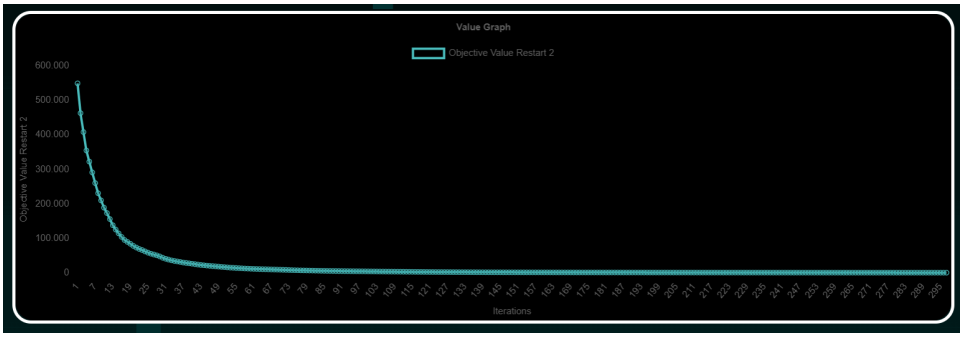
	
State Akhir	
	
Objective Function	Objective Function: 470

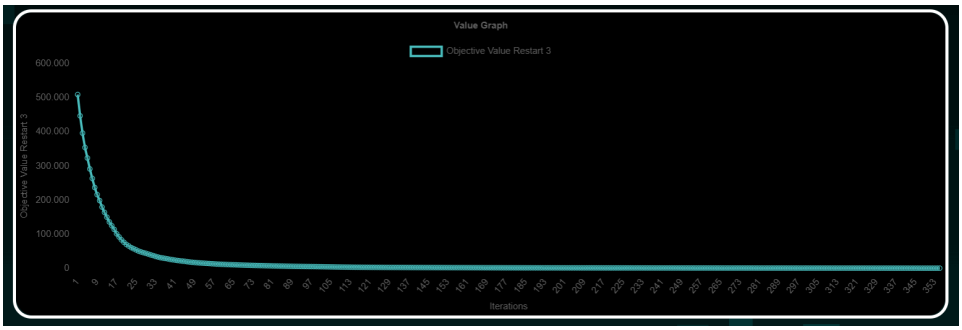
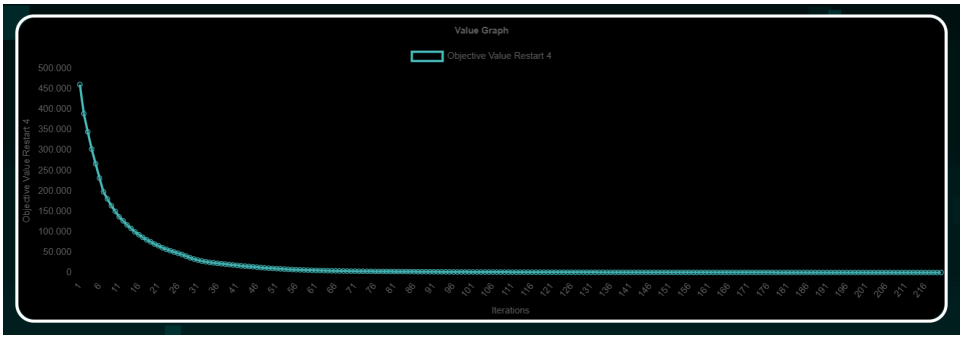
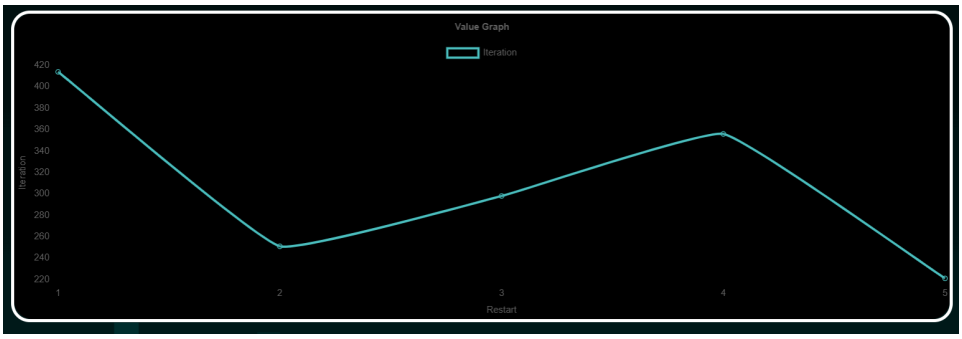
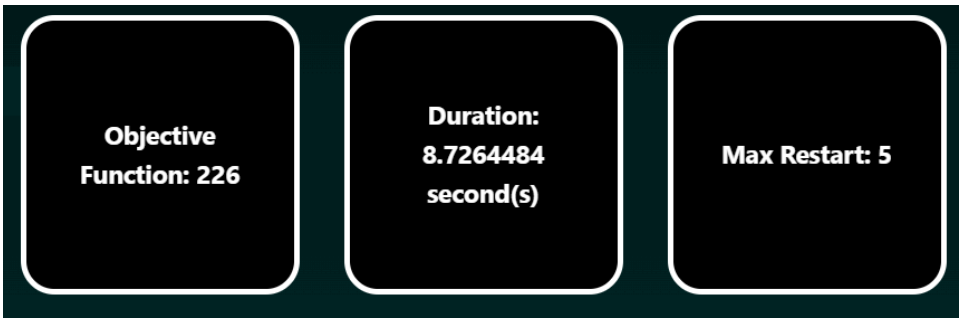
	
Durasi	Durasi: 2.111972 seconds

III. Random Restart Hill Climbing

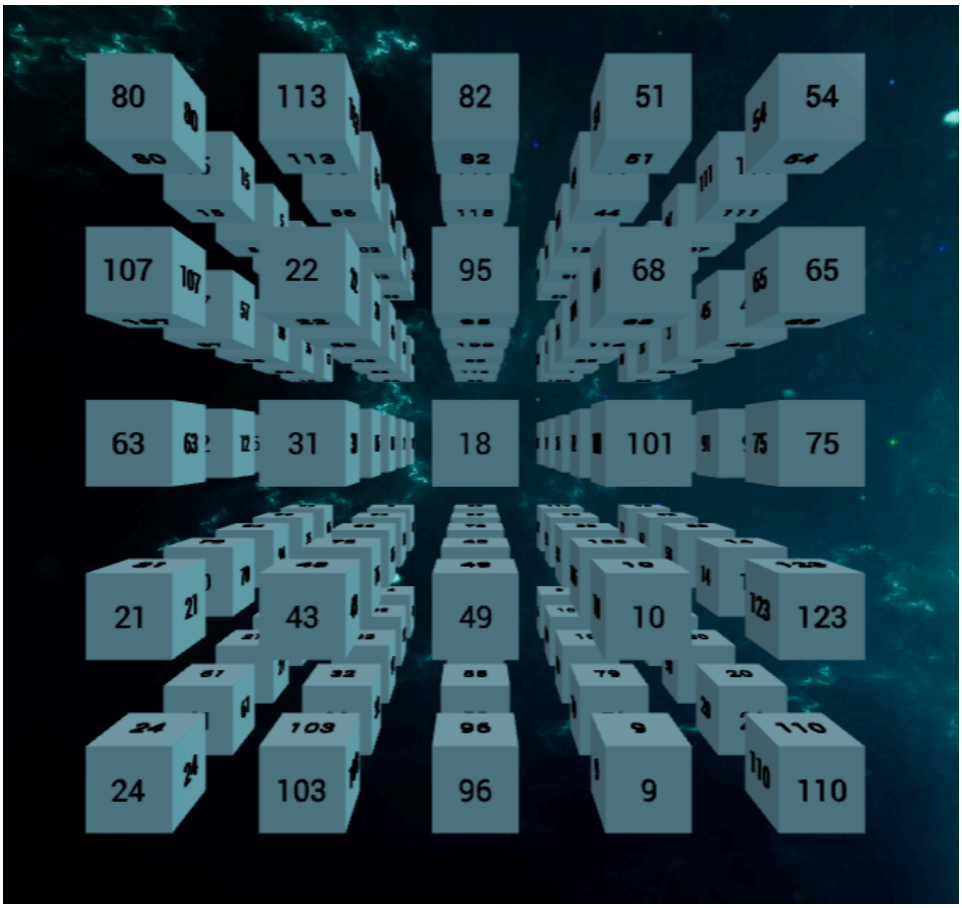
a. Eksperimen 1

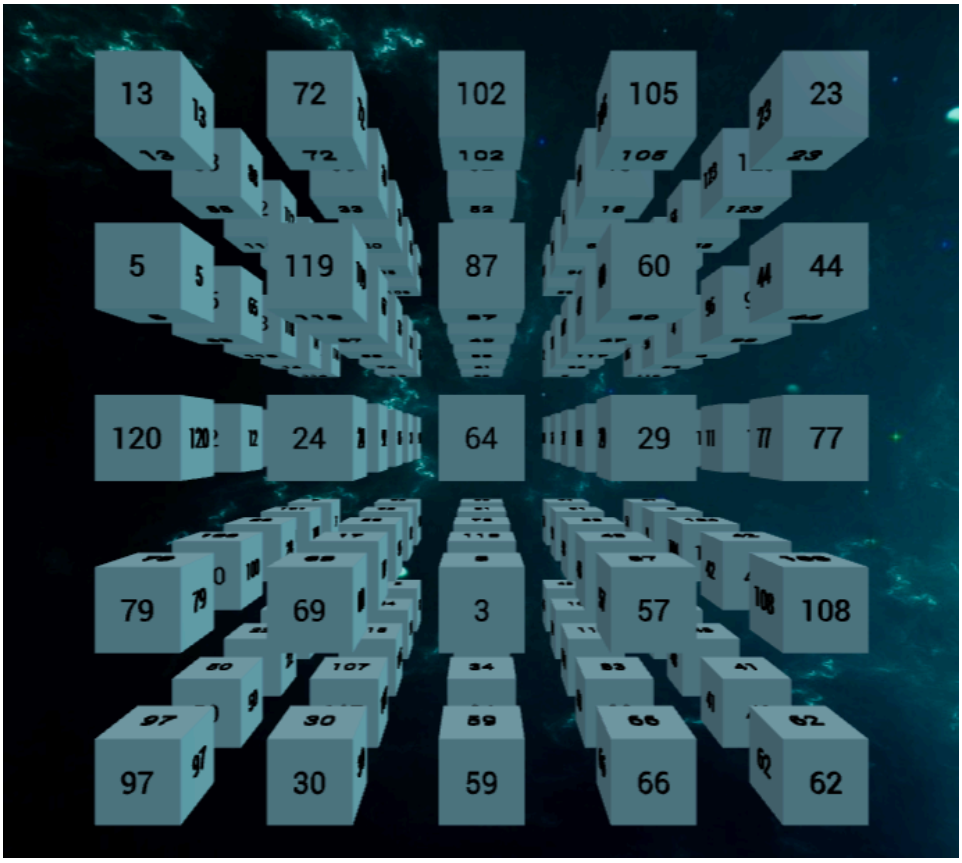
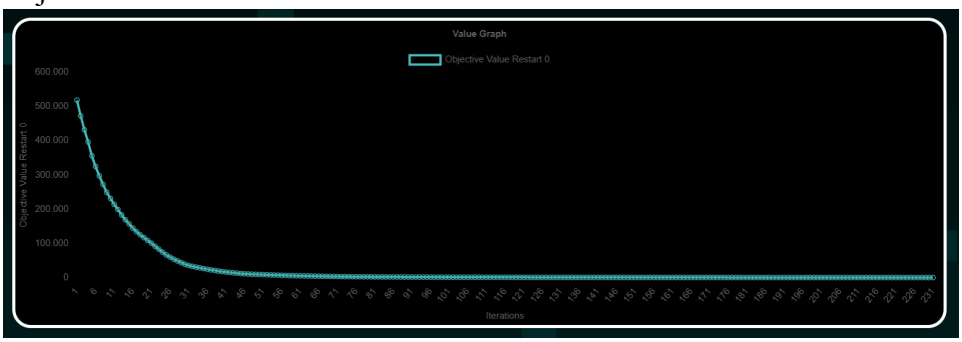
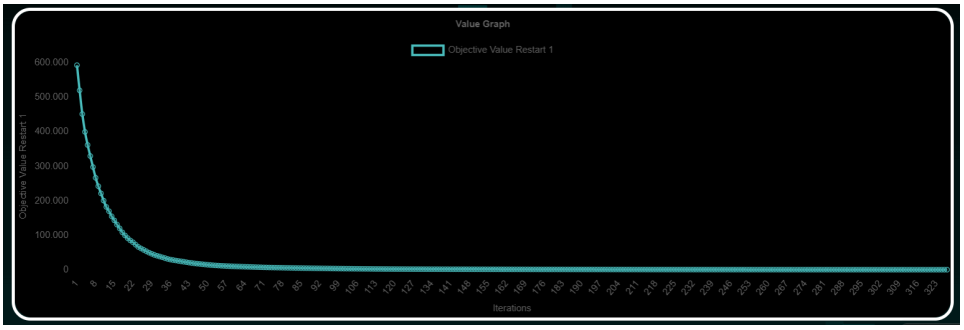
State Awal	 111 103 44 53 63 35 30 38 90 70 115 87 121 78 106 10 58 100 24 47 118 73 119 104 9

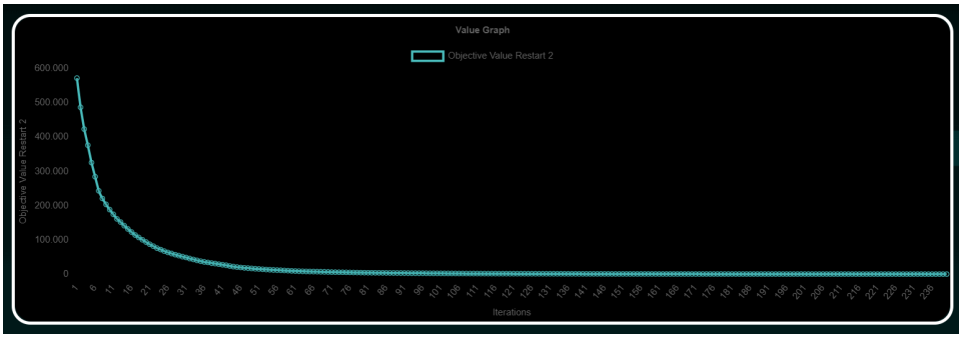
State Akhir	
Objective Function	Objective Function: 226  

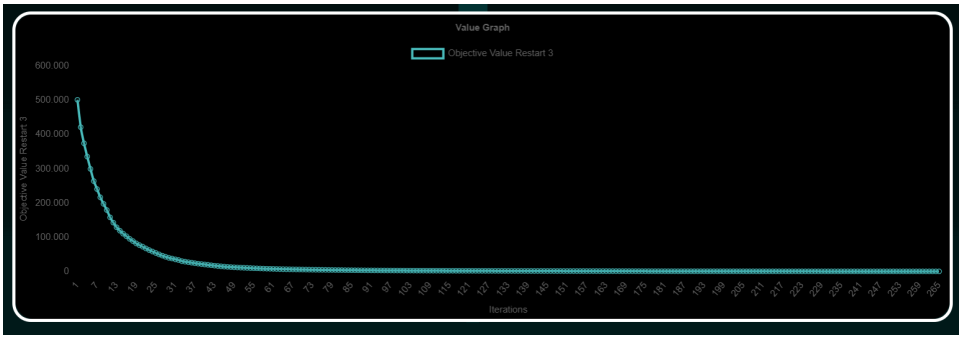
	  
Durasi	Durasi: 8.7264484 seconds 

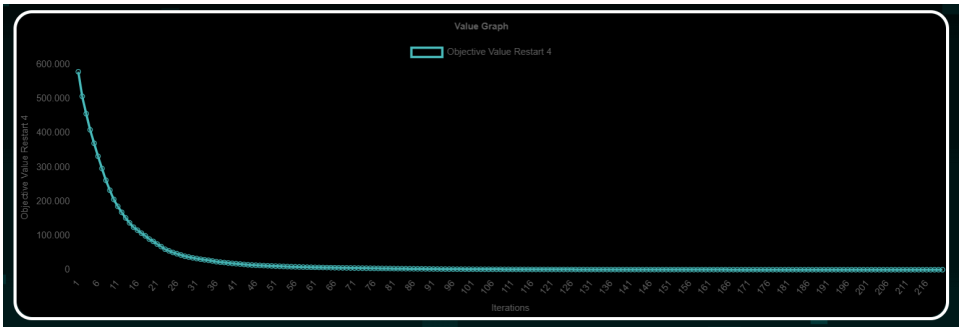
b. Eksperimen 2

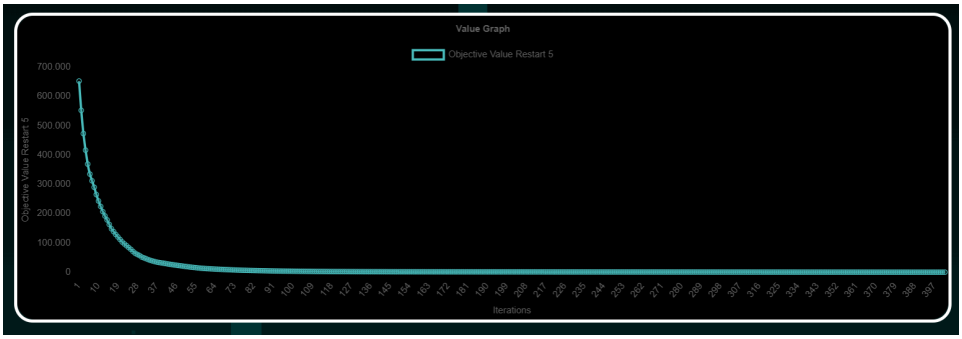
State Awal	 80 113 82 51 54 107 22 95 68 65 63 31 18 101 75 21 43 49 10 123 24 103 96 9 110

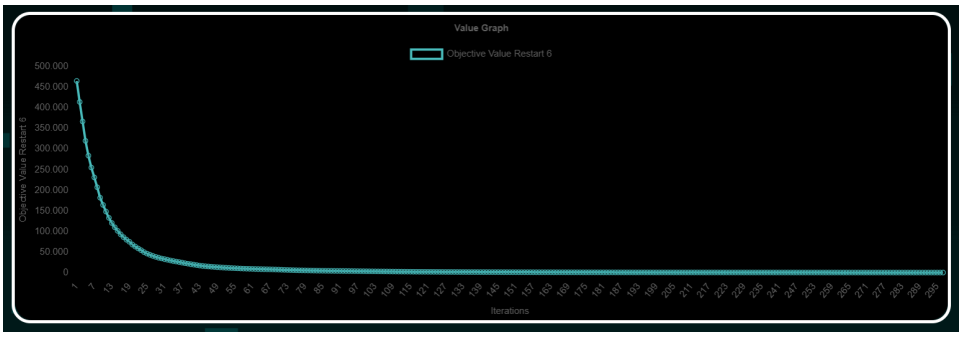
State Akhir	
Objective Function	Objective Function: 180  

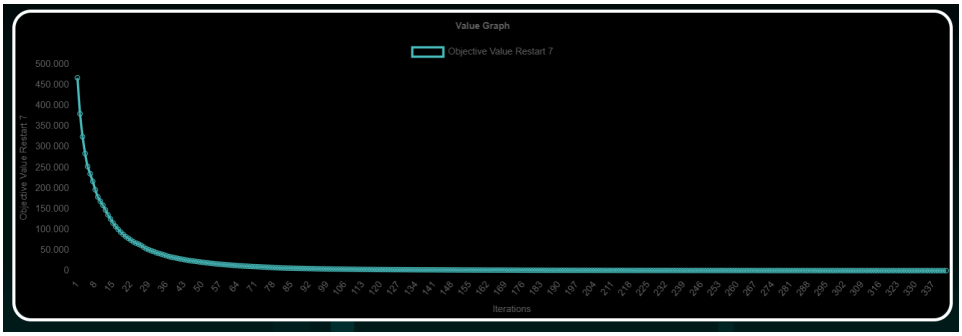
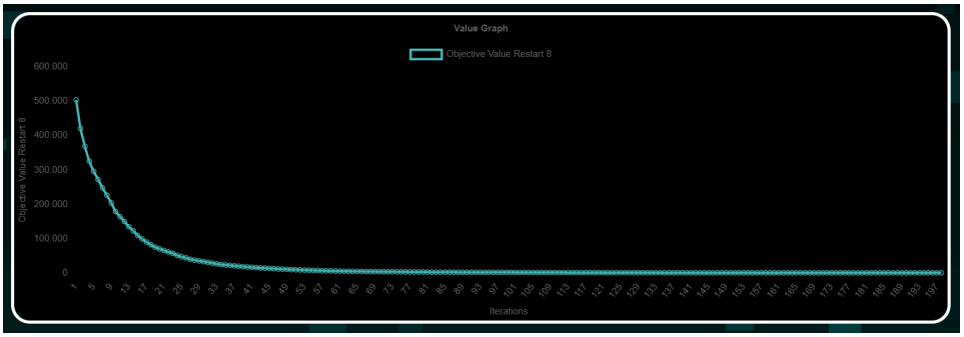
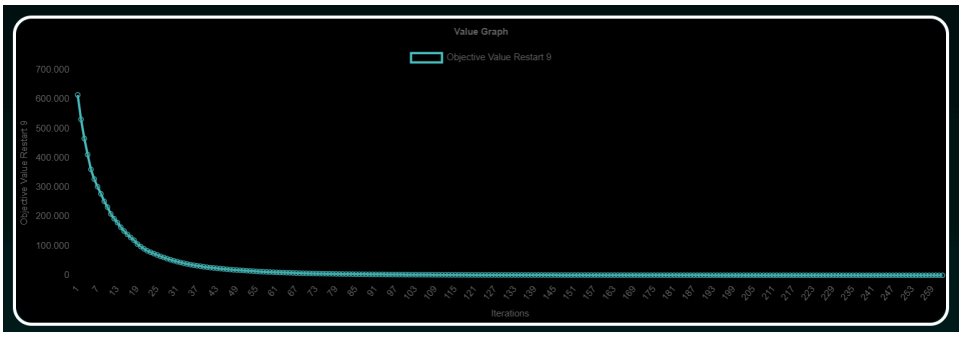
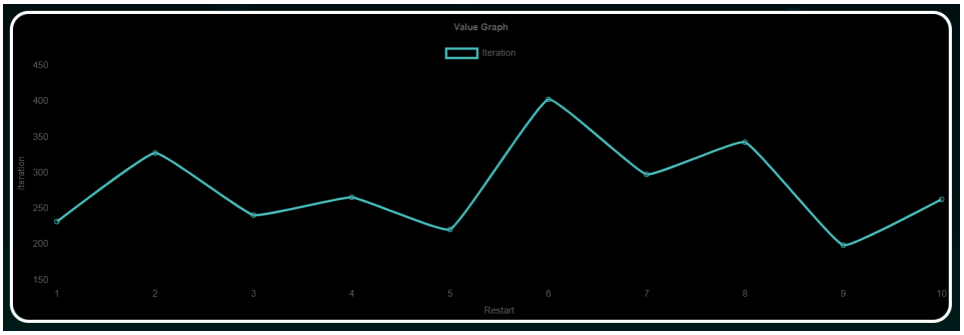

Value Graph
Objective Value Restart 2
Objective Value Restart 2
Iterations

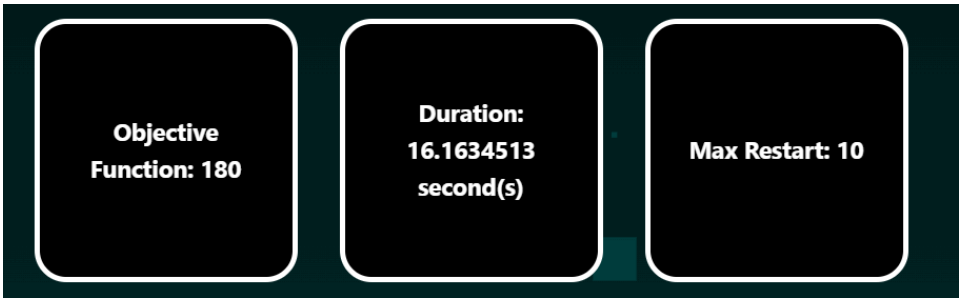

Value Graph
Objective Value Restart 3
Objective Value Restart 3
Iterations


Value Graph
Objective Value Restart 4
Objective Value Restart 4
Iterations

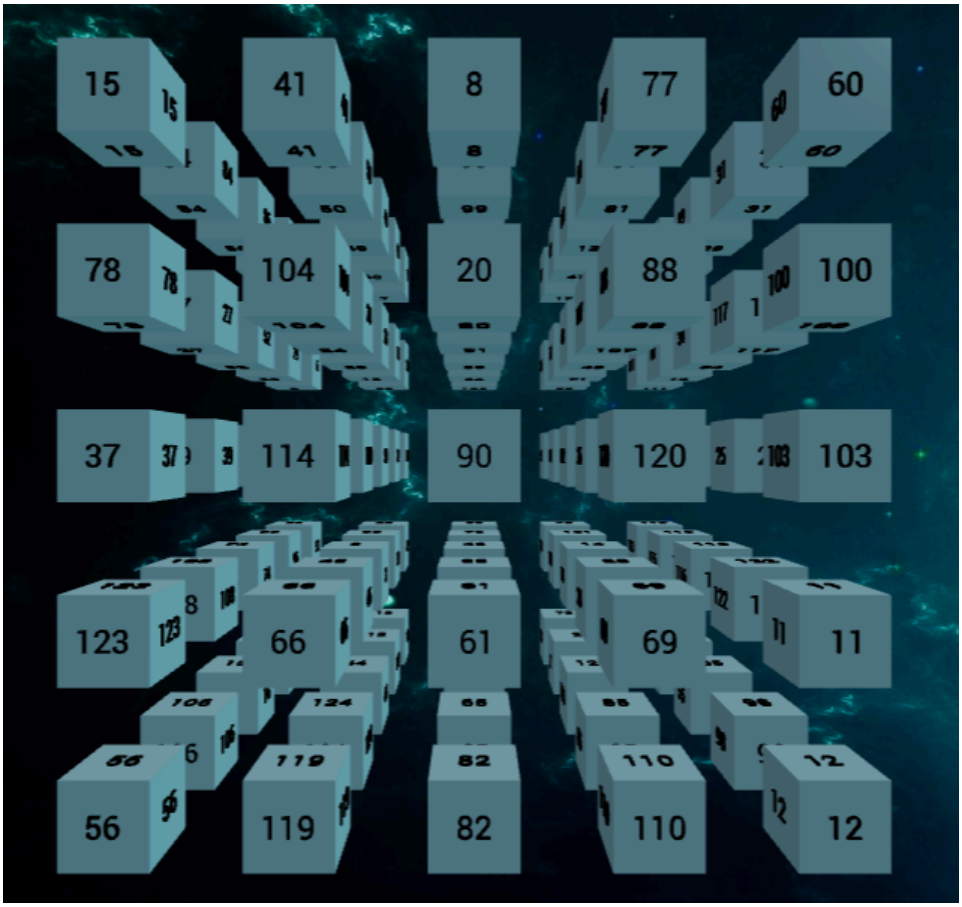

Value Graph
Objective Value Restart 5
Objective Value Restart 5
Iterations

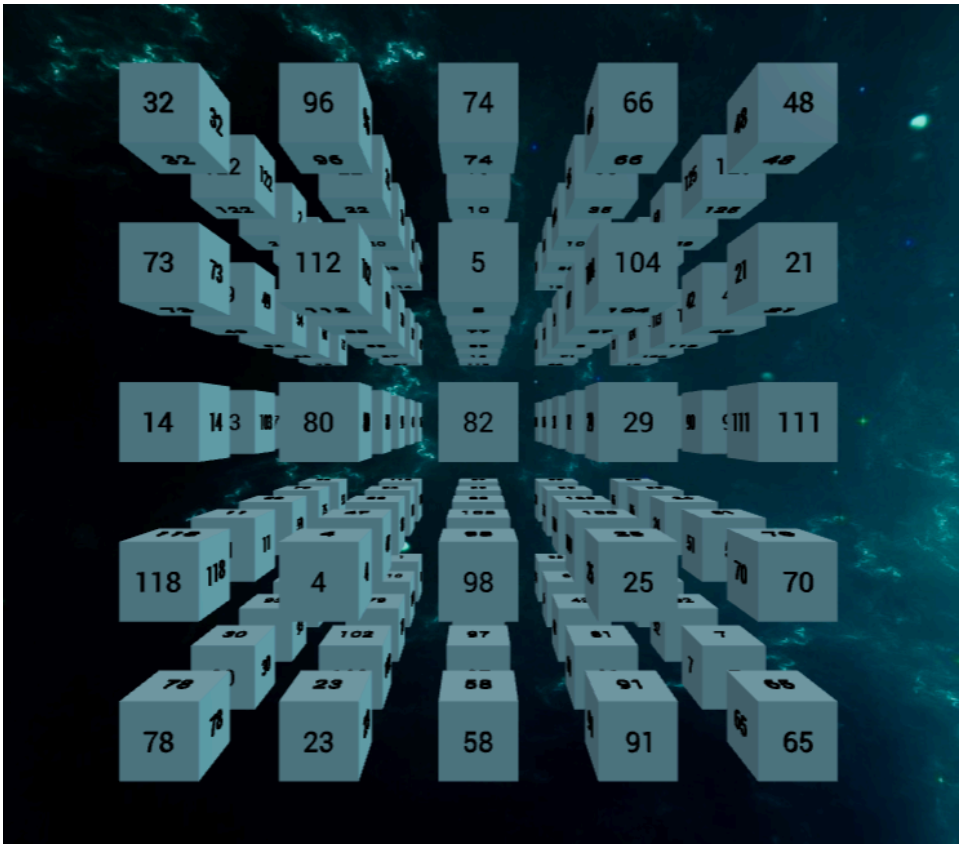
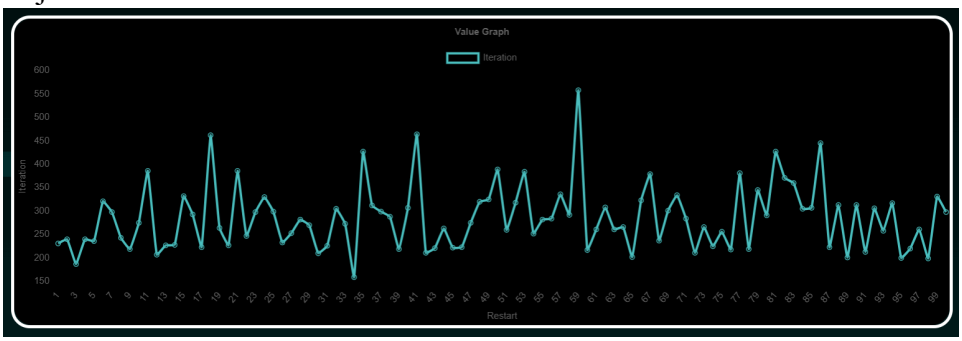
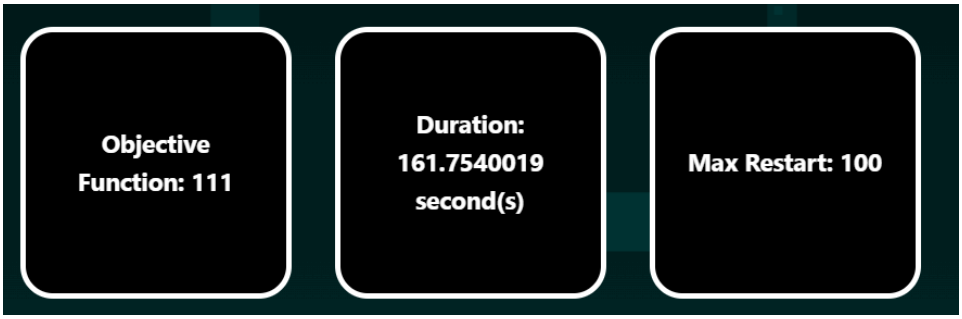

Value Graph
Objective Value Restart 6
Objective Value Restart 6
Iterations

	   
Durasi	Durasi: 16.1634513 seconds

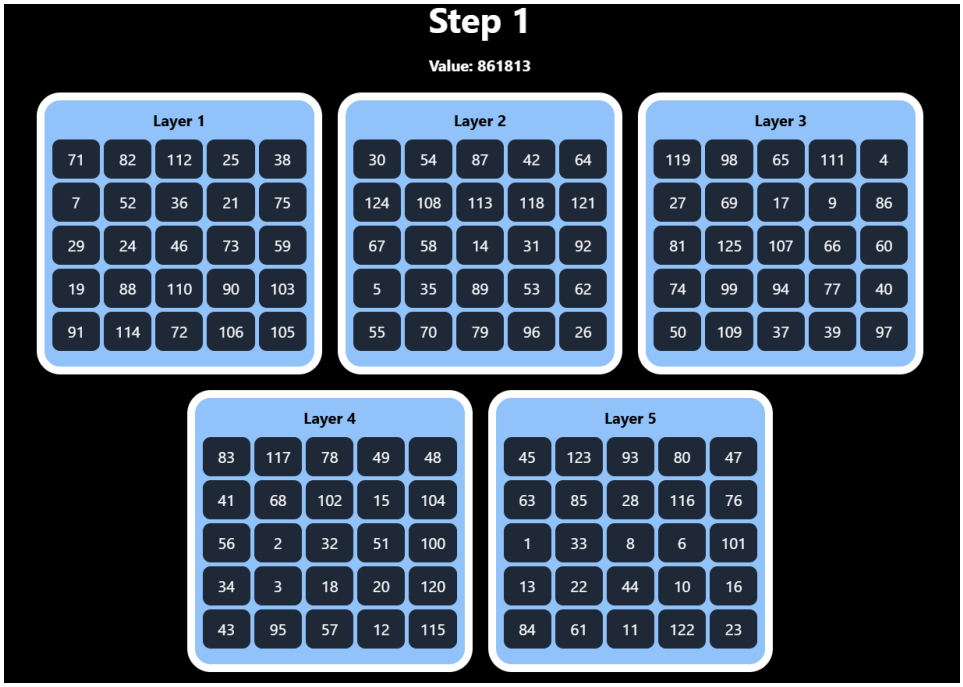
c. Eksperimen 3

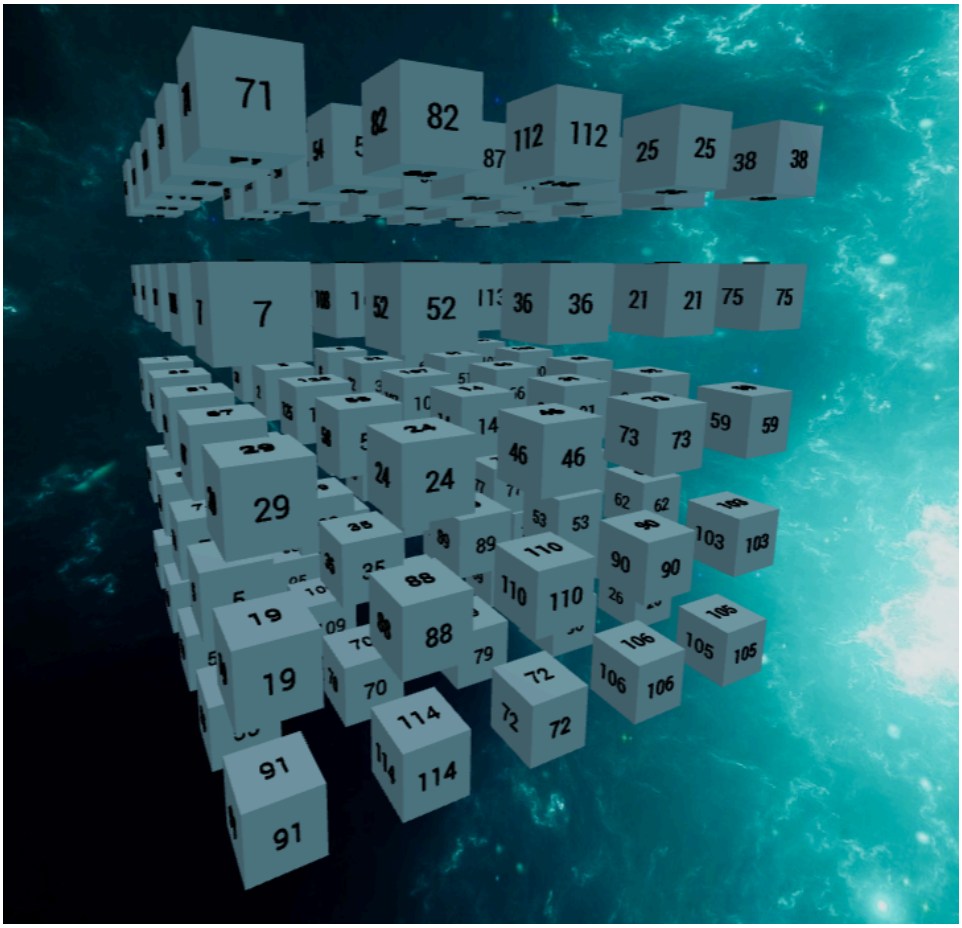
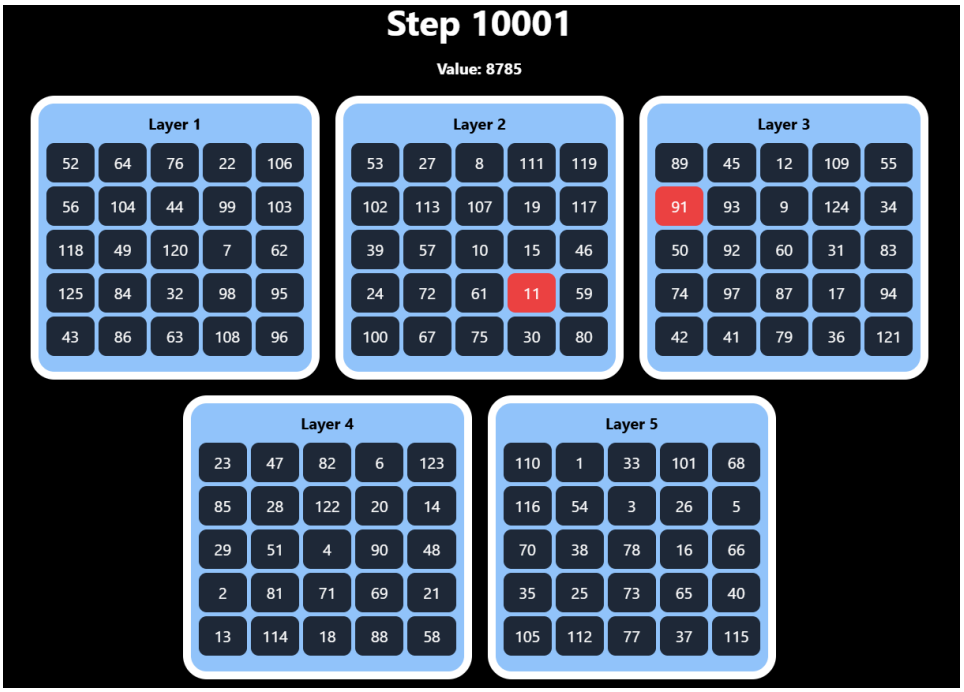
State Awal	

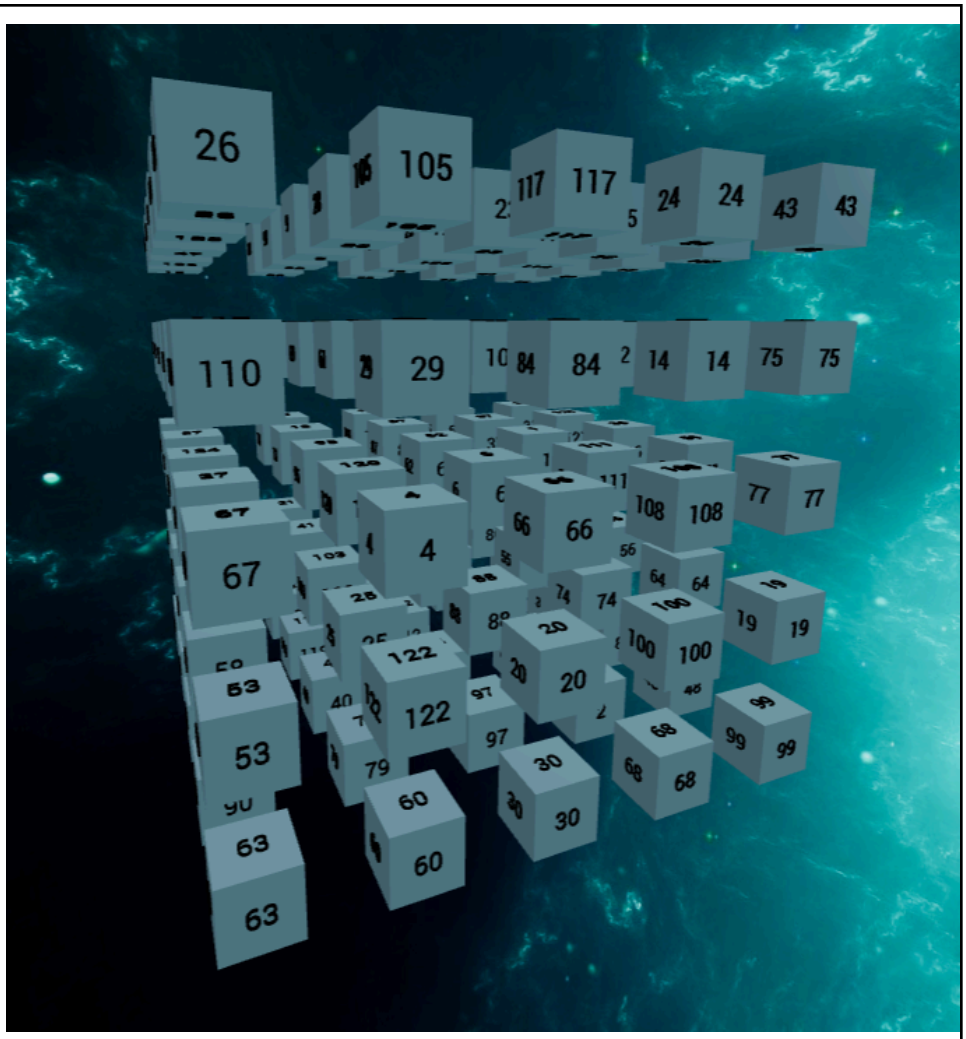
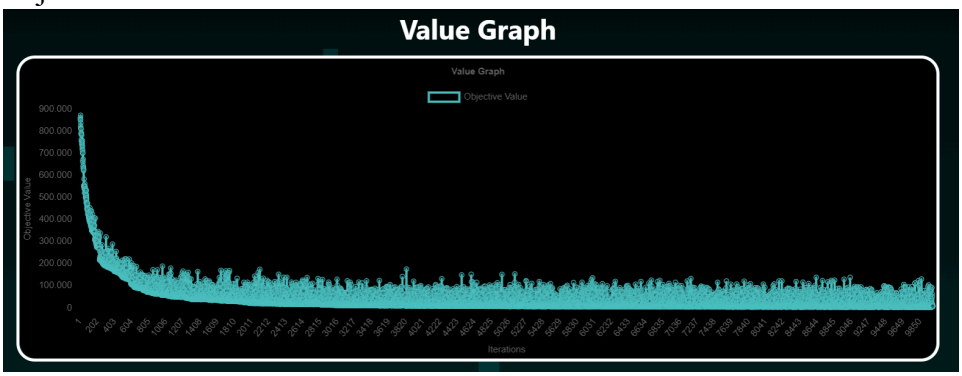
State Akhir	
Objective Function	Objective Function: 111 
Durasi	Durasi: 161.7540019 seconds 

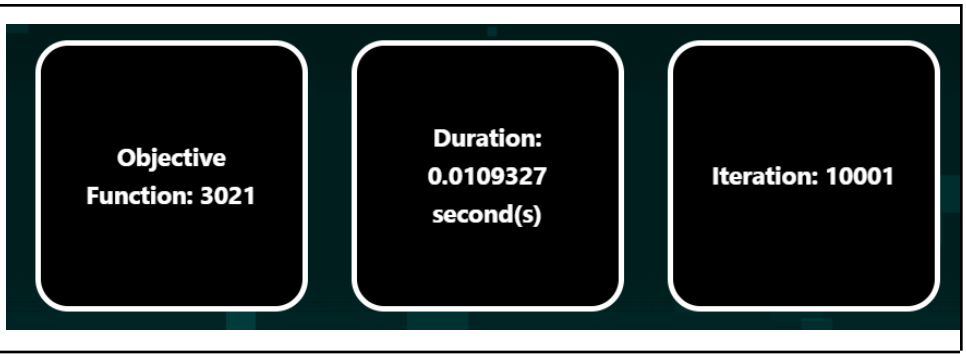
IV. Stochastic Hill-Climbing

a. Eksperimen 1

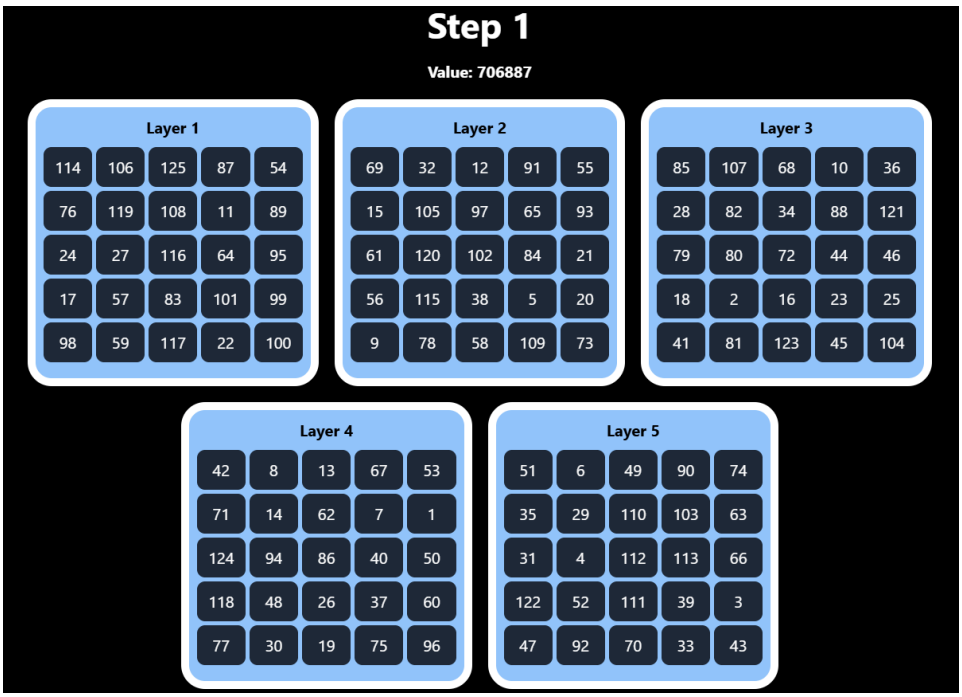
State Awal	

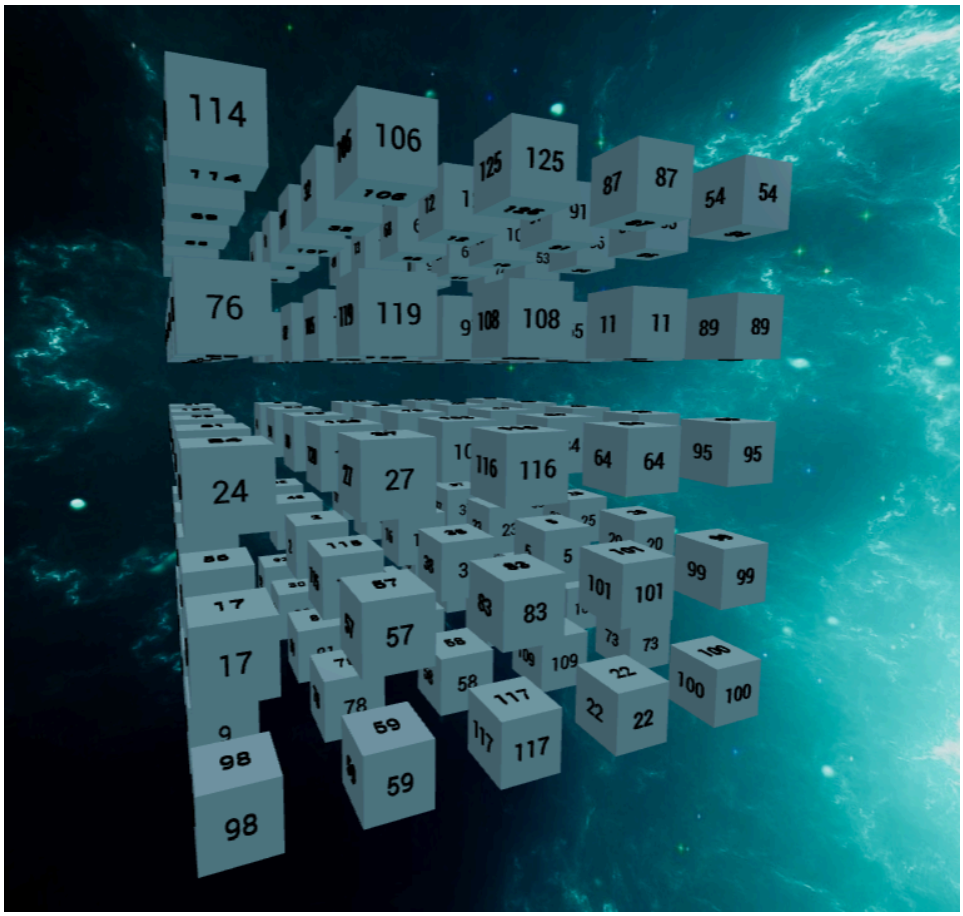
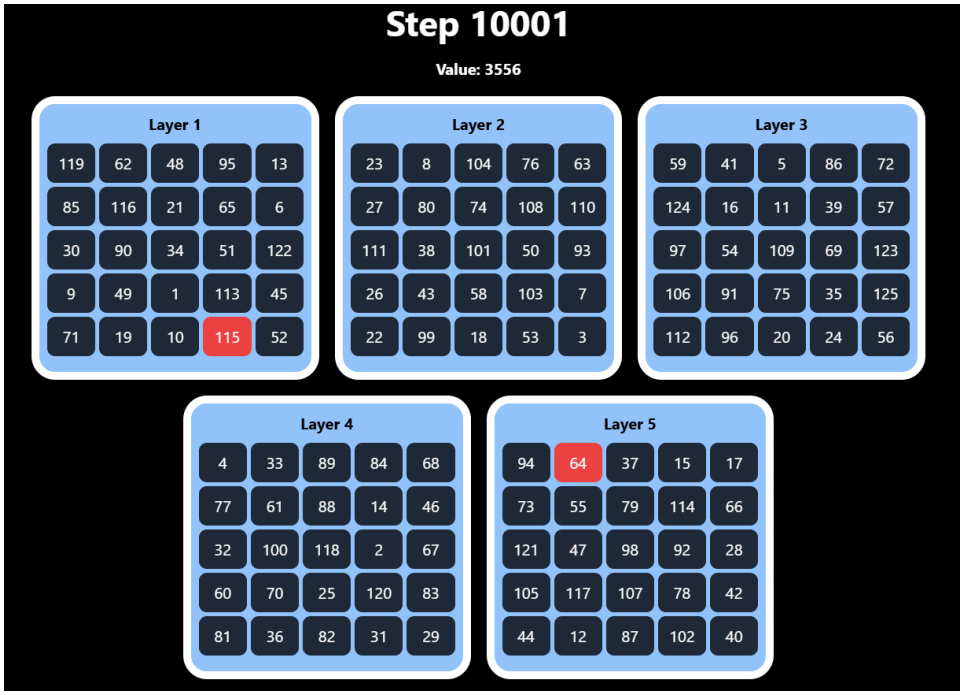
<table>
<tr>
<td></td>
<td>  </td>
</tr>
<tr>
<td>State Akhir</td>
<td> 
</td>
</tr>
</table>

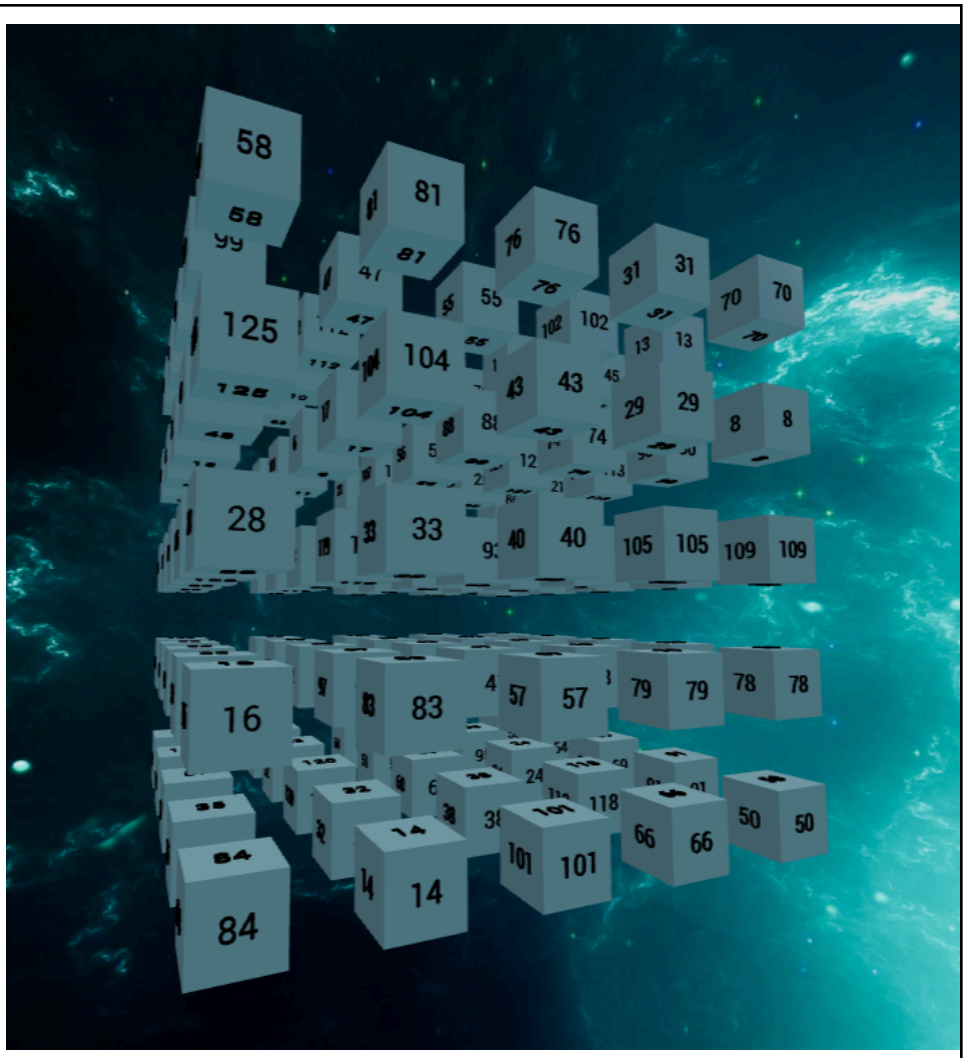
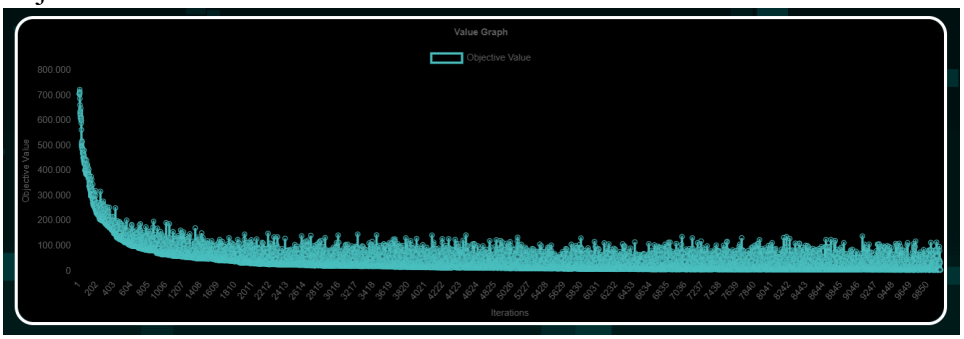
	
Objective Function	Objective Function: 3021 
Durasi	Durasi: 0.0109327 seconds

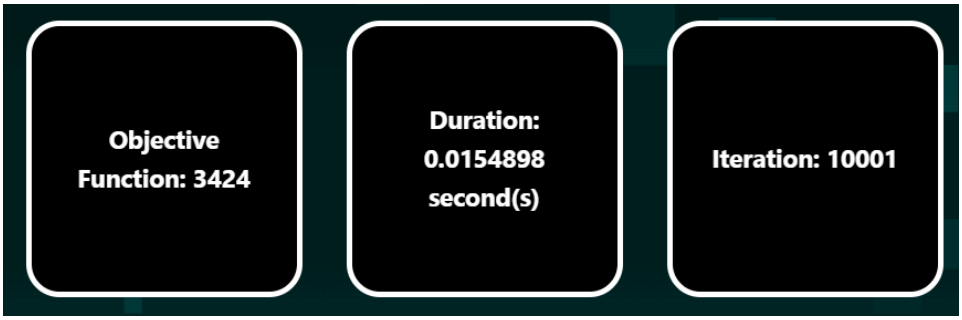
	

b. Eksperimen 2

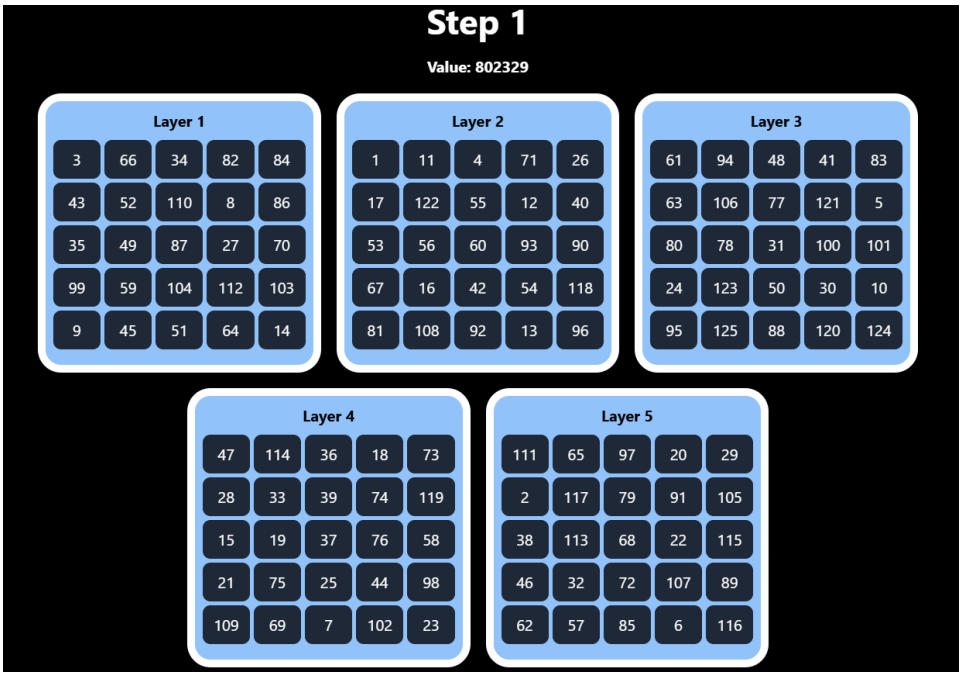
State Awal	

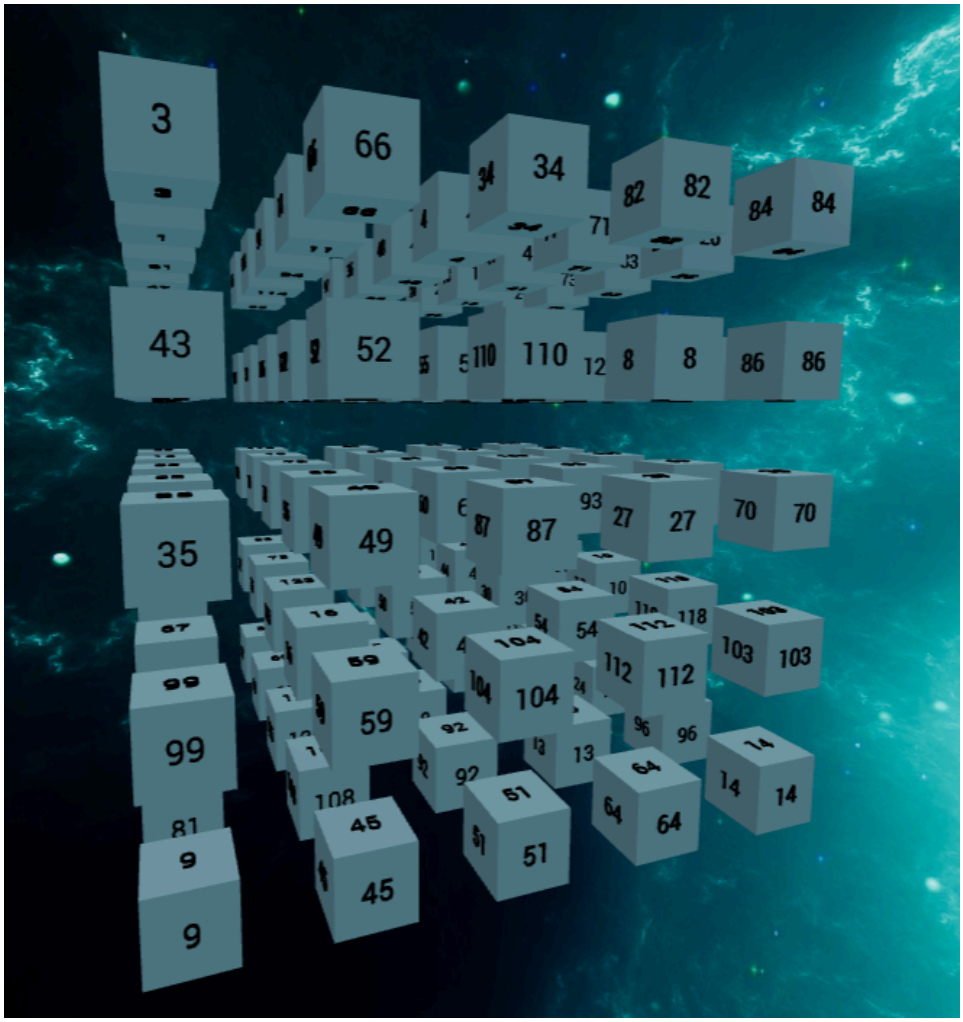
<table>
<tr>
<td></td>
<td> 
</td>
</tr>
<tr>
<td>State Akhir</td>
<td> 
</td>
</tr>
</table>

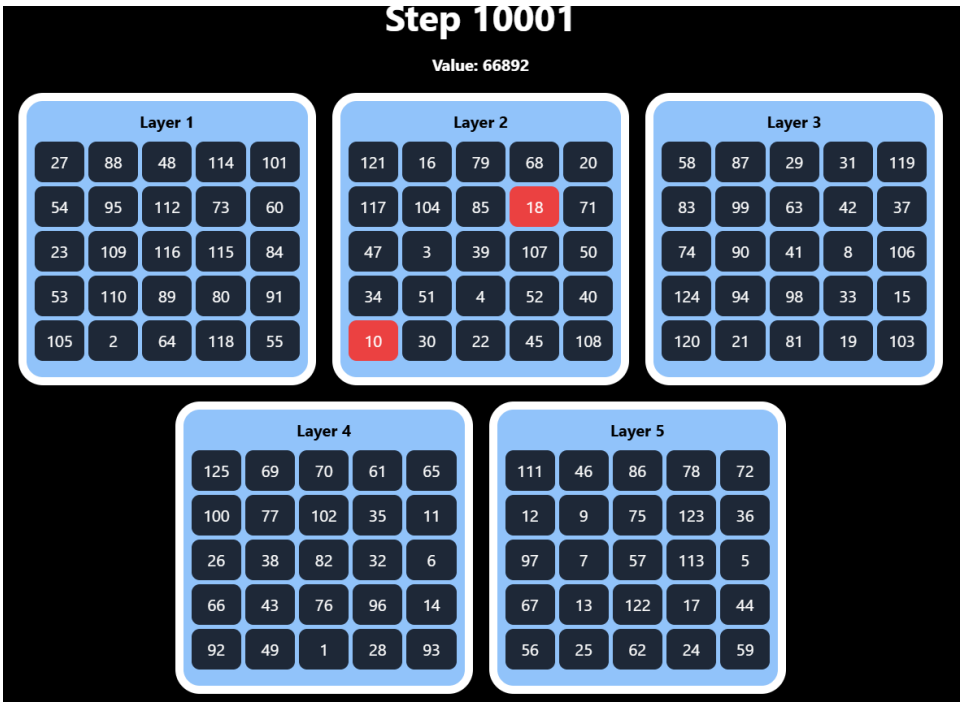
	
Objective Function	Objective Function: 3424 
Durasi	Durasi: 0.0154898 seconds

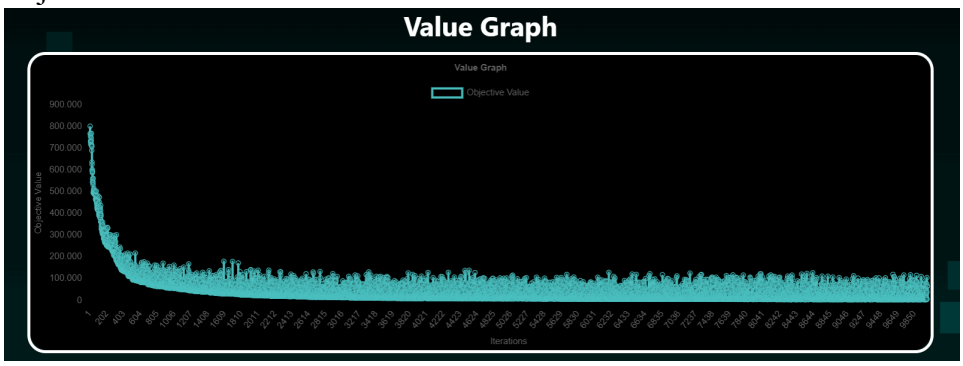
	

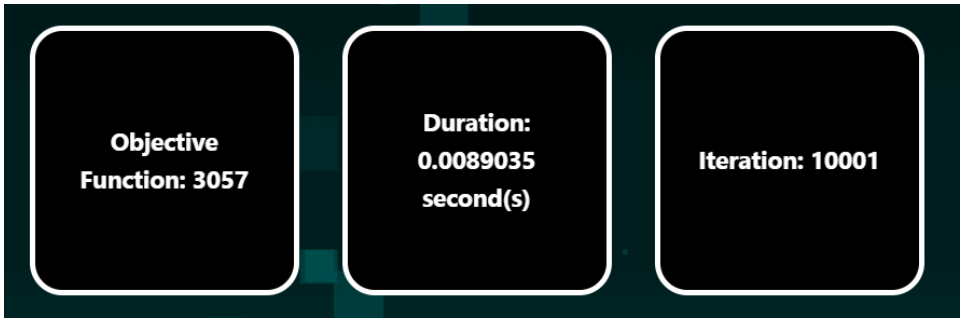
c. Eksperimen 3

State Awal	

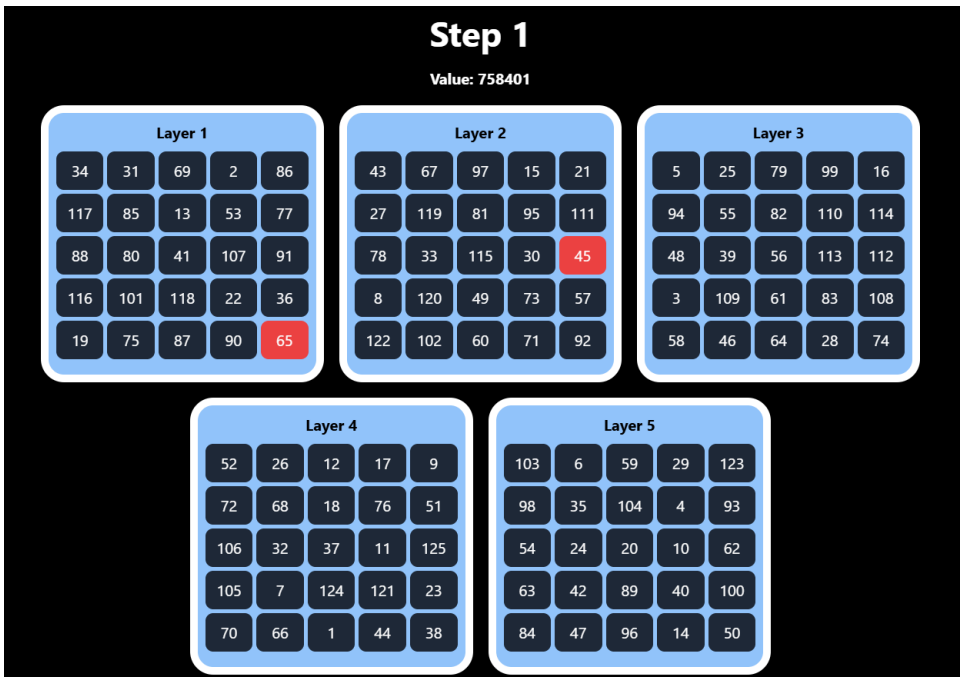
State Akhir	 Step 10001 Value: 66892 Layer 1 27 88 48 114 101 54 95 112 73 60 23 109 116 115 84 53 110 89 80 91 105 2 64 118 55 Layer 2 121 16 79 68 20 117 104 85 18 71 47 3 39 107 50 34 51 4 52 40 10 30 22 45 108 Layer 3 58 87 29 31 119 83 99 63 42 37 74 90 41 8 106 124 94 98 33 15 120 21 81 19 103 Layer 4 125 69 70 61 65 100 77 102 35 11 26 38 82 32 6 66 43 76 96 14 92 49 1 28 93 Layer 5 111 46 86 78 72 12 9 75 123 36 97 7 57 113 5 67 13 122 17 44 56 25 62 24 59

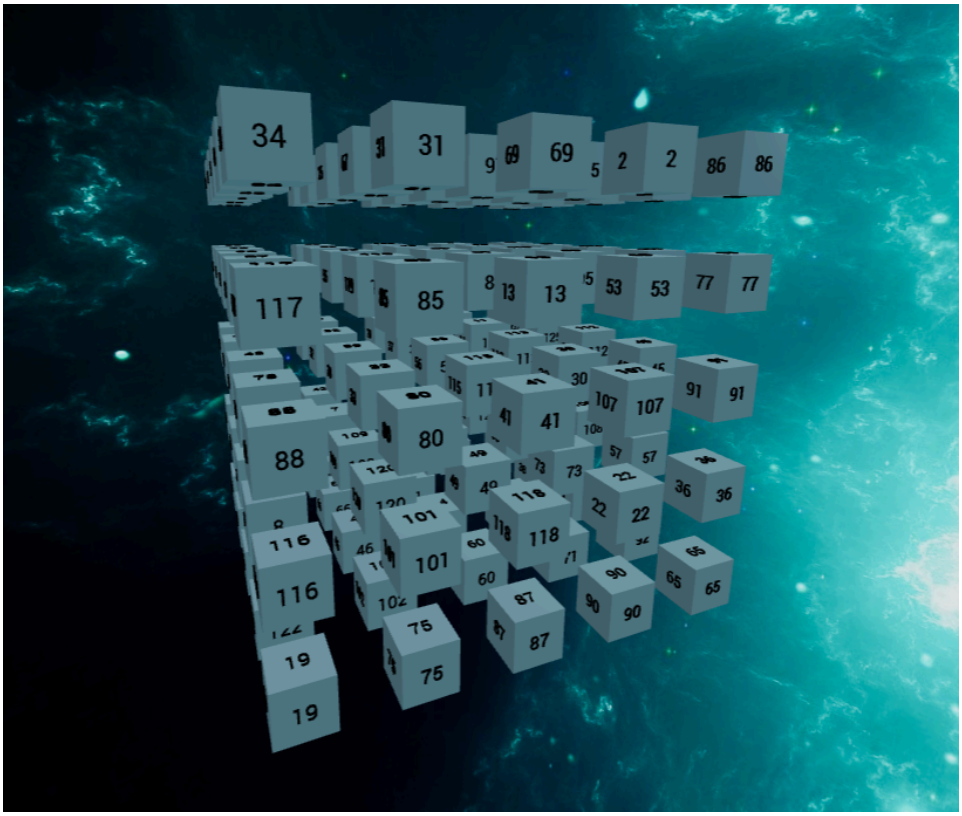
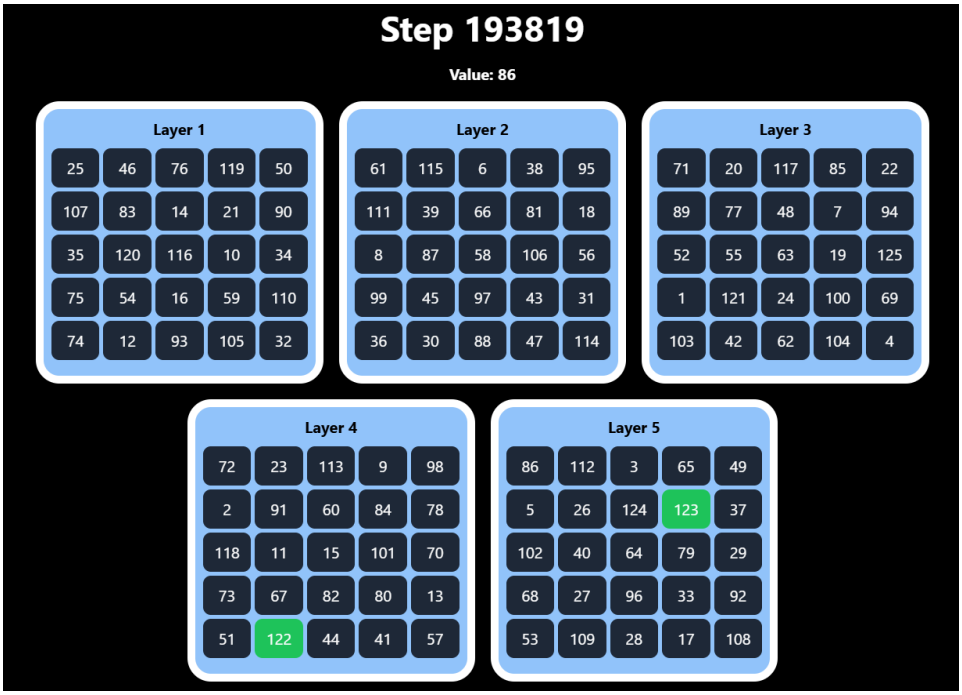
Objective Function	Objective Function: 3057 
Durasi	Durasi: 0.0089035 seconds

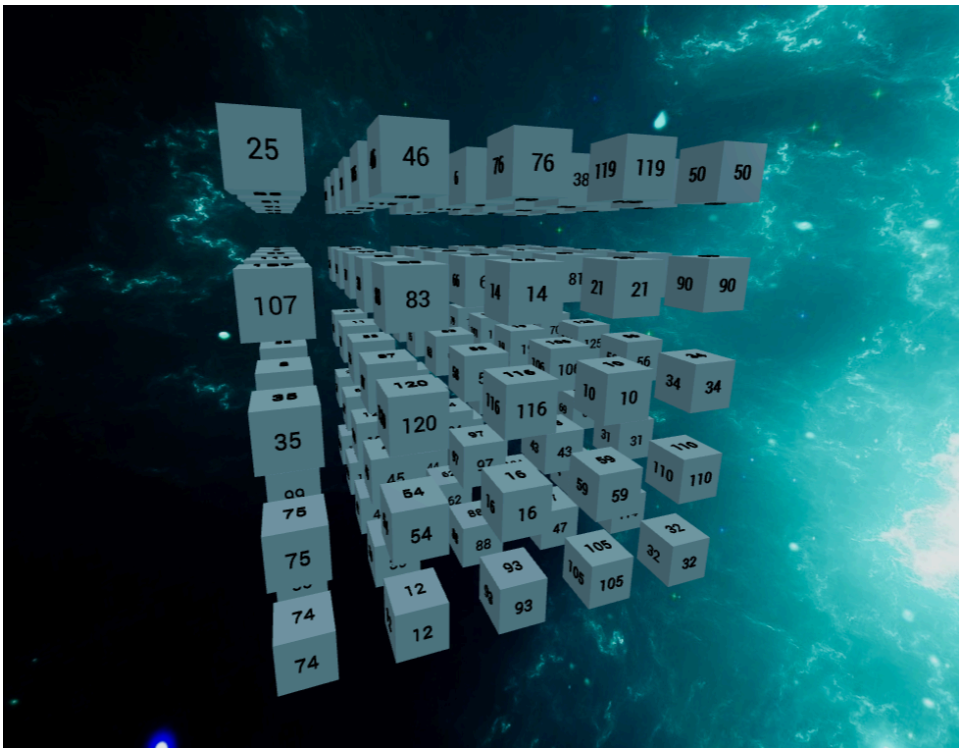
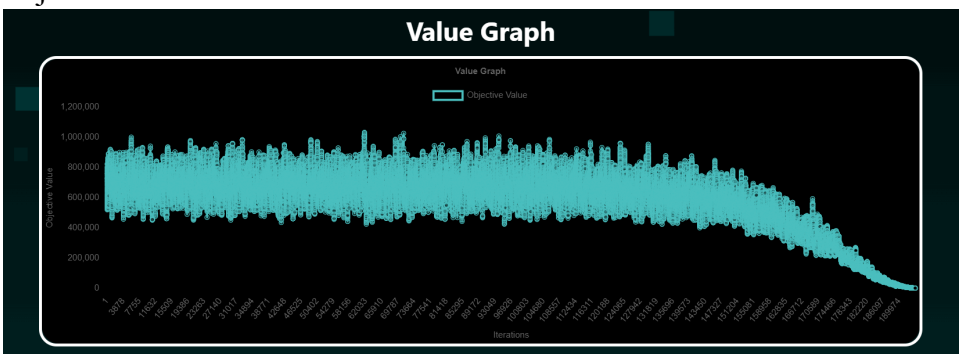
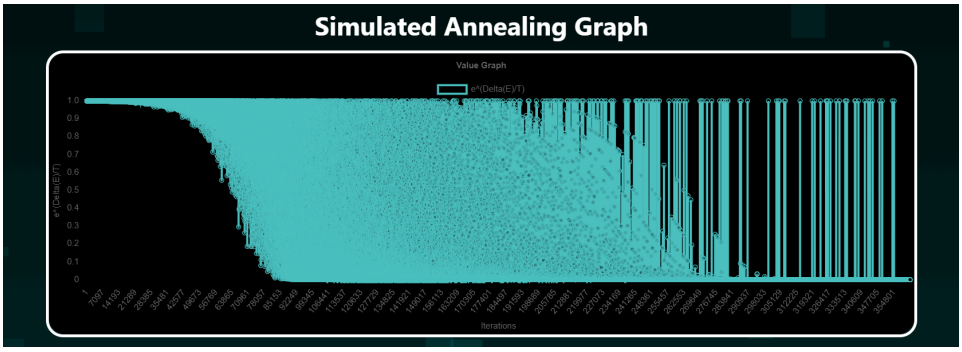
	

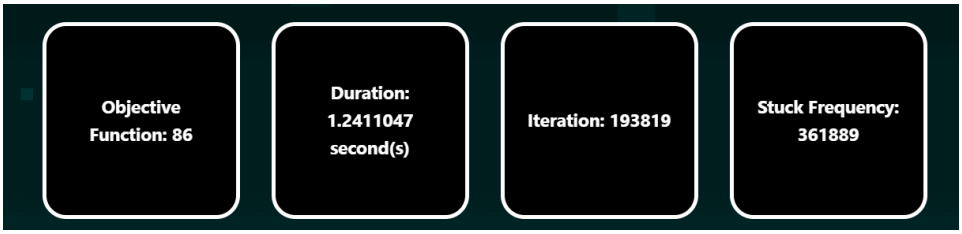
V. Simulated Annealing

a. Eksperimen 1

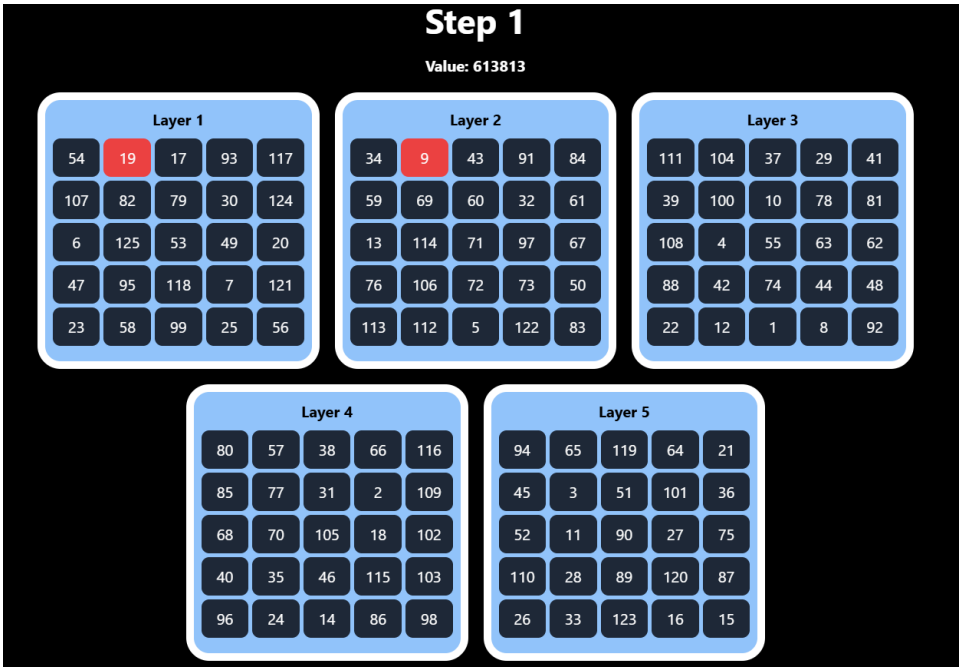
State Awal	

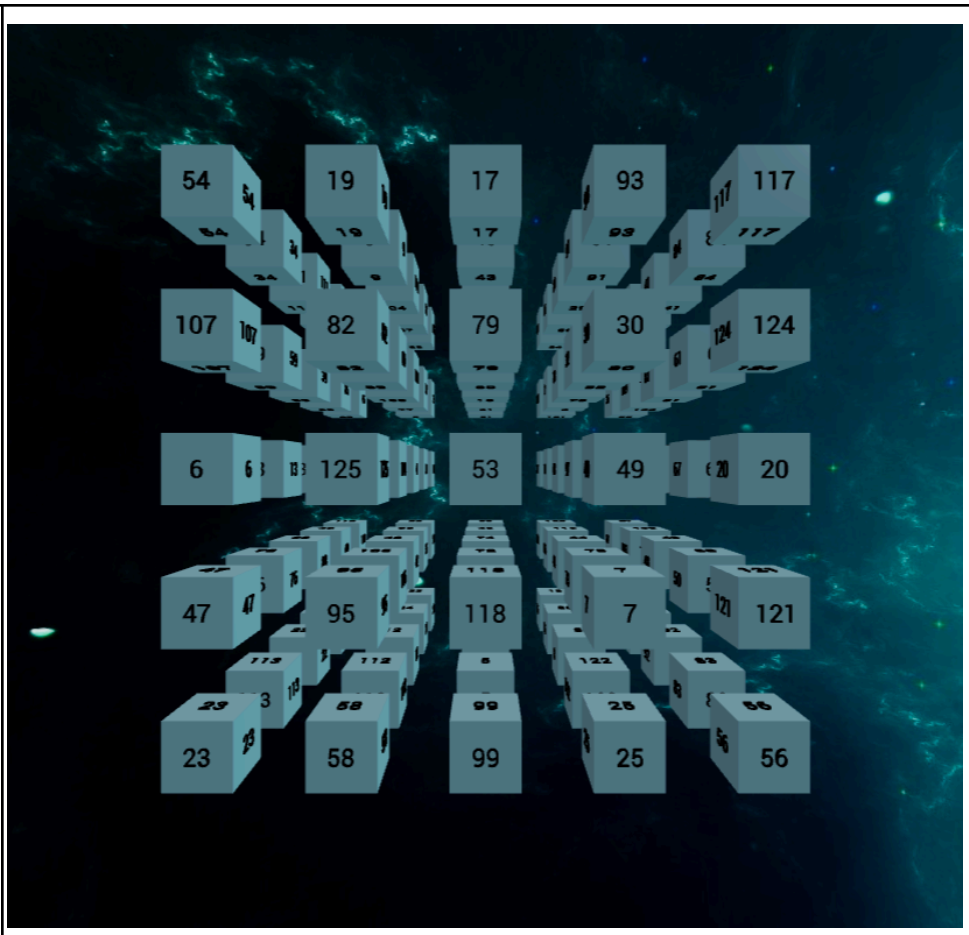
<table>
<tr>
<td></td>
<td>  </td>
</tr>
<tr>
<td>State Akhir</td>
<td> 
</td>
</tr>
</table>

	
Objective Function	Objective Function: 86 
Plot e^(delta(E)/T)	
Durasi dan Frekuensi Stuck	Durasi: 1.2411047 seconds Frekuensi Stuck: 361889

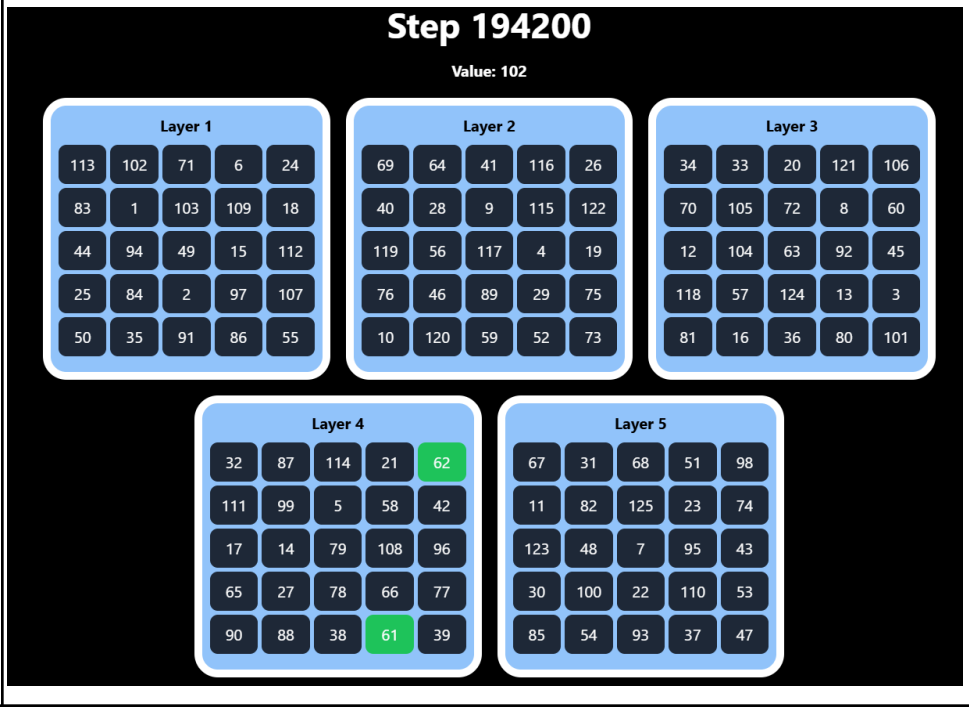
	

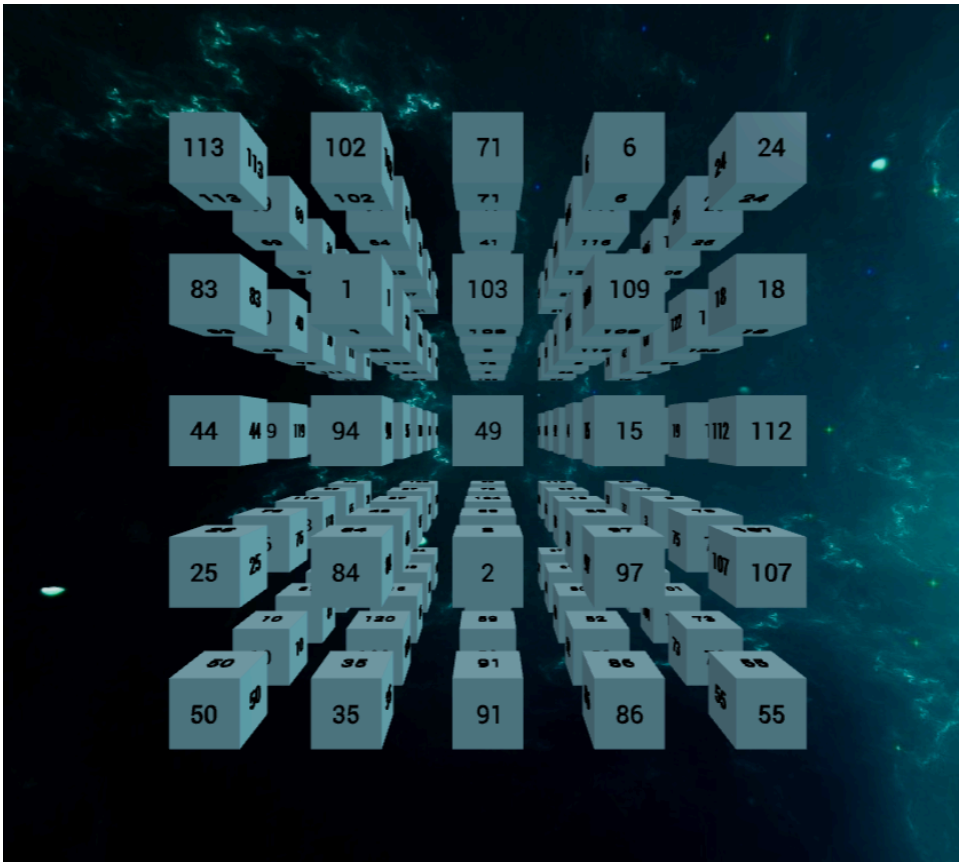
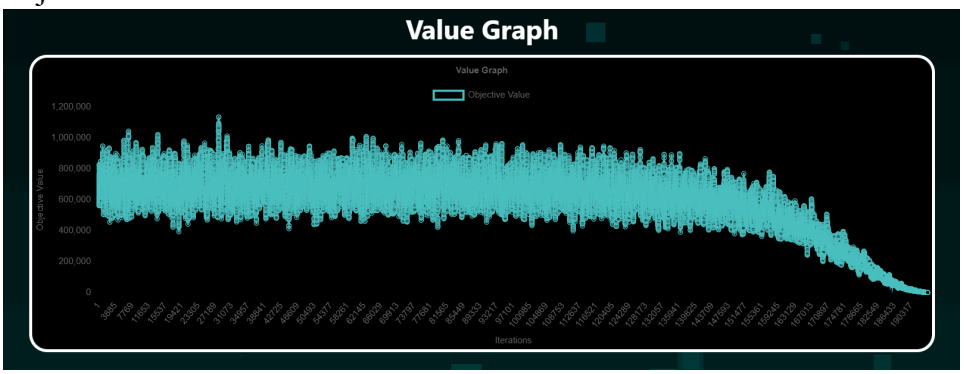
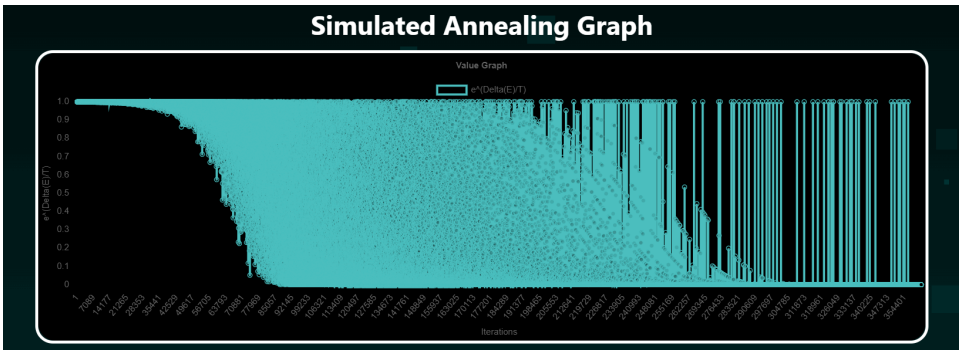
b. Eksperimen 2

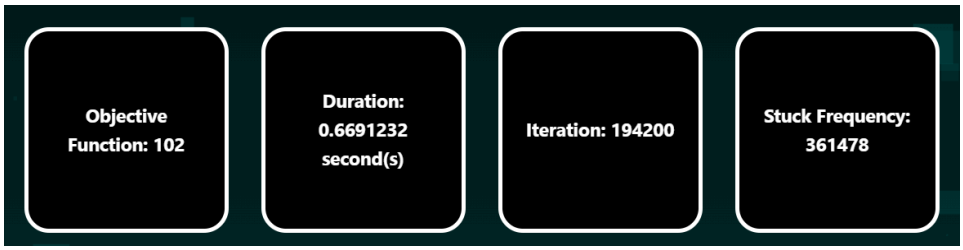
State Awal	


54
19
17
93
117
107
82
79
30
124
6
125
53
49
20
47
95
118
7
121
23
58
99
25
56

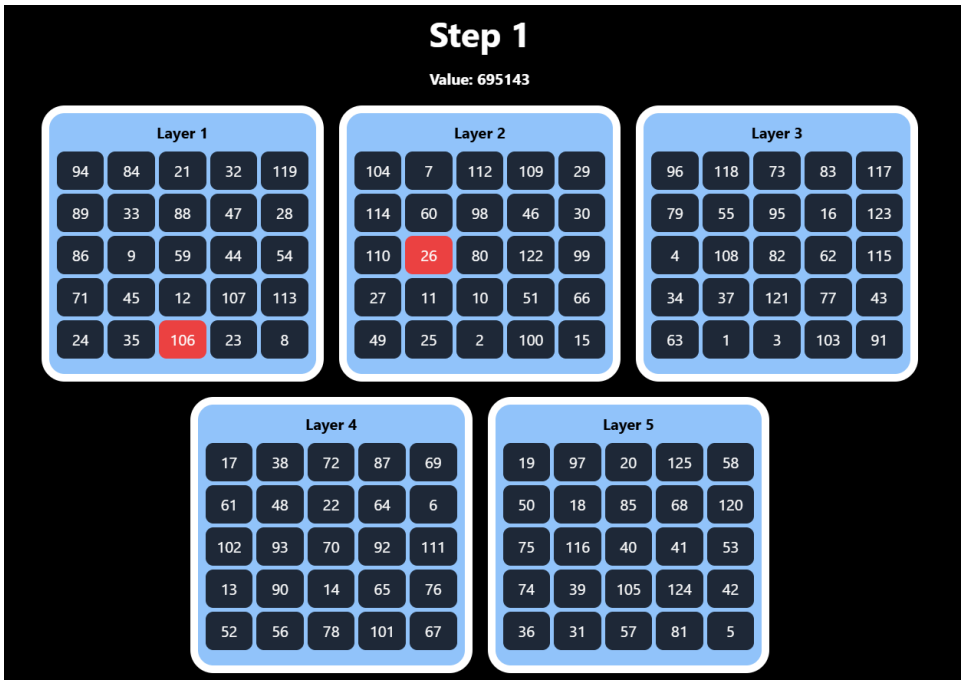
State Akhir

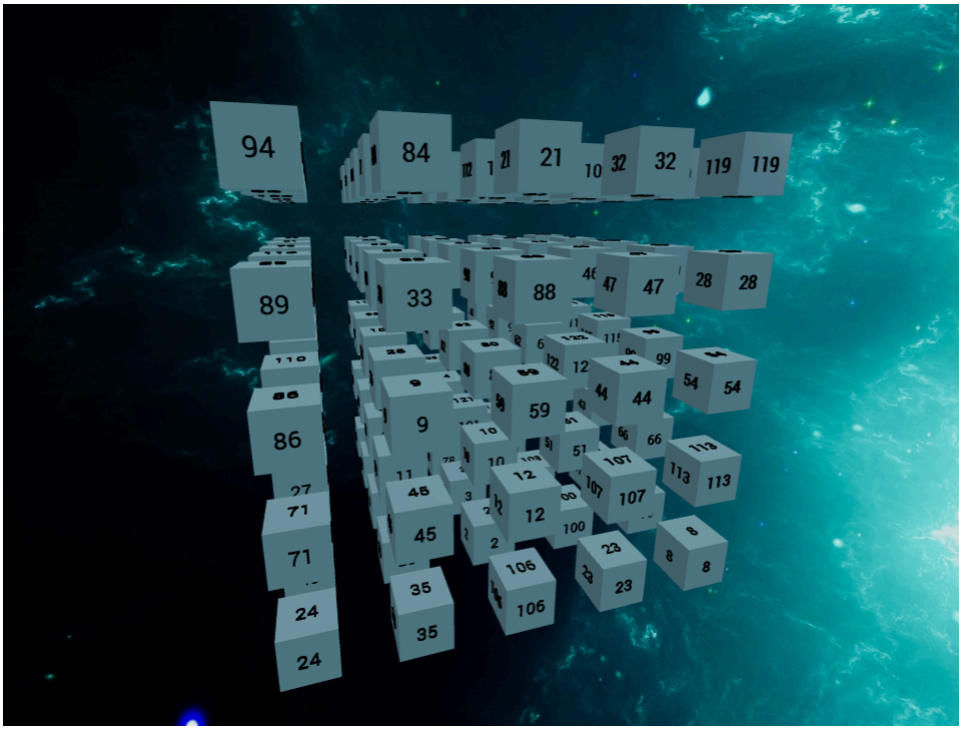
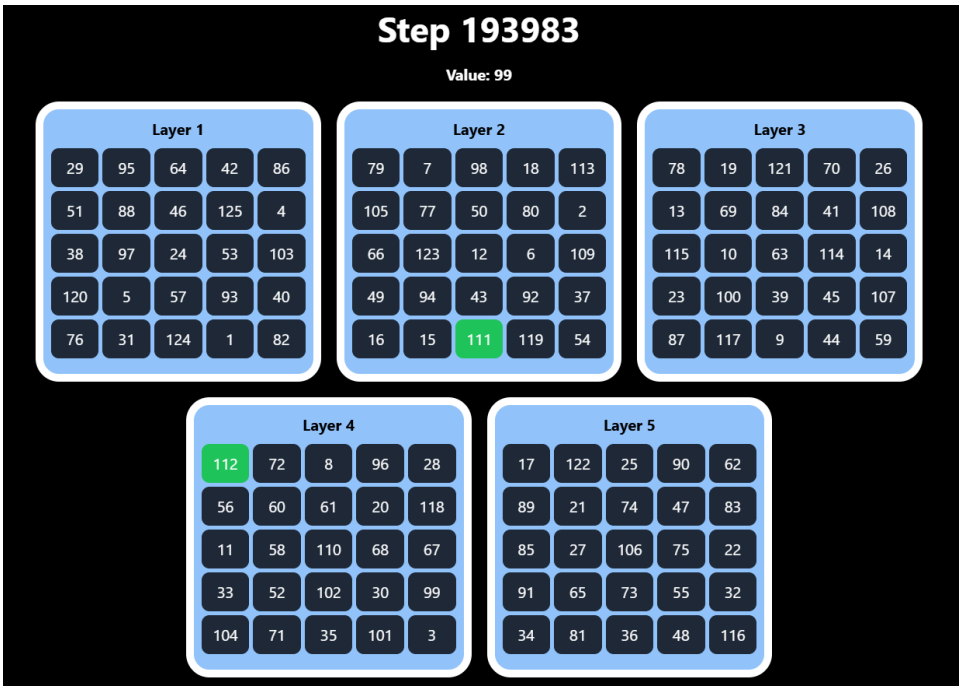

Step 194200
Value: 102
Layer 1
113 102 71 6 24
83 1 103 109 18
44 94 49 15 112
25 84 2 97 107
50 35 91 86 55
Layer 2
69 64 41 116 26
40 28 9 115 122
119 56 117 4 19
76 46 89 29 75
10 120 59 52 73
Layer 3
34 33 20 121 106
70 105 72 8 60
12 104 63 92 45
118 57 124 13 3
81 16 36 80 101
Layer 4
32 87 114 21 62
111 99 5 58 42
17 14 79 108 96
65 27 78 66 77
90 88 38 61 39
Layer 5
67 31 68 51 98
11 82 125 23 74
123 48 7 95 43
30 100 22 110 53
85 54 93 37 47

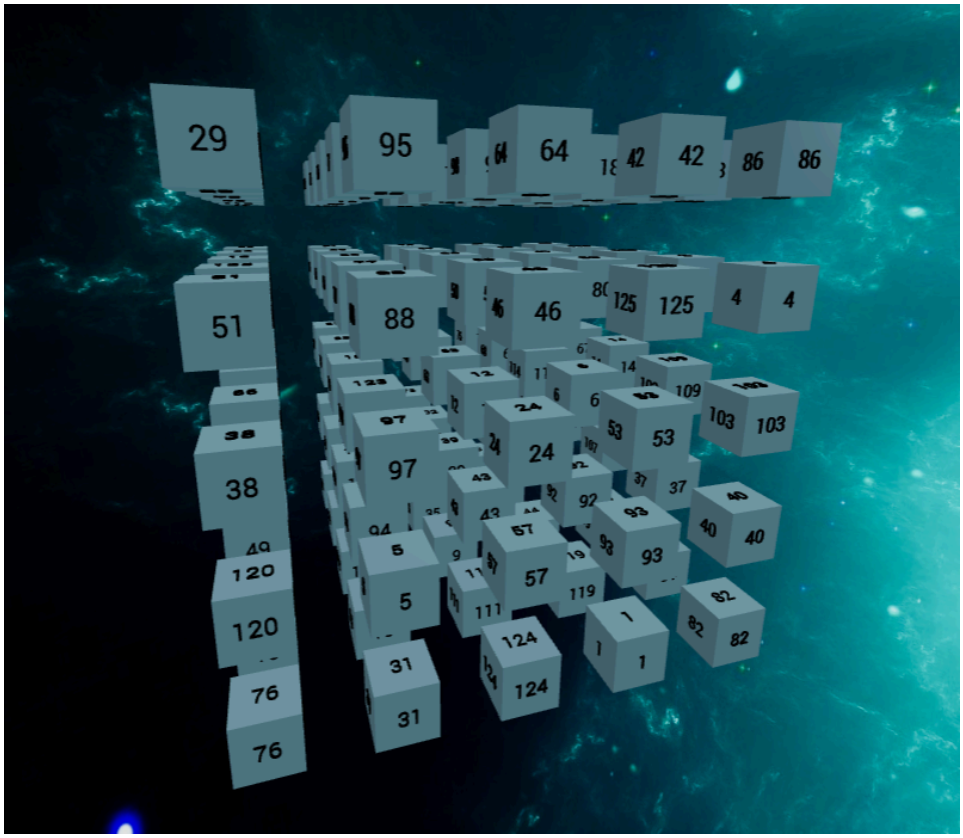
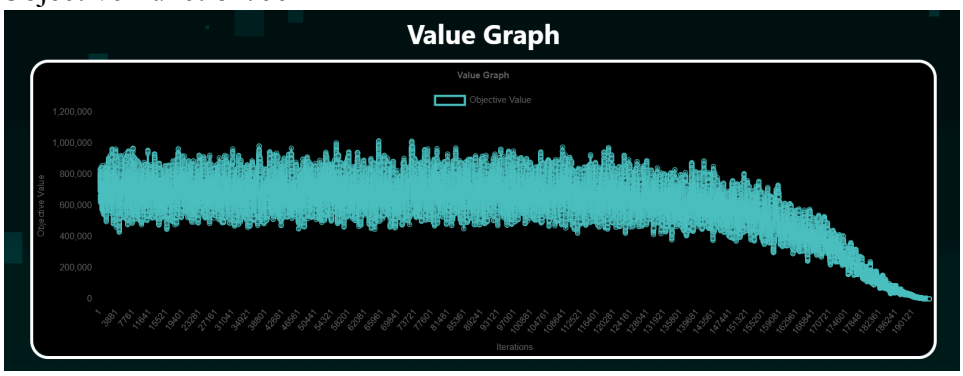
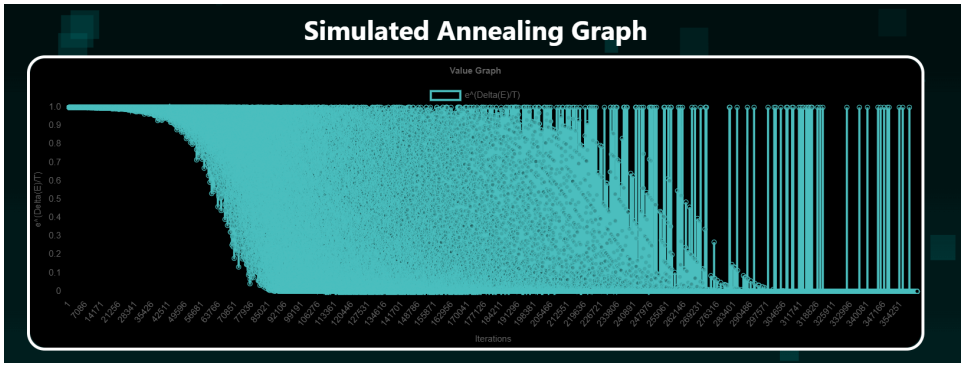
	
Objective Function	Objective Function: 102 
Plot e^(delta(E)/T)	

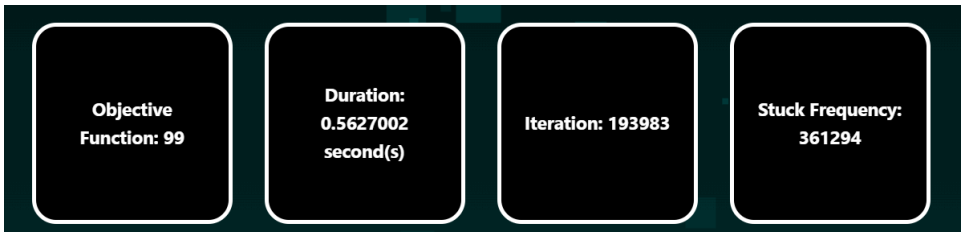
Durasi dan Frekuensi Stuck	Durasi: 0.6691232 seconds Frekuensi Stuck: 361478 

c. Eksperimen 3

State Awal	

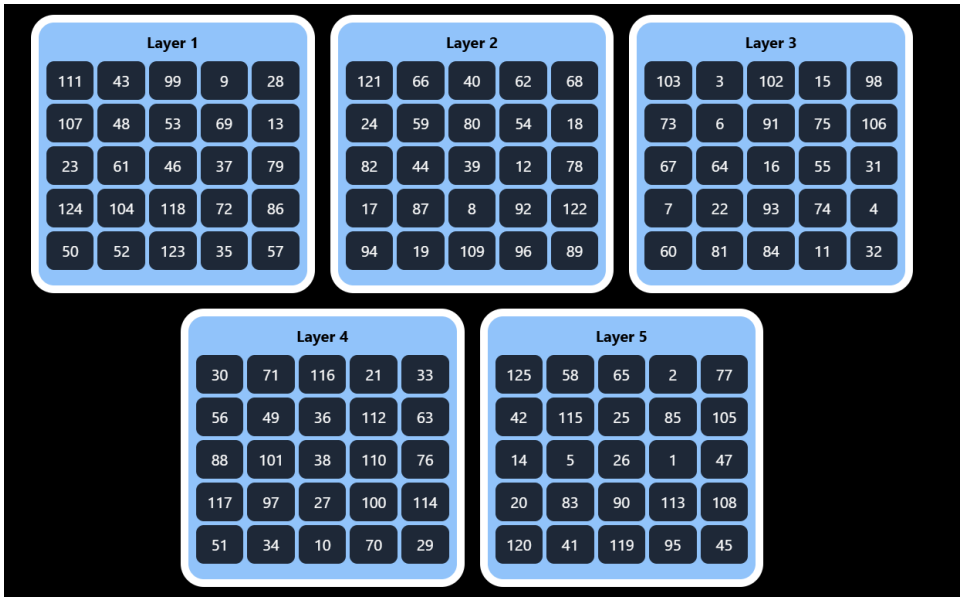
<table>
<tr>
<td></td>
<td>  </td>
</tr>
<tr>
<td>State Akhir</td>
<td> 
</td>
</tr>
</table>

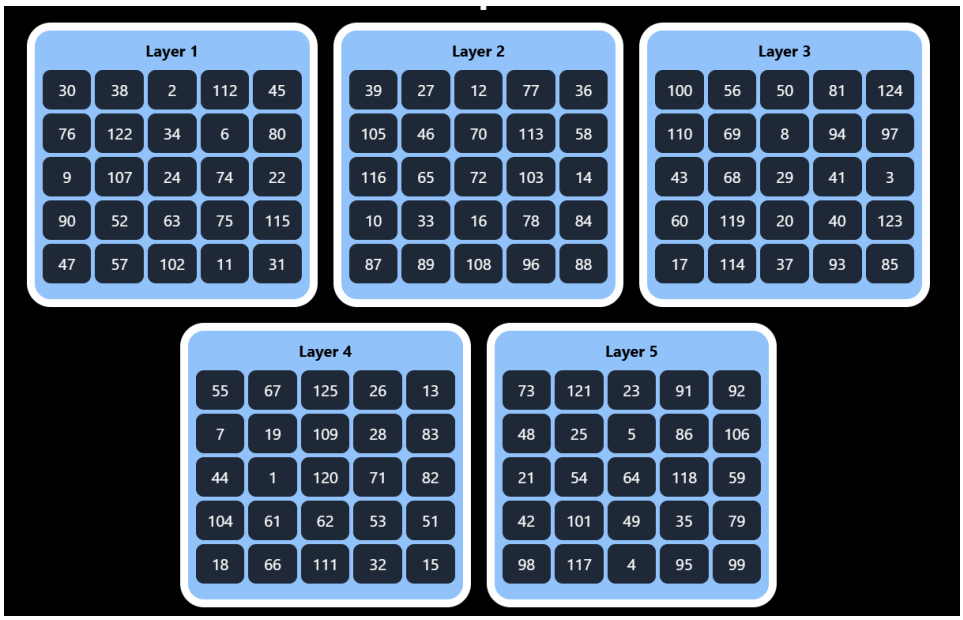
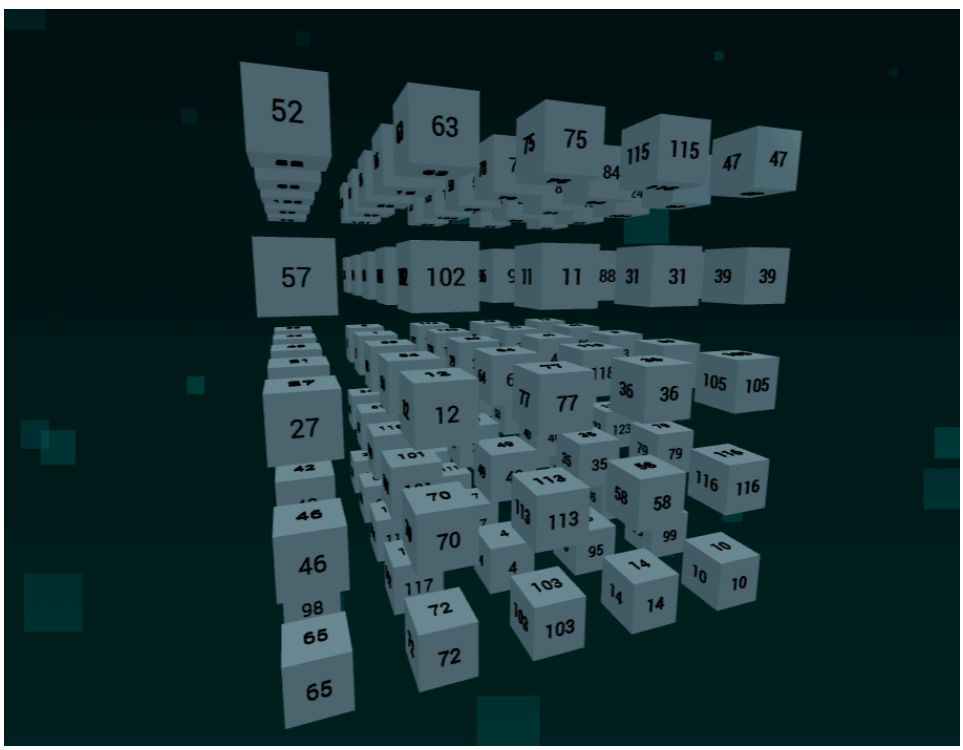
	
Objective Function	Objective Function: 99 
Plot e^(delta(E)/T)	

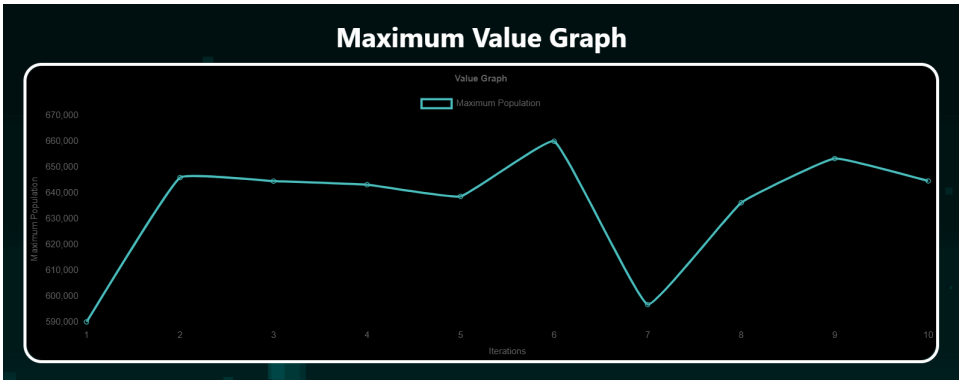
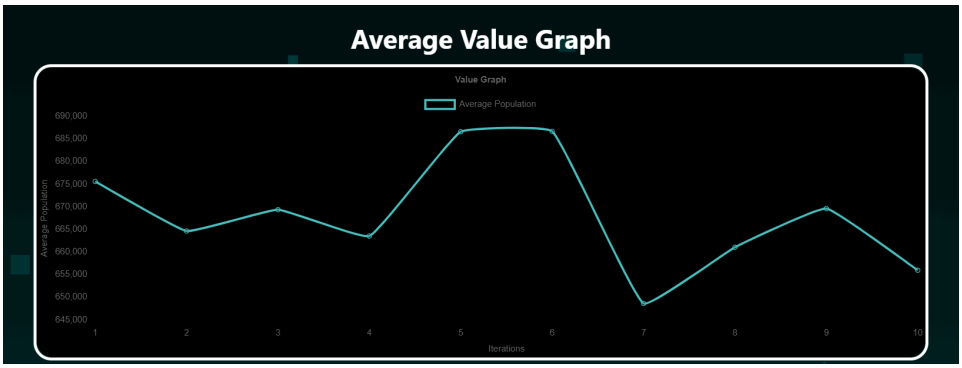
Durasi dan Frekuensi Stuck	Durasi: 0.5627002 seconds Frekuensi Stuck: 361294 

VI. Genetic Algorithm

a. Eksperimen 1

State Awal	

	
State Akhir	
Objective Function	Objective Function: 644714

	 
Durasi	Durasi: 0.0001407 seconds

b. Eksperimen 2

State Awal	

Layer 1

65	40	29	42	13
66	64	124	46	49
24	107	94	79	1
2	15	109	88	19
59	115	82	30	57

Layer 2

84	25	83	87	76
44	62	100	122	60
18	47	114	102	55
7	125	33	117	93
32	28	45	51	9

Layer 3

118	72	39	48	98
111	80	34	22	67
56	14	123	108	78
3	50	43	54	119
6	53	106	31	52

Layer 4

21	4	112	17	37
121	27	73	10	86
110	89	70	85	38
71	92	41	77	16
5	99	20	104	8

Layer 5

105	81	74	23	12
91	90	11	36	103
95	68	75	69	61
101	35	63	26	96
97	58	120	116	113

Layer 1

45	54	57	81	35
100	116	98	53	66
93	78	74	68	88
36	76	71	8	40
118	49	89	38	27

Layer 2

44	51	82	46	99
67	70	32	31	24
61	106	94	111	16
2	28	62	107	34
52	87	83	25	29

Layer 3

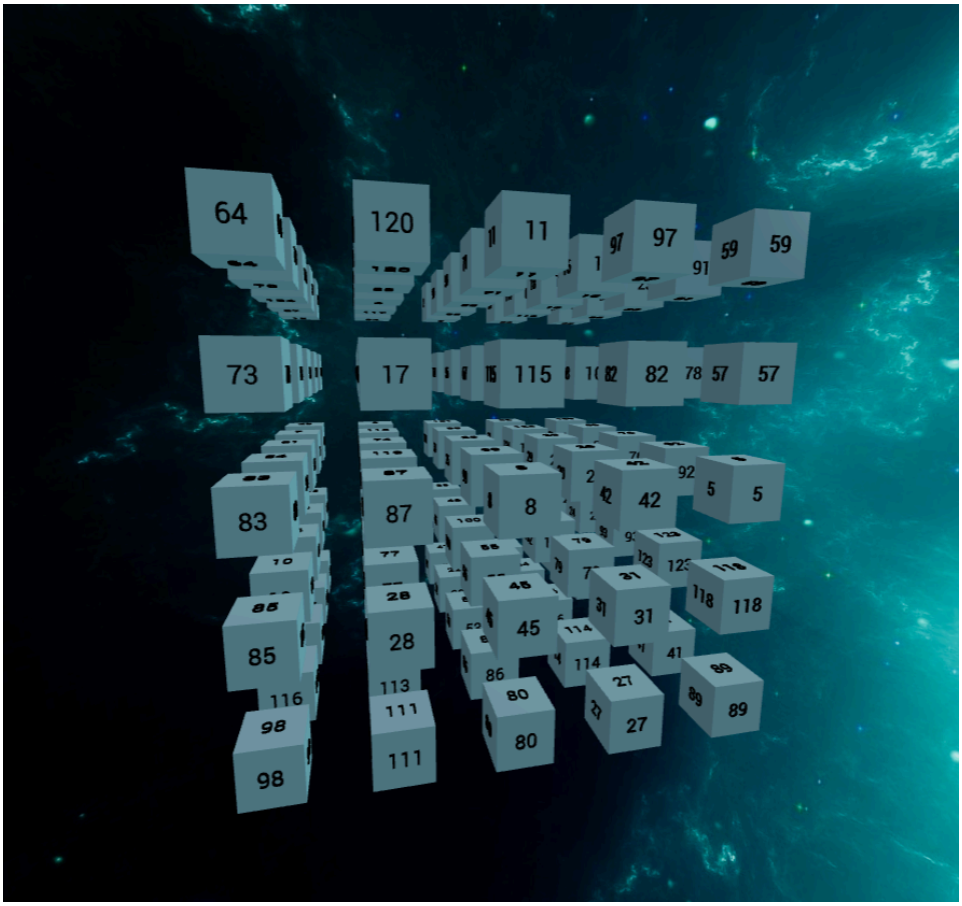
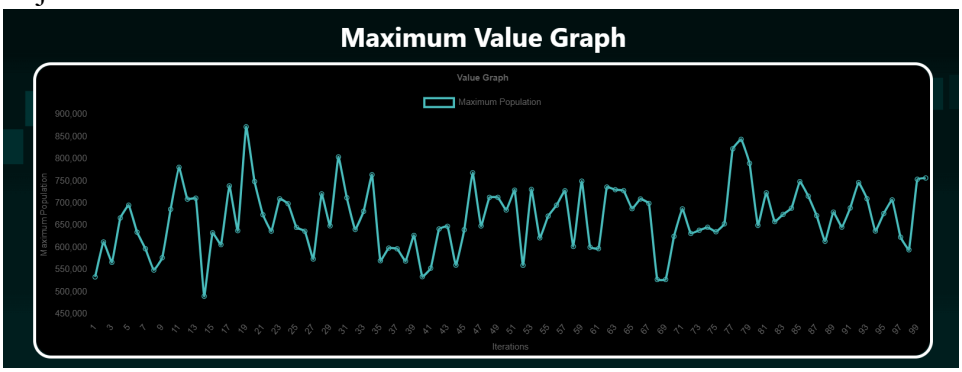
63	10	77	55	79
123	30	48	75	47
15	3	22	84	124
113	33	13	60	19
37	69	117	43	86

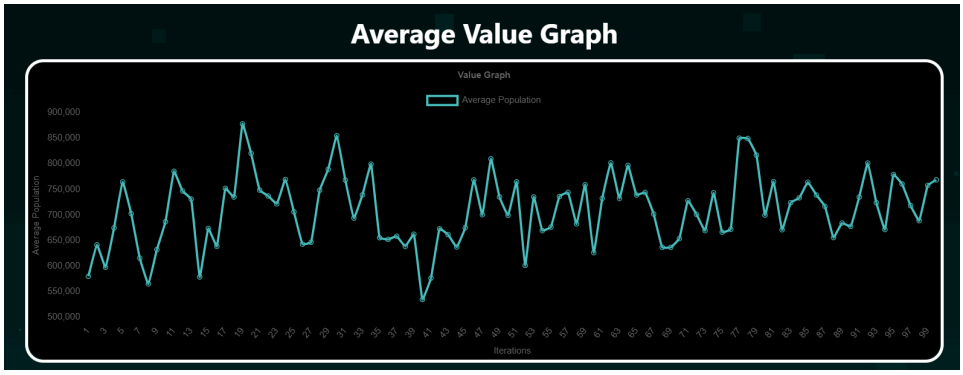
Layer 4

114	41	104	4	50
18	42	5	85	6
101	23	58	125	56
12	21	90	110	96
122	91	14	9	1

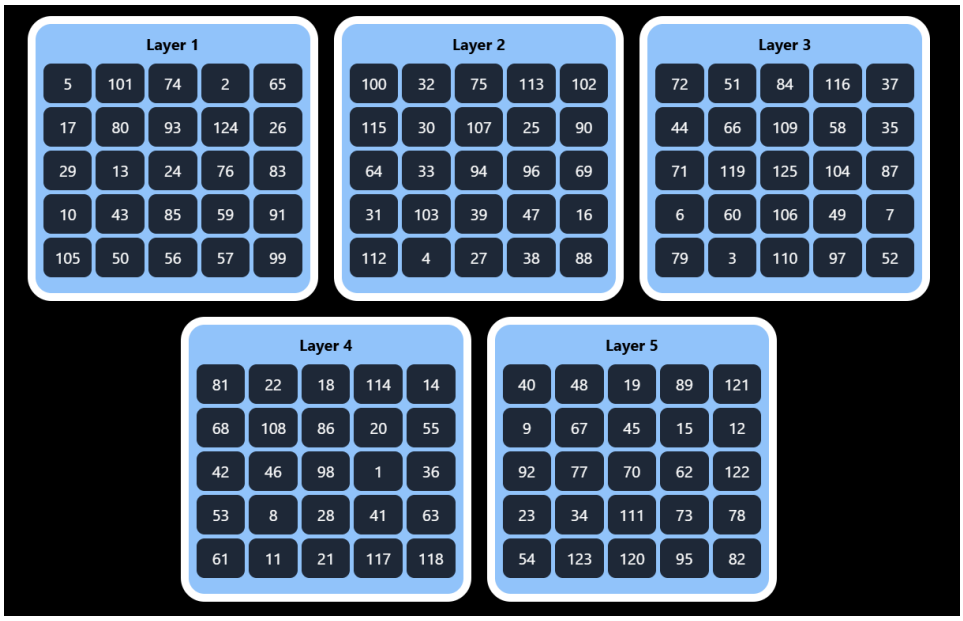
Layer 5

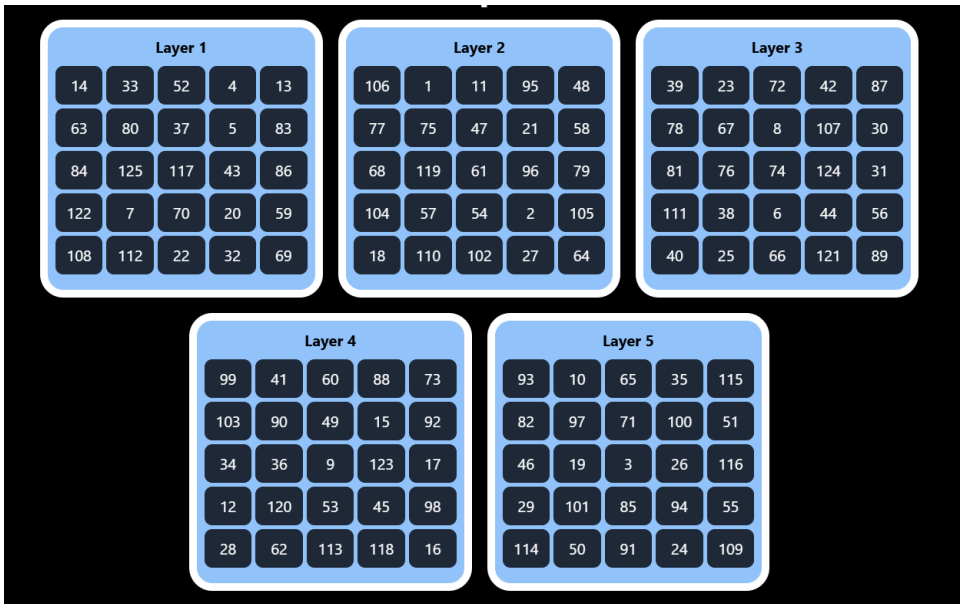
7	112	72	64	120
11	97	59	73	17
109	105	26	39	92
119	95	103	108	65
20	115	80	102	121

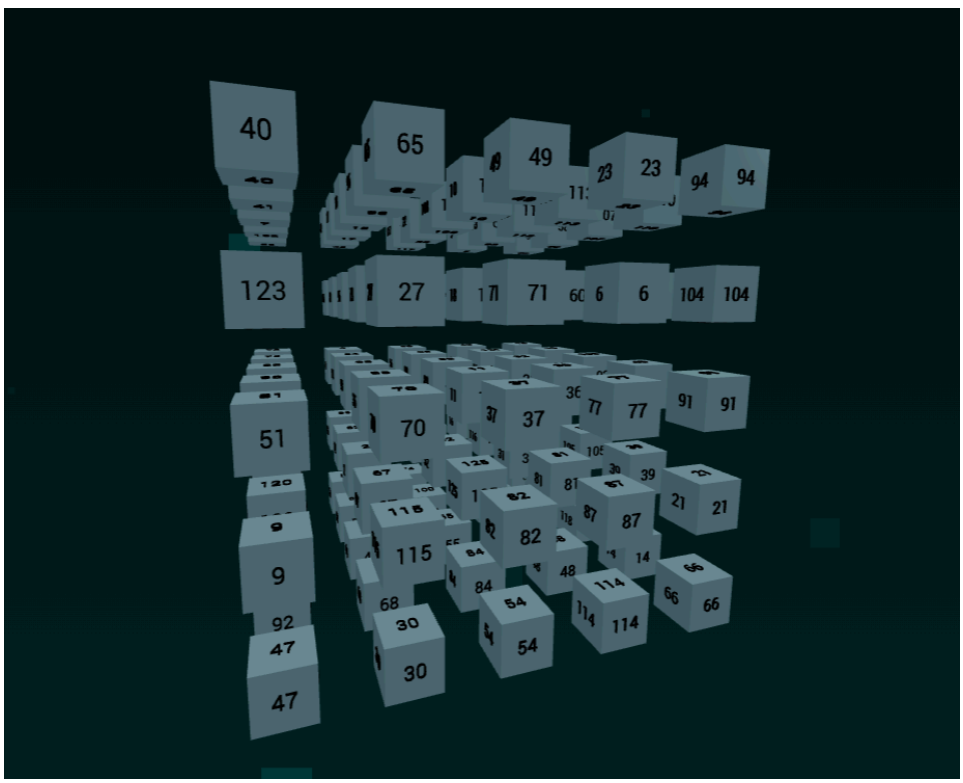
State Akhir	
Objective Function	Objective Function: 756789 

	
Durasi	Durasi: 0.0041902 seconds

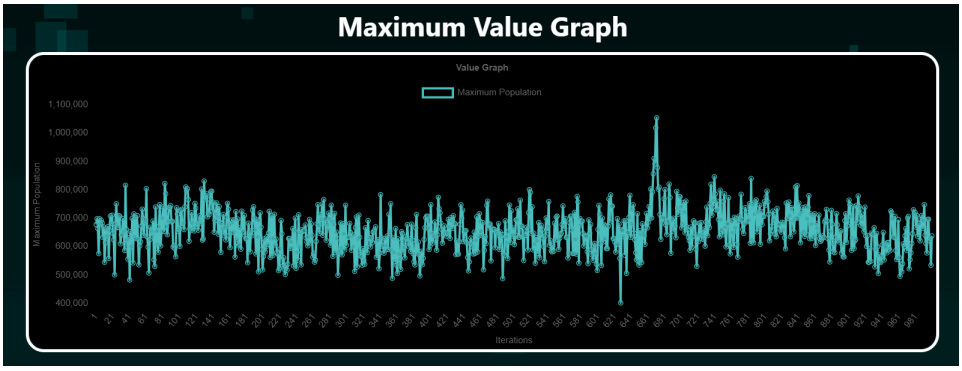
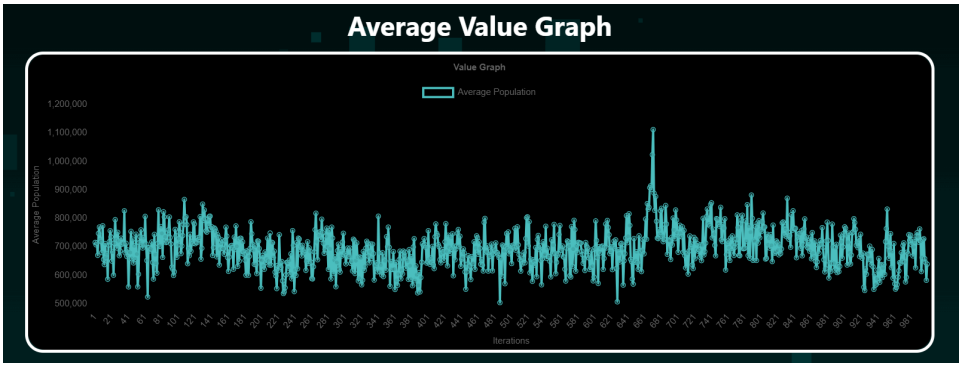
c. Eksperimen 3

State Awal	

<table>
<tr><td></td><td>


</td></tr>
<tr><td>State Akhir</td><td>


</td></tr>
<tr><td>Objective Function</td><td>Objective Function: 637975</td></tr>
</table>

	 
Durasi	Durasi: 0.0474343 seconds

d. Eksperimen 4

State Awal	

Layer 1

61	100	122	8	18
123	86	87	36	49
121	59	112	37	72
56	74	82	63	5
120	48	57	83	40

Layer 2

109	16	75	81	80
103	54	89	85	43
101	62	94	46	104
15	124	29	71	35
28	41	76	55	44

Layer 3

107	114	26	14	24
33	12	73	65	115
118	106	60	95	4
91	96	105	31	69
99	88	70	93	79

Layer 4

39	34	21	7	30
108	68	38	50	119
9	23	32	47	6
22	113	42	51	84
64	52	98	11	58

Layer 5

10	1	117	20	125
110	3	97	111	116
67	13	53	45	92
77	25	78	66	90
102	27	17	19	2

Layer 1

114	119	74	104	105
90	51	60	93	112
107	63	23	67	96
116	27	124	118	52
100	97	11	30	78

Layer 2

33	54	85	92	37
69	84	40	3	46
70	32	28	20	9
111	41	26	14	125
4	77	65	115	18

Layer 3

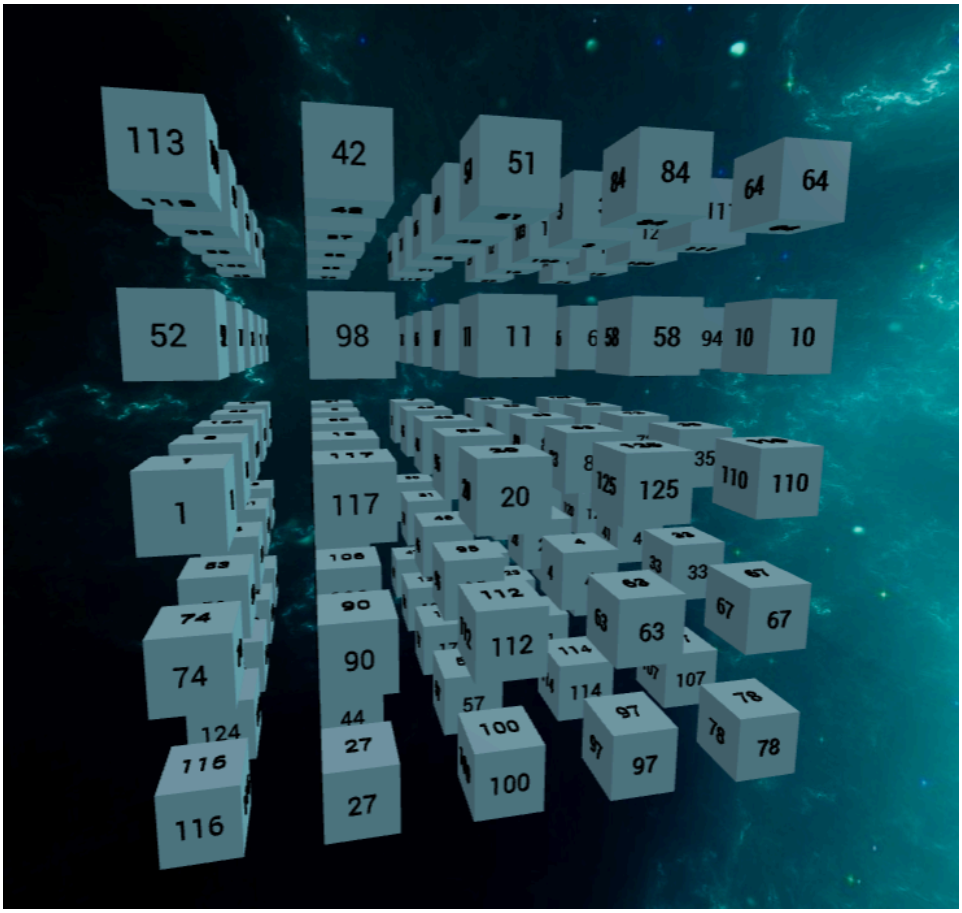
87	17	42	58	34
91	121	88	109	25
71	12	15	2	110
82	113	117	73	59
81	7	64	106	43

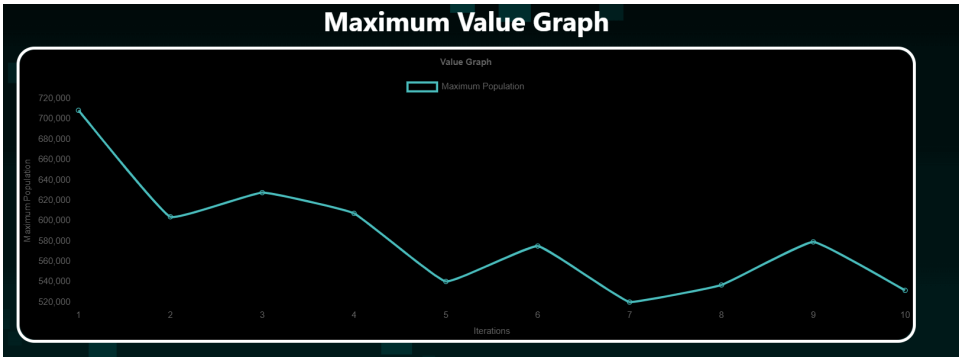
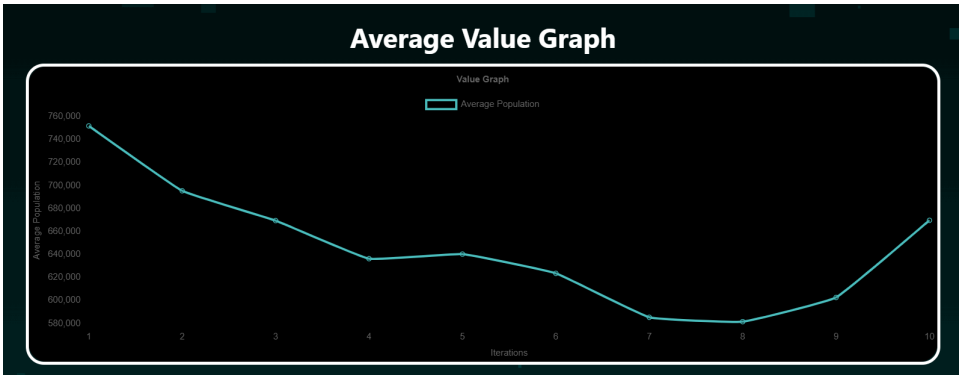
Layer 4

5	45	80	49	47
101	98	61	31	56
120	95	50	102	103
44	19	57	6	29
39	53	76	35	79

Layer 5

99	48	1	16	10
68	89	8	38	123
86	66	122	72	24
62	22	21	36	108
75	94	55	13	83

State Akhir	
Objective	Objective Function: 531635

Function	 
Durasi	Durasi: 0.0005077 seconds

e. Eksperimen 5

State Awal	

Layer 1

87	36	70	28	1
40	47	56	30	11
124	86	39	8	50
25	60	73	92	91
97	65	95	82	106

Layer 2

104	5	120	15	32
112	22	96	38	113
66	21	27	102	93
85	35	26	24	125
105	111	89	64	80

Layer 3

103	12	2	44	117
123	99	69	88	108
84	98	52	110	51
53	42	62	34	3
90	101	68	17	16

Layer 4

119	19	72	74	94
57	20	114	23	55
9	49	71	43	122
116	13	67	61	79
33	83	100	63	115

Layer 5

18	4	59	107	77
109	46	54	75	31
10	41	29	37	14
7	58	118	6	45
121	81	78	76	48

Layer 1

67	83	40	81	74
116	86	102	36	18
95	4	54	105	72
80	44	122	123	56
94	119	8	23	124

Layer 2

17	104	1	101	11
62	28	53	42	63
108	39	64	106	85
115	89	38	31	26
121	15	96	100	33

Layer 3

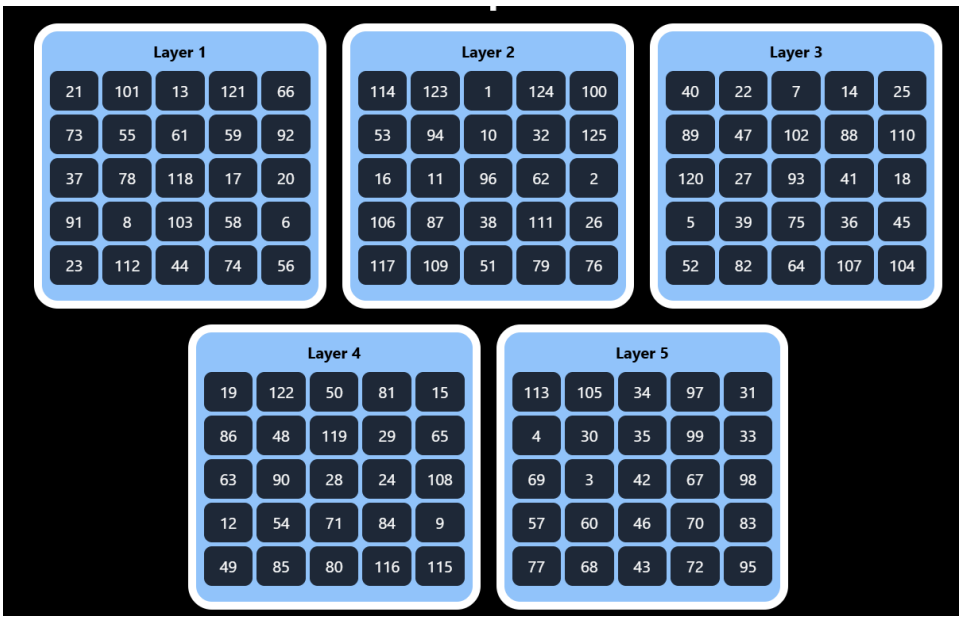
111	66	2	51	14
35	97	77	125	69
30	59	58	82	6
84	75	20	91	61
24	110	76	7	68

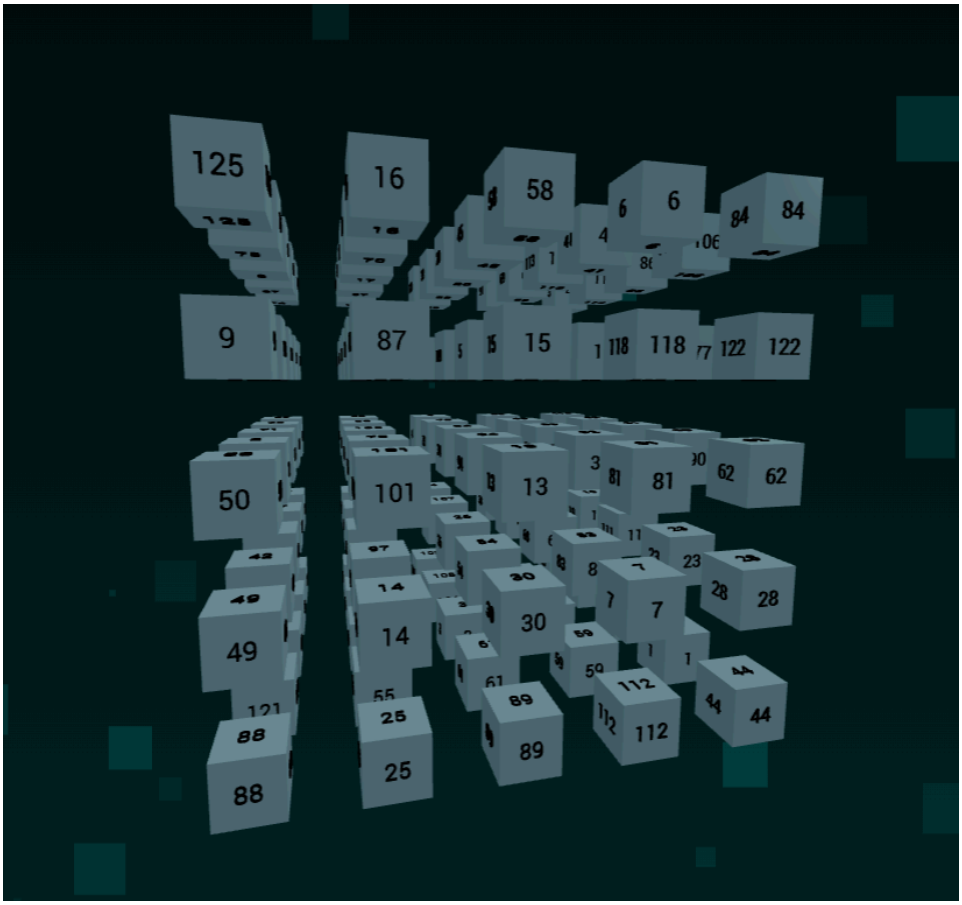
Layer 4

109	50	22	79	52
114	29	87	118	41
46	99	93	5	19
92	25	90	103	113
27	43	55	70	60

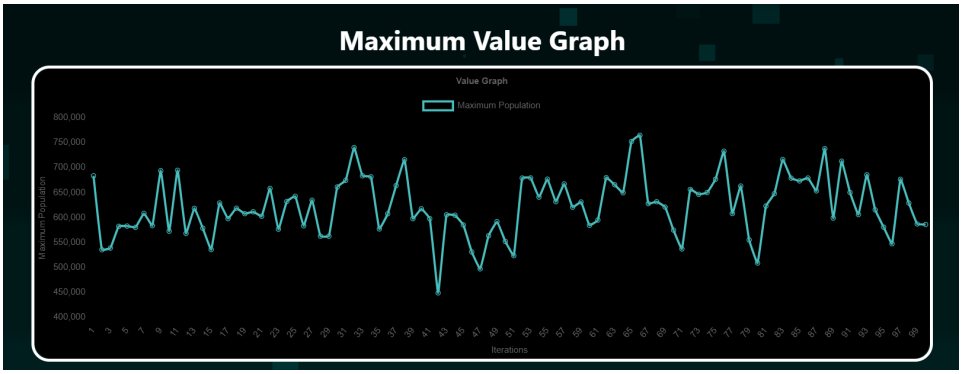
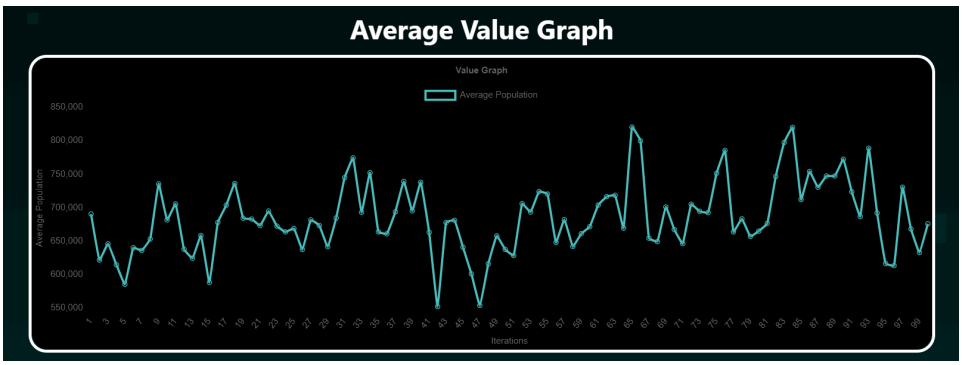
Layer 5

16	48	112	34	71
65	21	98	12	120
10	13	78	88	73
3	107	32	9	117
45	37	47	49	57

<table>
<tr><td></td><td>


</td></tr>
<tr><td>State Akhir</td><td>


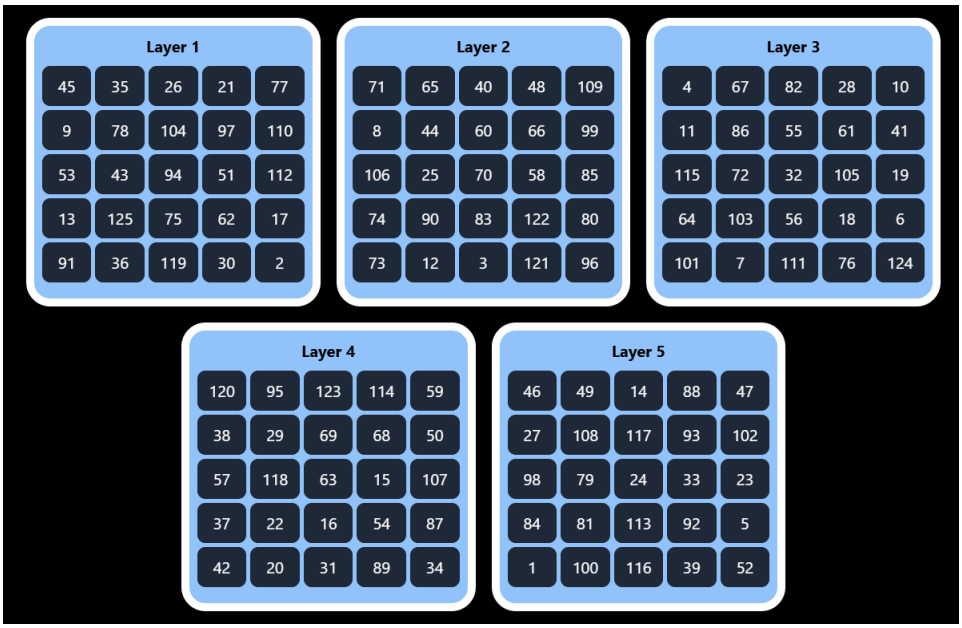
</td></tr>
<tr><td>Objective</td><td>Objective Function: 586111</td></tr>
</table>

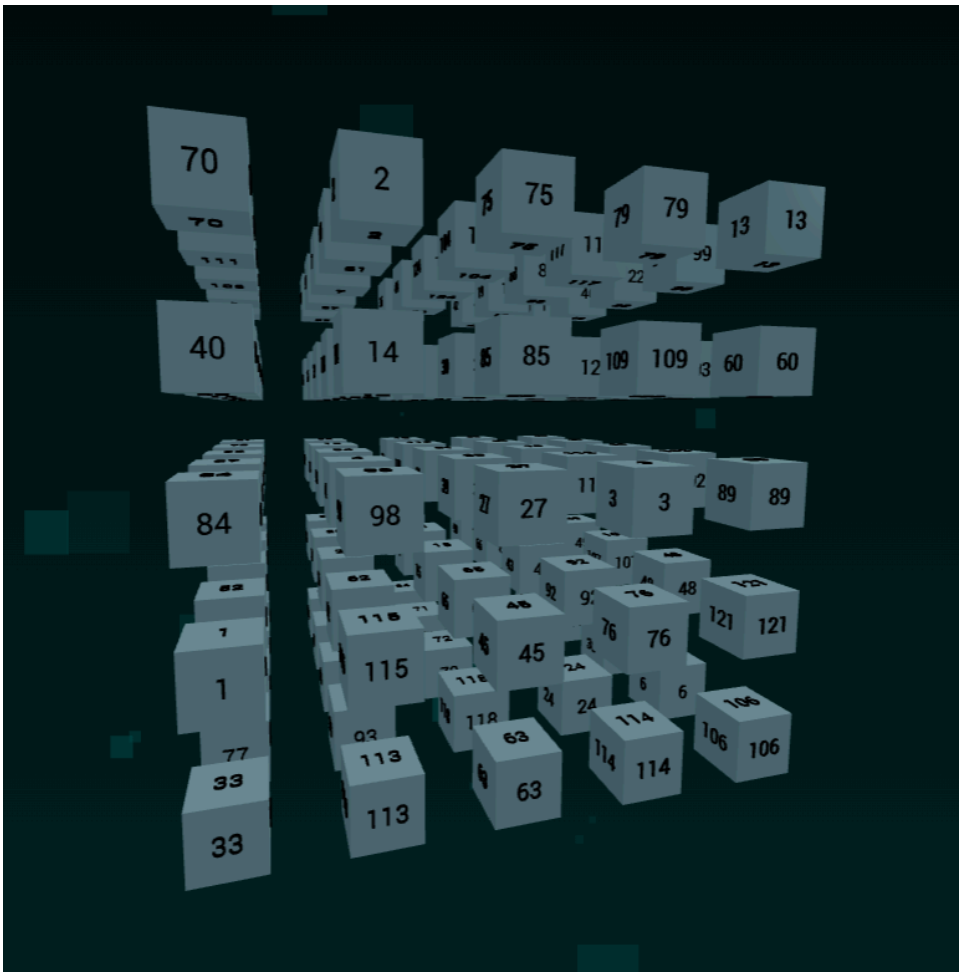
Function	 
Durasi	Durasi: 0.005933 seconds

f. Eksperimen 6

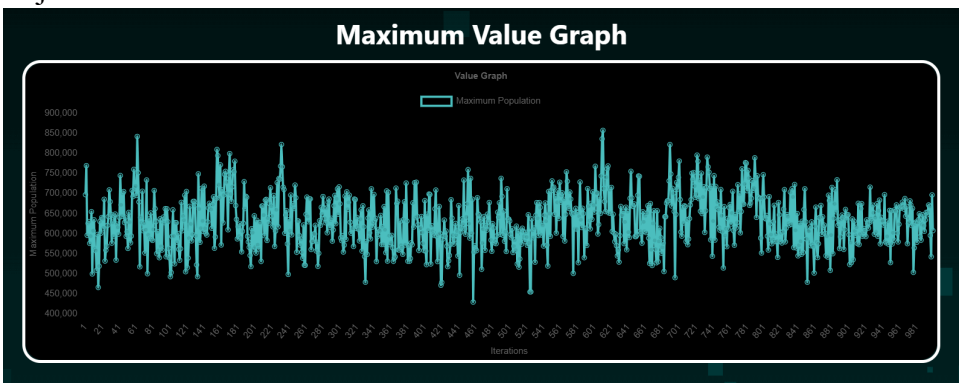
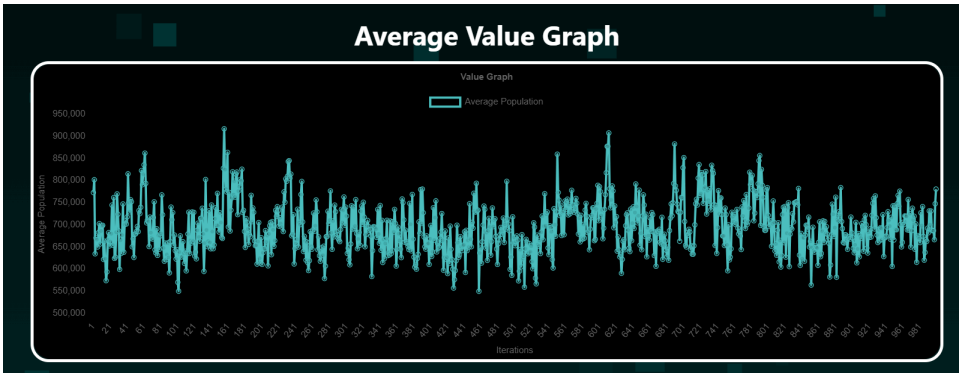
State Awal	

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

<table>
<tr>
<td></td>
<td>


</td>
</tr>
<tr>
<td>State Akhir</td>
<td>


</td>
</tr>
</table>

Objective Function	Objective Function: 607562  
Durasi	Durasi: 0.0701568 seconds

g. Eksperimen 7

State Awal	

Layer 1

86	8	122	71	110
43	74	59	65	39
98	50	101	107	45
19	84	75	111	53
73	14	112	121	117

Layer 2

12	22	123	57	62
58	16	108	79	47
11	114	72	3	5
41	44	94	90	68
32	30	4	87	89

Layer 3

52	96	70	102	61
26	55	38	97	88
40	78	83	93	125
54	7	46	60	13
106	35	76	66	124

Layer 4

120	6	42	67	109
31	34	33	63	23
24	92	28	104	99
64	51	36	81	113
48	103	2	105	115

Layer 5

18	20	85	69	15
116	82	27	9	100
37	77	21	119	25
80	10	118	49	95
1	91	29	17	56

Layer 1

62	112	48	98	66
105	120	3	80	67
117	51	35	11	27
20	114	75	16	34
40	8	36	86	57

Layer 2

90	87	95	99	106
89	123	71	121	119
19	37	44	47	49
55	22	2	83	23
5	17	78	53	15

Layer 3

58	101	79	92	115
56	82	94	91	109
61	107	32	64	122
24	31	73	21	10
50	81	102	25	63

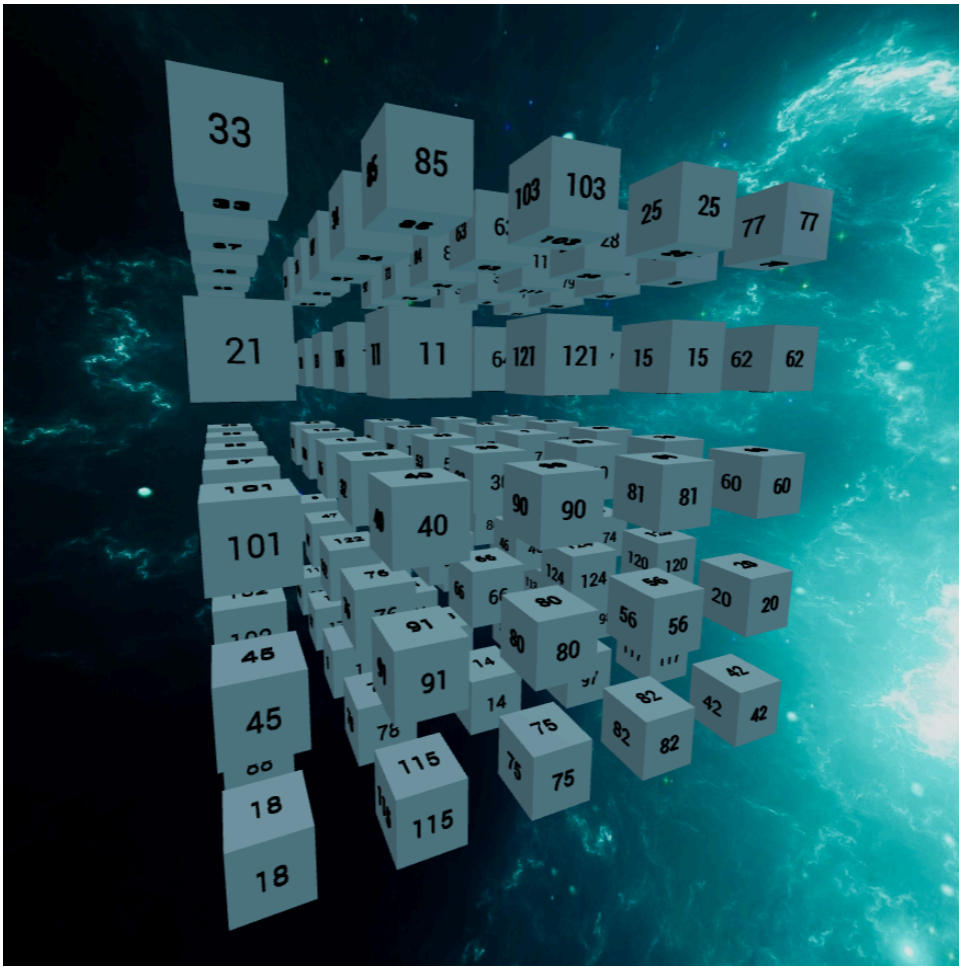
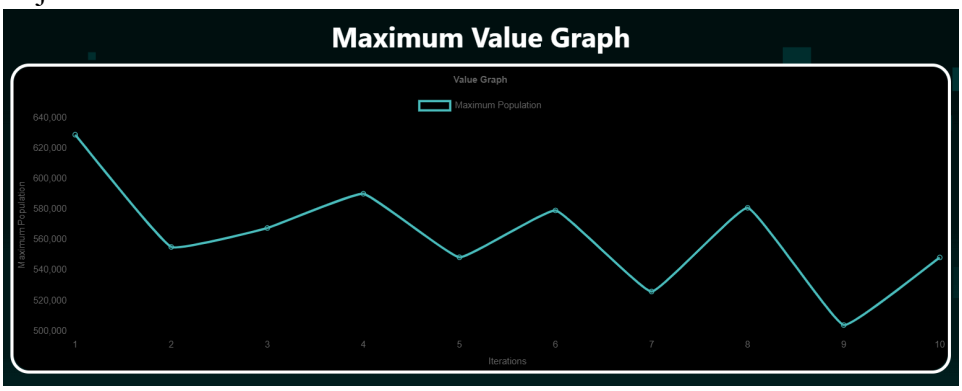
Layer 4

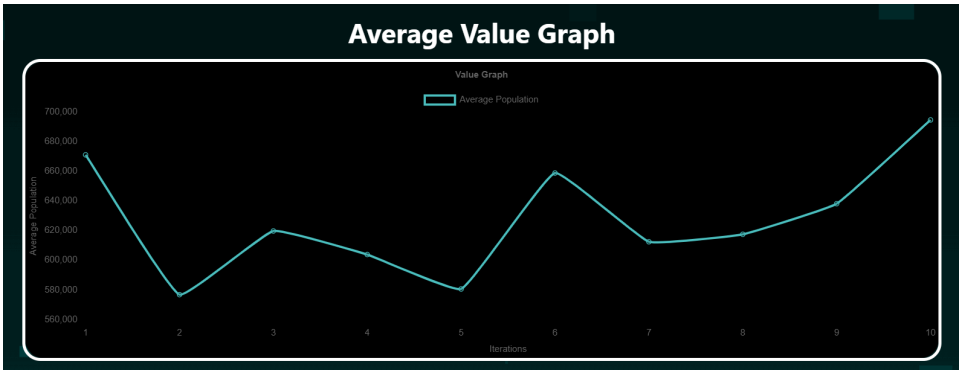
124	41	29	113	26
39	68	18	42	12
1	108	111	7	100
45	97	116	54	76
84	52	70	46	85

Layer 5

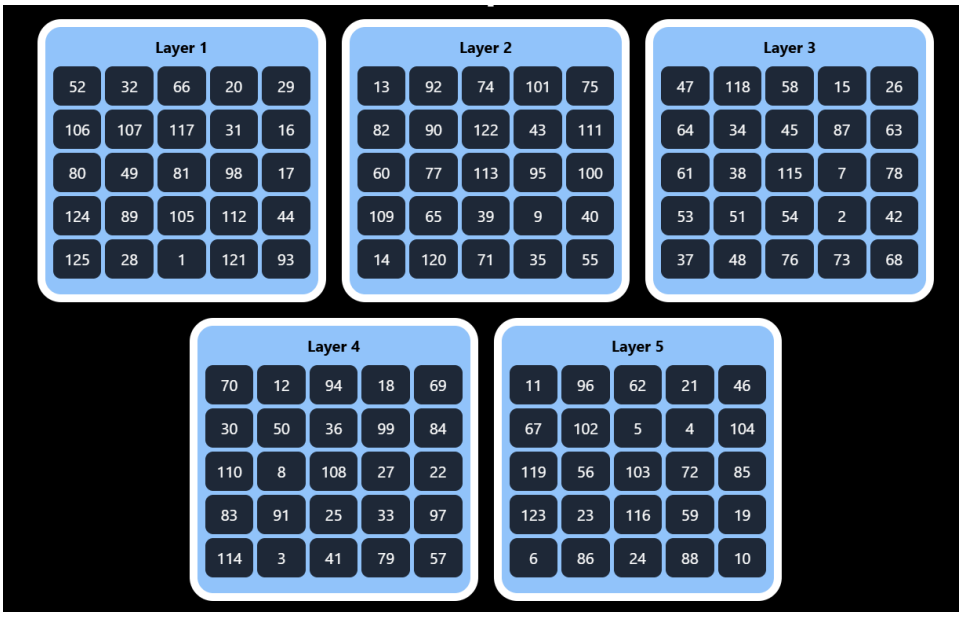
43	72	30	118	14
9	38	77	13	104
96	28	74	110	103
69	33	125	93	4
65	6	60	88	59

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

State Akhir	
Objective Function	Objective Function: 548406 

	
Durasi	Durasi: 0.0012337 seconds

h. Eksperimen 8

State Awal	

Layer 1

101	79	87	69	36
33	125	40	43	75
73	90	20	39	71
2	38	5	85	120
92	80	108	122	96

Layer 2

64	59	23	42	119
49	74	102	54	82
57	113	109	104	72
9	46	118	121	12
123	83	115	25	13

Layer 3

22	110	35	7	29
31	98	86	47	93
24	114	21	65	88
41	53	103	56	77
34	116	14	111	51

Layer 4

89	94	45	16	100
11	19	117	37	26
48	15	44	63	62
32	76	55	68	124
81	60	28	91	99

Layer 5

3	18	27	8	70
105	107	10	66	1
106	67	97	17	112
4	84	50	6	30
52	58	95	78	61

Layer 1

6	56	12	27	28
90	44	118	97	94
40	3	124	115	73
123	29	87	63	16
120	15	78	52	5

Layer 2

32	75	13	79	108
91	71	114	39	121
21	53	57	18	80
83	30	50	55	66
88	42	77	34	9

Layer 3

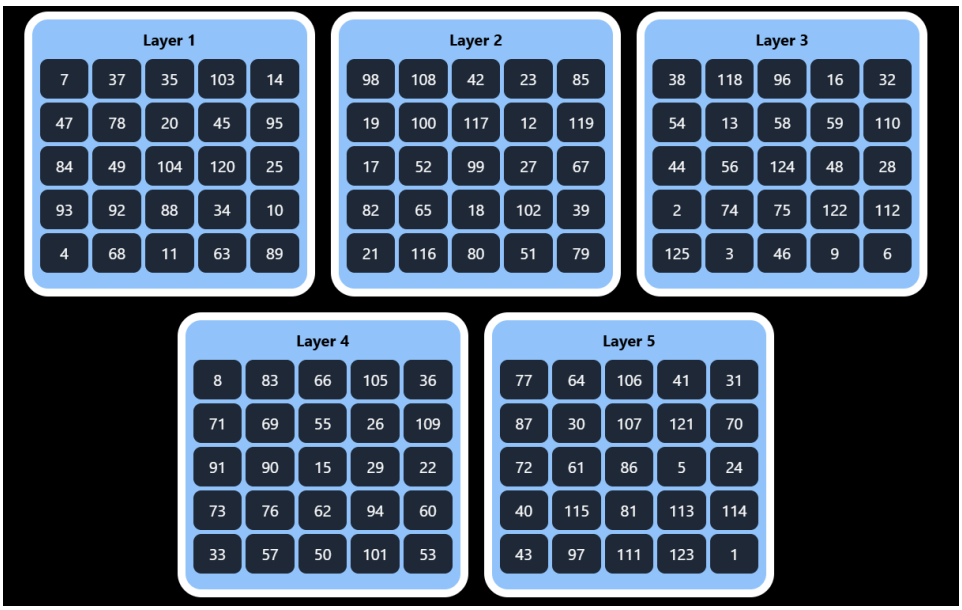
20	113	33	84	41
62	10	46	107	102
81	60	22	122	68
47	89	25	109	116
69	98	100	70	82

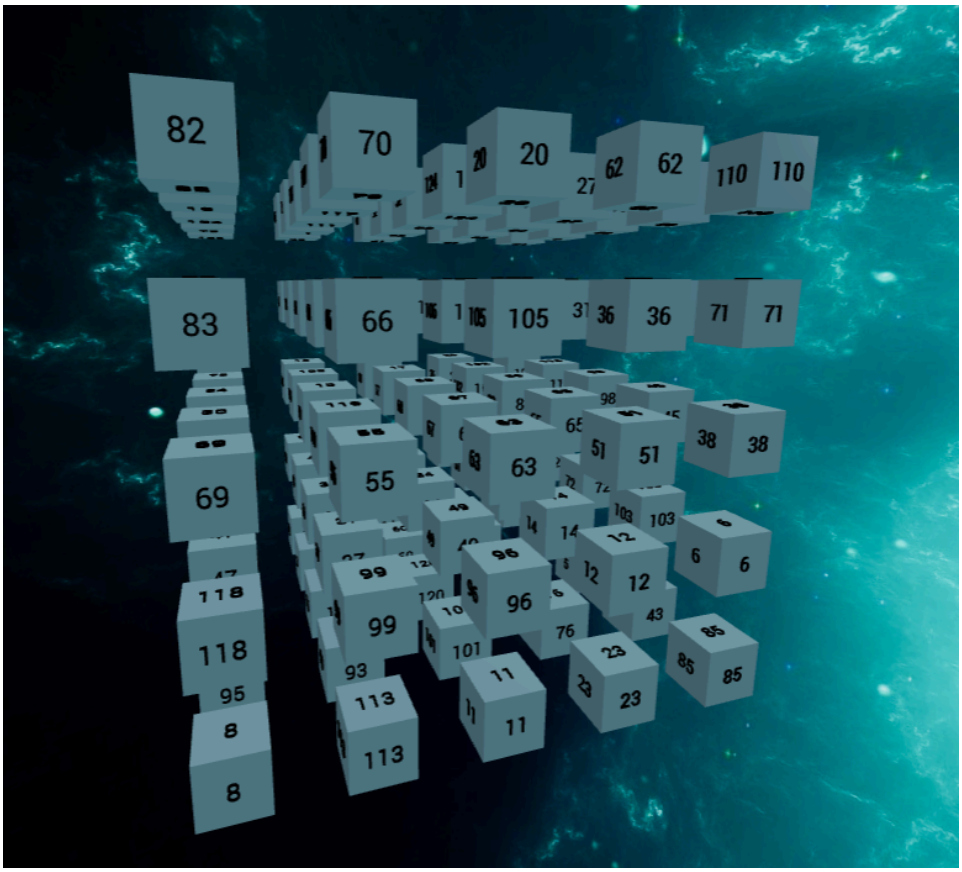
Layer 4

37	11	61	59	24
86	51	49	14	8
19	58	106	85	31
103	23	125	26	2
95	36	93	119	43

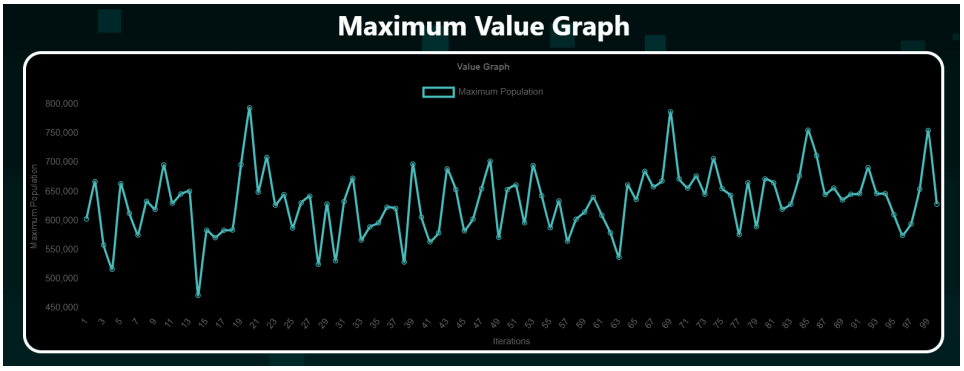
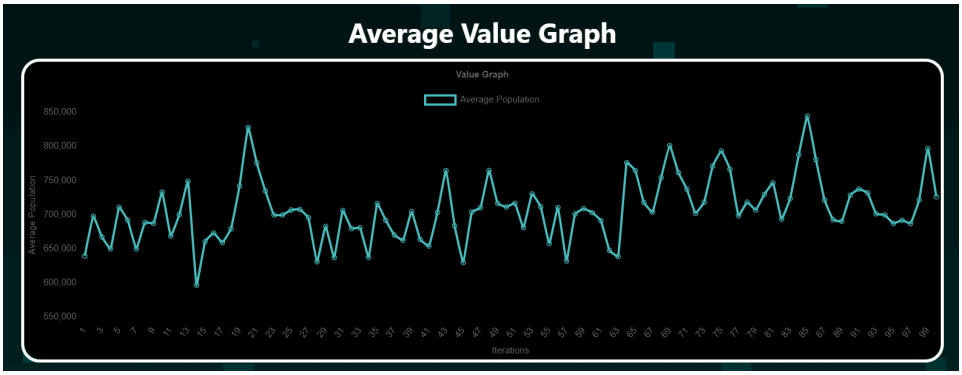
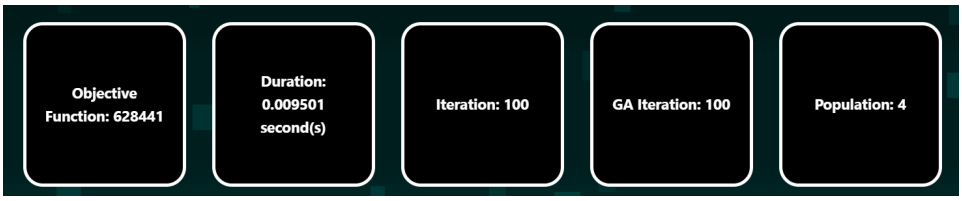
Layer 5

110	38	104	48	111
54	65	67	1	92
4	64	45	7	99
112	105	74	35	76
101	96	17	117	72

<table>
<tr>
<td></td>
<td>


</td>
</tr>
<tr>
<td>State Akhir</td>
<td>


</td>
</tr>
<tr>
<td>Objective Function</td>
<td>Objective Function: 628441</td>
</tr>
</table>

	 
Durasi	Durasi: 0.009501 

i. Eksperimen 9

State Awal	

Layer 1

71	31	96	122	42
38	65	69	66	9
57	91	14	11	76
21	44	68	94	43
118	67	62	12	36

Layer 2

23	8	22	26	100
110	63	30	120	7
49	33	119	97	17
116	53	92	39	106
46	47	99	70	15

Layer 3

93	77	54	89	95
123	81	101	48	86
114	34	52	13	20
18	108	64	104	2
124	85	51	50	83

Layer 4

61	121	40	55	115
73	16	74	37	98
19	80	105	84	107
103	56	60	27	78
4	5	88	75	29

Layer 5

45	79	90	113	3
59	35	24	87	112
10	117	102	25	41
72	111	1	58	82
109	6	32	28	125

Layer 1

68	64	15	42	24
40	71	41	55	80
85	112	113	110	43
106	38	99	27	48
30	45	86	78	20

Layer 2

59	101	11	69	12
1	100	98	21	115
105	60	107	10	5
35	79	96	102	104
57	56	122	4	120

Layer 3

83	61	92	13	37
95	19	8	33	75
47	103	62	121	74
82	97	51	53	76
44	70	66	50	84

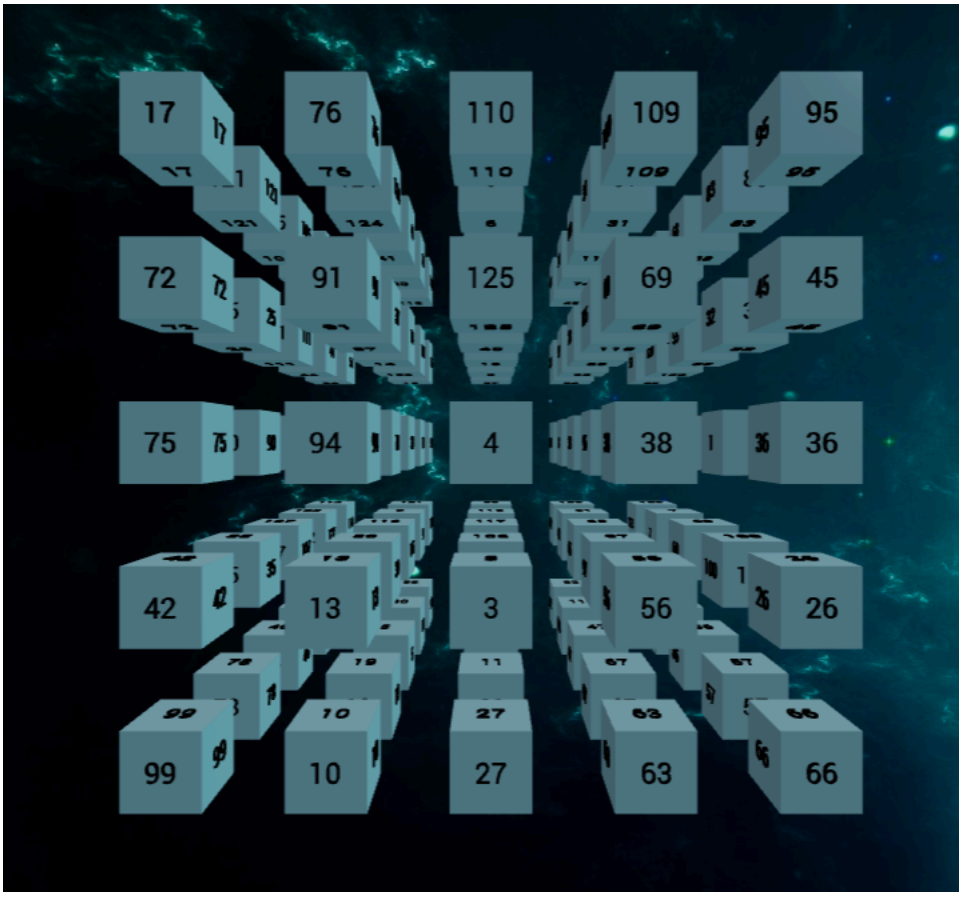
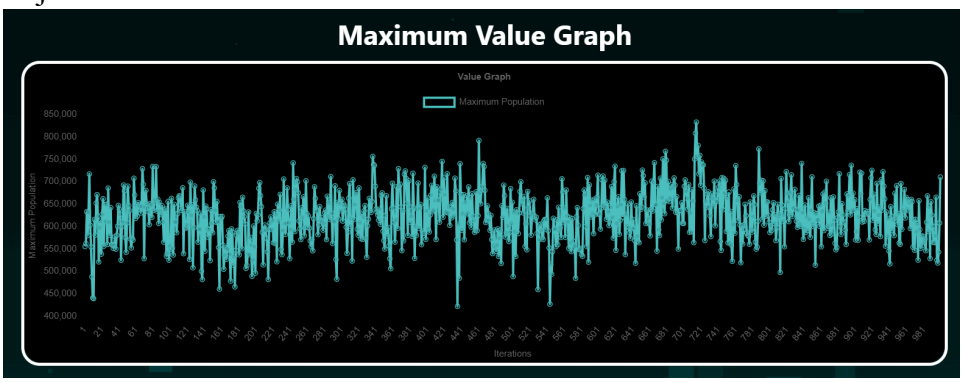
Layer 4

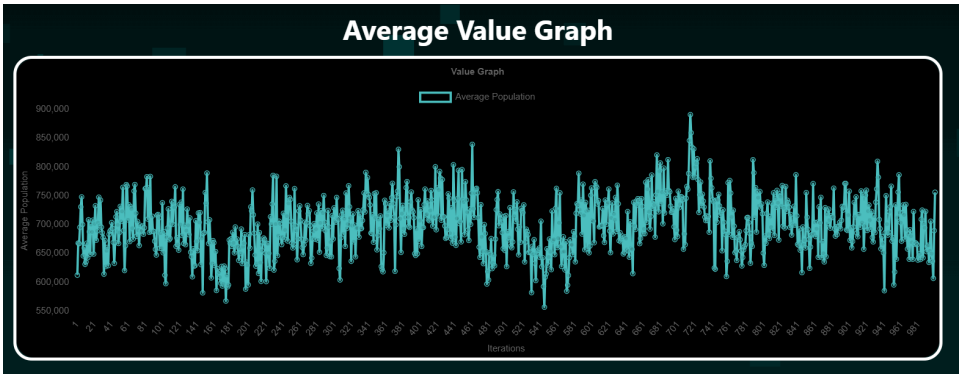
87	6	18	9	109
23	49	58	91	29
124	114	31	34	2
3	54	77	88	63
89	73	72	14	25

Layer 5

7	90	17	22	67
123	125	26	111	94
65	39	46	81	32
108	118	117	28	116
119	36	16	52	93

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

State Akhir	
Objective Function	Objective Function: 710500 

	
Durasi	Durasi: 0.0916769

Bab 4 - Analisis dan Pembahasan

Jenis Algoritma	Best State Value	Average Time (seconds)
Steepest Ascent Hill-Climbing	244	1.657
Sideways Move Hill-Climbing	284 (10 max sideway)	1.8059 (10 max sideway)
Random Restart Hill-Climbing	226 (5 restarts) 180 (10 restarts) 111 (100 restarts)	8.7264 (5 restarts) 16.16345 (10 restarts) 161.754 (100 restarts)
Stochastic Hill-Climbing	3021	0.0117
Simulated Annealing	86	0.8243
Genetic Algorithm	531635	0.0236

Dapat dilihat dari data di atas, bahwa tiap algoritma Local Search susah untuk mendapatkan global optimum. Hasil yang didapat hanya memiliki objective function yang berkisar di angka ratusan dan sangat sulit untuk mendapatkan angka objective di angka puluhan. Menurut kami, hal ini terjadi karena sifat random Magic Cube 5x5x5 yang besar sehingga susah untuk di handle dengan Local Search. Magic Cube 5x5x5 memiliki ruang solusi yang sangat besar sehingga susah untuk ditelusuri secara menyeluruh oleh algoritma Local Search. Hal ini juga dikarenakan ruang solusi yang harus memenuhi banyak kondisi simultan, seperti baris, kolom, tiang, diagonal bidang, dan diagonal ruang yang semuanya harus berjumlah sama dengan magic number.

Berdasarkan data hasil dari tiap algoritma Local Search, kami menyimpulkan bahwa algoritma terbaik untuk permasalahan Magic Cube ini adalah dengan menggunakan Simulated Annealing. Hal ini dibuktikan dengan Simulated Annealing sebagai satu-satunya algoritma yang menghasilkan objective function dibawah nilai 100. Hal ini karena sifat Simulated Annealing yang fleksibel dalam menghadapi local optima, dan mampu dalam mencari solusi secara lebih luas pada awal dan mempersempit pencarian menuju solusi optimal seiring waktu. Pencarian ini juga didukung dengan fungsi schedule (exponential value schedule) yang berkurang secara pelan dan tepat, sehingga pencarian dapat menyeluruh dengan mendukung random walk pada awal pencarian.

Sedangkan algoritma yang memberikan hasil terburuk adalah Stochastic Hill-Climbing dan Genetic Algorithm. Stochastic Hill-Climbing memberikan nilai objective function di angka

ribuan, dapat dilihat pada data eksperimen yang menghasilkan best objective value 3021 dalam 3 eksperimen. Hal ini terjadi karena sifat Stochastic yang menghasilkan suksesor random dan mengambil sifat Steepest Ascent yang tidak mendukung sideways move. Sifat-sifat ini membuat stochastic susah untuk bergerak menuju global minima dan tidak dapat keluar dari local optima. Sifat pengambilan random dan tidak mampu keluar dari local optima ini yang menjadi alasan dari hasil Stochastic yang buruk.

Untuk Genetic Algorithm, hasil algoritma ini sangat ditentukan oleh jumlah populasi dan maksimal iterasi yang bisa dilakukan. Hasil buruk algoritma ini pada eksperimen kami mungkin dikarenakan jumlah iterasi dan jumlah populasi yang kami tentukan terlalu kecil sehingga tidak memaksimalkan sifat sharing info antar individu state. Agar mendapatkan hasil yang lebih baik, nilai dari jumlah populasi dan maksimal iterasi harus dibuat besar sehingga dapat menghasilkan objective function yang lebih baik. Alasan lain adalah masalah *prematur convergence*, dimana populasi solusi menjadi terlalu seragam terlalu cepat, sehingga kehilangan keragaman individu yang diperlukan untuk menjelajahi ruang solusi yang lebih luas. Hal ini menyebabkan algoritma ini terjebak dalam local optima yang jauh dari solusi global minima.

Dari aspek durasi proses pencarian, algoritma yang menggunakan Highest Valued Successor umumnya lebih lambat dibandingkan algoritma lain yang menggunakan Random Successor sebagai fungsi pembangkit suksesornya. Hal ini terjadi karena Highest Valued Successor yang menelusuri semua kemungkinan suksesor dari current state, untuk tiap iterasi pencarian. Proses iterasi dan pencarian ini sangat berat apalagi ditambah dengan perhitungan objective function dari tiap-tiap suksesor state.

Berdasarkan durasi pencarian, algoritma yang memberikan hasil objective function terbaik dengan waktu yang tercepat adalah Simulated Annealing. Hal ini karena Simulated Annealing yang memaksimalkan proses pencarian semua kemungkinan state, dibandingkan memfokuskan pencarian suksesor terbatas hanya pada suksesor-suksesor dari current state seperti pada Highest Valued Successor. Algoritma berikut yang juga cepat adalah Stochastic Hill Climbing. Algoritma ini cepat karena pembangkitan suksesor yang menggunakan Random Successor. Durasi dari algoritma ini juga sangat bergantung pada max iterasi yang dilakukan. Max iterasi yang semakin besar akan menghasilkan durasi pencarian yang besar pula, dan sebaliknya.

Berdasarkan konsistensi algoritma, algoritma yang memberikan hasil konsisten adalah Steepest Ascent, Sideway Move, Random Restart, dan Simulated Annealing. Algoritma-algoritma ini (selain Simulated Annealing) menghasilkan hasil yang konsisten karena sifatnya yang membangkitkan suksesor dengan Highest Valued Successor, sehingga flow pencarian konsisten dalam menuju global optima. Sedangkan untuk Simulated Annealing, algoritma ini menghasilkan hasil yang konsisten karena fungsi scheduling yang menurun secara pelan,

sehingga pencarian solusi dapat dilakukan secara menyeluruh walaupun menggunakan konsep Random Successor

Jumlah Populasi berpengaruh terhadap solusi pencarian, dimana semakin tinggi jumlah populasi akan menghasilkan hasil pencarian yang bagus, dan sebaliknya. Hal ini karena dengan semakin banyak individu yang melakukan sharing informasi saat iterasi pencarian, pencarian ruang solusi dapat dilakukan lebih menyeluruh. Sedangkan untuk parameter jumlah iterasi, semakin banyak maksimal jumlah iterasi akan menghasilkan hasil pencarian yang bagus, dan sebaliknya. Hal ini karena proses iterasi yang semakin banyak akan menghasilkan jumlah individu yang semakin banyak dan tiap keberadaan individu melambangkan satu bagian dari ruang solusi, sehingga ruang solusi dapat ditelusuri lebih luas dibandingkan jumlah individu yang lebih kecil.

Bab 5 - Kesimpulan dan Saran

Berdasarkan hasil analisis kami, dapat disimpulkan dari seluruh algoritma local search yang digunakan, algoritma yang terbaik adalah algoritma **Simulated Annealing**. Simulated Annealing mampu menghasilkan nilai objective function di bawah 100, yang secara signifikan lebih baik dibandingkan dengan algoritma lain. Keberhasilan ini disebabkan oleh mekanisme **cooling schedule** yang secara bertahap mengurangi temperatur, memungkinkan algoritma ini untuk menjelajahi ruang solusi yang luas pada awalnya dan kemudian semakin fokus mendekati solusi optimal seiring waktu. Selain itu, fleksibilitas Simulated Annealing dalam menghadapi local optima sangat membantu mengatasi kompleksitas permasalahan magic cube. Di sisi lain, **Stochastic Hill-Climbing** dan **Genetic Algorithm** menunjukkan hasil yang kurang memuaskan dengan nilai objective function yang berada pada kisaran ribuan.

Saran:

- Mengingat keterbatasan Local Search, sebaiknya mencoba metode lain yang lebih mampu menangani masalah dengan ruang solusi besar, seperti algoritma berbasis **Deep Learning**, yang dapat memberikan optimasi dengan ruang solusi yang lebih besar.
- Algoritma seperti Simulated Annealing dan Genetic Algorithm sangat sensitif terhadap parameter seperti cooling rate dan mutation rate. Dengan melakukan eksperimen parameter tuning dengan lebih intensif bisa membantu untuk mencapai hasil yang lebih optimal dan stabil. Diperlukan penentuan schedule yang tepat untuk permasalahan ini, sehingga dapat memberikan hasil yang paling optimal dan efisien.

Bab 6 - Pembagian Kerja

NIM	**Pembagian Kerja**
13522029	Algoritma BackEnd
13522043	FrontEnd + Integrasi
13522054	Algoritma BackEnd
13522110	FrontEnd + Random Restart

Bab 7 - Referensi

- Bergman, M., & Montanari, A. (2007). "Parallel tempering for the glass transition in spin glasses". Physical Review E, 75(5), 051126.
- Wikipedia contributors. (n.d.). Crossover (genetic algorithm). In Wikipedia, The Free Encyclopedia. Retrieved October 2, 2024, from https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)
- Weisstein, E. W. (n.d.). Perfect magic cube. In Wolfram MathWorld. Retrieved October 2, 2024, from https://mathworld.wolfram.com/PerfectMagicCube.html
- Trump, K. (n.d.). Magic cubes: Overview of 1x1 to 5x5 magic cubes. Retrieved October 2, 2024, from https://www.trump.de/magic-squares/magic-cubes/cubes-1.html