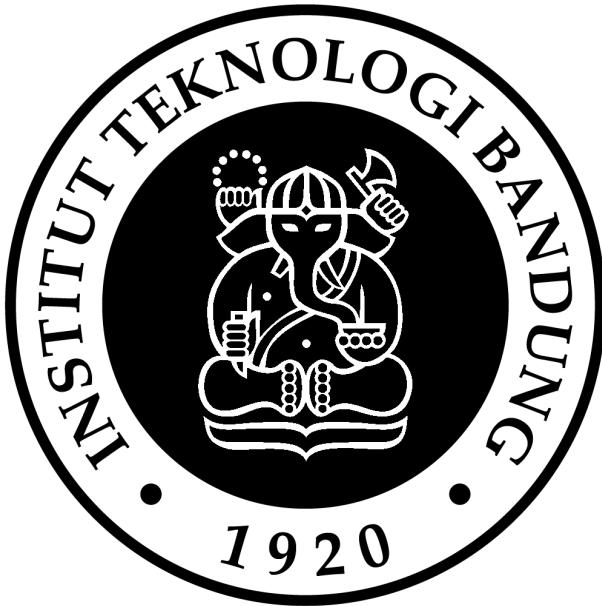


**Laporan Tugas Besar 1 IF3170 Inteligensi Artifisial  
Pencarian Solusi *Diagonal Magic Cube* dengan *Local  
Search***

**Semester I Tahun 2024/2025**



**Disusun Oleh :**

Maulvi Ziadinda Maulana	13522122
Muhammad Dzaki Arta	13522149
Albert Ghazaly	13522150
Muhammad Rasheed Qais Tandjung	13522158

**INSTITUT TEKNOLOGI BANDUNG  
2024**

## I. Deskripsi Persoalan

Diagonal magic cube merupakan kubus yang tersusun dari angka 1 hingga  $n^3$  tanpa pengulangan dengan  $n$  adalah panjang sisi pada kubus tersebut. Dengan menerapkan algoritma-algoritma local search untuk menyusun angka-angka pada sebuah kubus berukuran 5x5x5 **Diagonal Magic Cube**. Kubus ini disusun menggunakan angka 1 hingga  $n^3$  (125 untuk  $n=5$ ) tanpa pengulangan, dengan persyaratan bahwa setiap baris, kolom, tiang, dan seluruh diagonal harus memiliki jumlah yang sama, yang disebut **magic number**.

### Spesifikasi Masalah

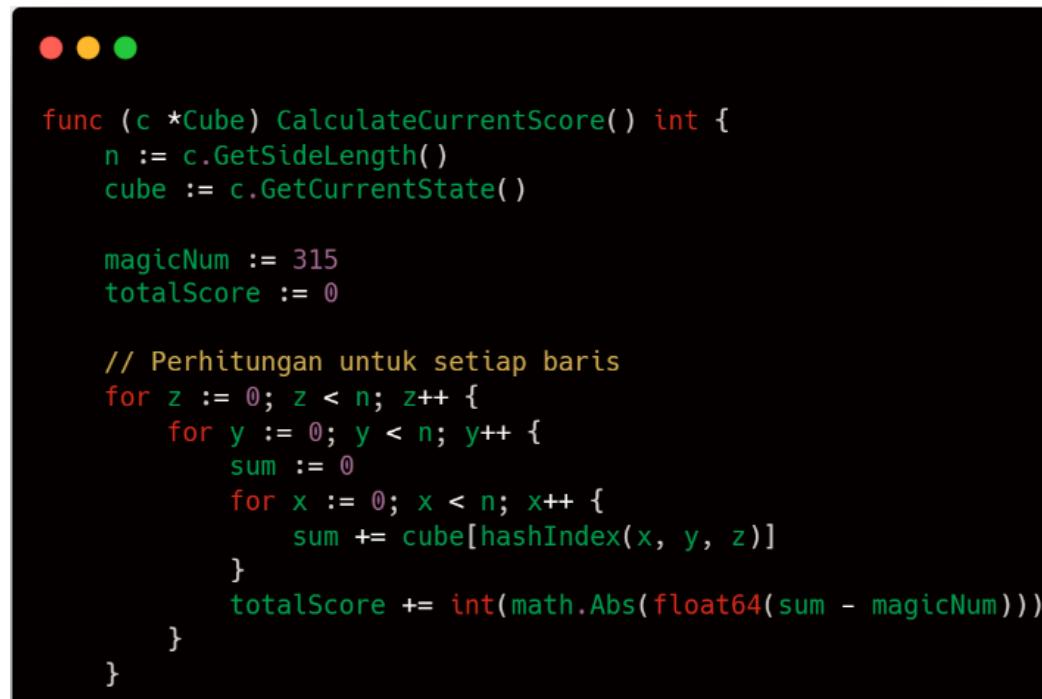
1. **Diagonal Magic Cube:** Kubus ini memiliki properti khusus di mana setiap deret angka dalam baris, kolom, tiang, serta seluruh diagonalnya harus memiliki jumlah yang sama dengan magic number. Magic number ini tidak harus termasuk dalam rentang angka pada kubus (1 hingga 125) dan bukan bagian dari angka-angka yang tersusun dalam kubus tersebut.
2. **Implementasi Algoritma Local Search:** Dalam program ini, kami telah mengimplementasikan tiga algoritma local search sebagai solusi terhadap masalah Diagonal Magic Cube, yaitu:
  - a. **Hill-Climbing Steepest Ascent**
  - b. **Hill-Climbing Sideways Move**
  - c. **Hill-Climbing Stochastic**
  - d. **Random-Restart Hill Climbing**
  - e. **Simulated Annealing**
  - f. **Genetic Algorithm.**
3. **Eksperimen dan Pengumpulan Data:** Setiap algoritma dijalankan sebanyak tiga kali dengan parameter tertentu, dan data seperti state awal dan akhir dari kubus, nilai akhir dari objective function, durasi pencarian solusi, serta grafik perubahan nilai objective function terhadap iterasi dicatat untuk dianalisis. Hasil ini digunakan untuk mengevaluasi efektivitas masing-masing algoritma dalam menyelesaikan masalah Diagonal Magic Cube.
4. **Visualisasi dan Analisis Hasil:** Program ini mampu memvisualisasikan state awal dan akhir kubus, serta grafik yang menggambarkan hasil eksperimen. Berdasarkan data yang diperoleh, dilakukan analisis mendalam untuk mengevaluasi kemampuan masing-masing algoritma dalam mendekati global optimum, konsistensi hasil, serta perbandingan durasi pencarian antar-algoritma.
5. **Spesifikasi Tambahan (Bonus):** Selain program dapat melakukan seluruh algoritma hill-climbing. Program juga dapat melakukan fitur replay dalam bentuk “video player” untuk memvisualisasikan proses pencarian solusi pada setiap tahap iterasi. Fitur ini mencakup fungsi play/pause, progress bar yang dapat diatur, dan pengaturan kecepatan.

## II. Pembahasan

### A. Pemilihan *Objective Function*

Fungsi `CalculateCurrentScore()` berfungsi sebagai objective function yang memberikan nilai pada susunan saat ini. Dengan cara perhitungan skornya adalah:

1. **Setiap Baris, Kolom, dan Tiang:** Untuk setiap baris, kolom, dan tiang dalam kubus, fungsi menghitung selisih antara jumlah angka pada garis tersebut dengan magic number (315). Semakin kecil selisih ini, semakin mendekati kondisi ideal. Nilai absolut dari selisih ini ditambahkan ke dalam `totalScore`.
2. **Diagonal Ruang:** Fungsi juga mempertimbangkan empat diagonal utama yang melintasi seluruh kubus. Setiap diagonal dihitung secara independen, dan selisih antara jumlah angka pada diagonal dengan magic number dikontribusikan ke total skor.
3. **Diagonal Bidang:** Untuk setiap potongan bidang pada kubus, dua diagonal dihitung pada masing-masing bidang, dan selisihnya dengan magic number juga ditambahkan ke dalam skor.
4. **Hasil Akhir:** `totalScore` dihitung sebagai jumlah dari semua selisih absolut antara jumlah setiap garis dan magic number. Nilai ini dibalikkan dengan tanda negatif, sehingga semakin mendekati nol (atau negatif kecil) berarti kubus berada dalam kondisi yang semakin optimal.



```
func (c *Cube) CalculateCurrentScore() int {
    n := c.GetSideLength()
    cube := c.GetCurrentState()

    magicNum := 315
    totalScore := 0

    // Perhitungan untuk setiap baris
    for z := 0; z < n; z++ {
        for y := 0; y < n; y++ {
            sum := 0
            for x := 0; x < n; x++ {
                sum += cube[hashIndex(x, y, z)]
            }
            totalScore += int(math.Abs(float64(sum - magicNum)))
        }
    }
}
```

```

// Perhitungan untuk setiap kolom
for z := 0; z < n; z++ {
    for x := 0; x < n; x++ {
        sum := 0
        for y := 0; y < n; y++ {
            sum += cube[hashIndex(x, y, z)]
        }
        totalScore += int(math.Abs(float64(sum - magicNum)))
    }
}

// Perhitungan untuk setiap tiang
for y := 0; y < n; y++ {
    for x := 0; x < n; x++ {
        sum := 0
        for z := 0; z < n; z++ {
            sum += cube[hashIndex(x, y, z)]
        }
        totalScore += int(math.Abs(float64(sum - magicNum)))
    }
}

// Perhitungan untuk seluruh diagonal ruang
for i := 0; i < n; i++ {
    sum1 := 0
    sum2 := 0
    sum3 := 0
    sum4 := 0
    for j := 0; j < n; j++ {
        sum1 += cube[hashIndex(j, j, j)]
        sum2 += cube[hashIndex(j, j, n-j-1)]
        sum3 += cube[hashIndex(j, n-j-1, j)]
        sum4 += cube[hashIndex(n-j-1, j, j)]
    }
    totalScore += int(math.Abs(float64(sum1 - magicNum)))
    totalScore += int(math.Abs(float64(sum2 - magicNum)))
    totalScore += int(math.Abs(float64(sum3 - magicNum)))
    totalScore += int(math.Abs(float64(sum4 - magicNum)))
}

// Perhitungan untuk seluruh diagonal bidang pada kubus
for z := 0; z < n; z++ {
    sum1 := 0
    sum2 := 0
    sum3 := 0
    sum4 := 0
    for i := 0; i < n; i++ {
        sum1 += cube[hashIndex(i, i, z)]
        sum2 += cube[hashIndex(i, n-i-1, z)]
        sum3 += cube[hashIndex(z, i, i)]
        sum4 += cube[hashIndex(z, i, n-i-1)]
    }
}

```

```
        totalScore += int(math.Abs(float64(sum1 - magicNum)))
        totalScore += int(math.Abs(float64(sum2 - magicNum)))
        totalScore += int(math.Abs(float64(sum3 - magicNum)))
        totalScore += int(math.Abs(float64(sum4 - magicNum)))
    }

    return -totalScore
}
```

## B. Implementasi Local Search

Untuk menjelaskan mekanisme dari pencarian solusi *magic cube* menggunakan *local search*, perlu dijelaskan terlebih dahulu terkait struktur data berupa kelas-kelas utama yang digunakan untuk permasalahan, berupa kelas *cube* dan kelas *solution*.

### 1. Kelas Cube

```
type Cube struct {
    currentState []int
    sideLength   int
    blockCount   int
    currentScore int
}
```

Kelas ini merepresentasikan objek utama pada persoalan, yaitu *magic cube* yang sedang ditinjau. Kelas ini menyimpan nilai-nilai kubus kecil pada atribut list *currentState*, yang memiliki ukuran  $5 \times 5 \times 5 = 125$ , sesuai dengan spesifikasi persoalan.

Selain itu, kelas ini juga menyimpan *sideLength* kubus (bernilai 5 pada kasus ini), jumlah kubus kecil (*blockCount*) pada kubus (125 pada kasus ini), serta nilai *objective function* dari instansiasi kubus saat ini pada atribut *currentScore*.

### 2. Kelas Solution

```
type Solution struct {
    Type      string          `json:"type"`
    Solution  []SolutionItem `json:"solutions"`
    AdditionalInfo []AdditionalInfo `json:"additionalInfo"`
}
```

Selain itu, terdapat kelas *Solution* yang menyimpan solusi penyelesaian dari sebuah algoritma. Solusi ini berupa sebuah *list* yang menyimpan informasi kubus pada setiap iterasi algoritma. *List* solusi ini disimpan pada atribut *Solution*, yang merupakan *list of SolutionItem*.

Juga terdapat atribut *Type* yang menyimpan nama dari algoritma yang menyelesaikan, serta atribut *AdditionalInfo* untuk menyimpan informasi tambahan solusi untuk setiap iterasi.

```
type SolutionItem struct {
    State      []int `json:"state"`
    Iteration   int  `json:"iteration"`
    Score       int  `json:"score"`
    Probability float64 `json:"probability"`
}
```

Sebuah SolutionItem merupakan solusi dari sebuah iterasi pada algoritma. Kelas ini menyimpan nilai dari setiap kubus kecil di kubus pada atribut State, nomor iterasi pada atribut Iteration, nilai *objective function* dari solusi pada atribut Score, serta atribut tambahan untuk kebutuhan algoritma *simulated annealing* yaitu atribut Probability.

```
type addtitionalInfor struct {
    // Additional info for genetic algorithm
    AvgScore int `json:"avgScore"`
}
```

Kelas Solution juga menyimpan *list of AdditionalInfo* untuk kebutuhan tambahan sebuah algoritma, contohnya untuk algoritma *genetic algorithm* yang membutuhkan informasi rata-rata *objective function* untuk setiap iterasi algoritma.

### 3. Hill-Climbing Steepest Ascent

```
func HillClimbingSteepest() class.Solution {
    currentCube := class.NewCube(5)
    currentCube.SetRandomStartState()
    currentScore := currentCube.GetCurrentScore()

    res := class.NewSolution()
    res.SetType("Steepest Ascent Hill Climbing")
    res.AddSolutionItem(0, currentScore,
    currentCube.GetCurrentState())

    i := 1

    for {
        bestSuccessor := currentCube.GetBestSuccessor()
        bestSuccessorScore := bestSuccessor.GetCurrentScore()
```

```

        if bestSuccessorScore > currentScore {
            currentCube = bestSuccessor.CopyCube()
            currentScore = bestSuccessorScore
            res.AddSolutionItem(i, currentScore,
currentCube.GetCurrentState())
            i++
        } else {
            break
        }
    }

    return *res
}

```

Algoritma ini akan membuat sebuah kubus awal dan mengalokasikan sebuah state random pada kubus menggunakan metode SetRandomStartState. Algoritma lalu akan melakukan pengulangan, yang pada setiap iterasi akan diambil successor dengan skor tertinggi melalui method GetBestSuccessor.

Jika didapat bahwa successor tertinggi memiliki nilai yang lebih rendah dari state saat ini, maka sudah disampaikan *local maximum* dan algoritma akan melakukan terminasi.

#### 4. Hill-Climbing Sideways Move

```

func HillClimbingSideways(maxIter int) class.Solution {
    currentCube := class.NewCube(5)
    currentCube.SetRandomStartState()
    currentScore := currentCube.GetCurrentScore()

    res := class.NewSolution()
    res.GetType("Hill Climbing with Sideways Move")
    res.AddSolutionItem(0, currentScore,
currentCube.GetCurrentState())

    i := 1

    maxCheck := 0

    for maxCheck != maxIter {

```

```

        bestSuccessor := currentCube.GetBestSuccessor()
        bestSuccessorScore := bestSuccessor.GetCurrentScore()

        if bestSuccessorScore == currentScore {
            maxCheck++
        }

        if bestSuccessorScore >= currentScore {
            currentCube = bestSuccessor.CopyCube()
            currentScore = bestSuccessorScore
            res.AddSolutionItem(i, currentScore,
            currentCube.GetCurrentState())

            i++

        } else {
            break
        }
    }

    return *res
}

```

Algoritma ini sama seperti *steepest ascent*, perbedaannya hanya di komparasi `bestSuccessor` dengan state saat ini. Untuk *steepest ascent*, jika nilai `successor ==` nilai saat ini, maka program akan terminasi. Untuk algoritma *sideways move*, hal ini tidak akan menyebabkan terminasi.

Namun terdapat jumlah maksimum *sideways move* pada algoritma yang disimpan pada variabel `maxIter`. Jika jumlah *sideways move* yang dilakukan (yang disimpan pada variabel `maxCheck`) sudah mencapai maksimum, algoritma akan melakukan terminasi.

## 5. Hill-Climbing Stochastic

```

func HillClimbingStochastic(maxIter int) class.Solution {
    currentCube := class.NewCube(5)
    currentCube.SetRandomStartState()
    currentScore := currentCube.GetCurrentScore()

```

```

    res := class.NewSolution()
    res.SetType("Stochastic Hill Climbing")
    res.AddSolutionItem(0, currentScore,
currentCube.GetCurrentState())

    i := 1

    for i+1 != maxIter {
        randomSuccessor := currentCube.GetRandomSuccessor()
        randomSuccessorScore :=
randomSuccessor.GetCurrentScore()

        currentCube = randomSuccessor.CopyCube()
        if currentScore < randomSuccessorScore {
            currentScore = randomSuccessorScore
            res.AddSolutionItem(i, currentScore,
currentCube.GetCurrentState())
        }
        i++
    }

    return *res
}

```

Hill Climbing Stochastic adalah algoritma *hill climbing* layaknya *steepest ascent hill climbing*, tetapi successor diambil secara *random* dan dibandingkan, jika lebih besar nilai *objective function*-nya, ditukar dengan *current*. Hal tersebut dilakukan sebanyak  $n$  kali yang disimpan sebagai parameter fungsi.

## 6. Random-Restart Hill Climbing

```

func HillClimbingRandomRestart(maxRestart int) class.Solution {
    currentCube := class.NewCube(5)
    currentCube.SetRandomStartState()
    currentScore := currentCube.GetCurrentScore()
    res := class.NewSolution()
    res.SetType("Random Restart Hill Climbing")
    res.AddSolutionItem(0, currentScore,
currentCube.GetCurrentState())

    i := 1

```

```

maxIter := 0

for maxIter != maxRestart {
    bestSuccessor := currentCube.GetBestSuccessor()
    bestSuccessorScore := bestSuccessor.GetCurrentScore()

    if bestSuccessorScore <= currentScore {
        currentCube.SetRandomStartState()
        currentScore = currentCube.GetCurrentScore()
        res.AddSolutionItem(i, currentScore,
currentCube.GetCurrentState())
        maxIter++
    } else {
        currentCube = bestSuccessor.CopyCube()
        currentScore = bestSuccessorScore
        res.AddSolutionItem(i, currentScore,
currentCube.GetCurrentState())
    }

    i++
}

return *res
}

```

Random Restart Hill Climbing adalah algoritma hill climbing yang cukup unik karena bisa menghindar *local maximum* dengan memilih *state random* untuk mengulang kembali. Hal tersebut disebut *restart* dan dibatasi sebanyak  $n$  kali yang disimpan sebagai parameter fungsi.

## 7. Simulated Annealing

```

const (
    T0                  = 100
    alpha               = 0.9
    threshold           = 0.5
    TEMPERATURE_THRESHOLD = 7
)

func getCurrentTemperature(k int) float64 {
    floatK := float64(k)

```

```
        return T0 / (1 + alpha*math.Log(1+floatK))
    }

func getProbability(deltaE int, temperature float64) float64 {
    floatDeltaE := float64(deltaE)
    return math.Exp(floatDeltaE / temperature)
}

func isAcceptance(probability float64) bool {
    return probability > threshold
}

func SimulatedAnnealing() class.Solution {
    currentCube := class.NewCube(5)
    currentCube.SetRandomStartState()
    currentScore := currentCube.GetCurrentScore()

    timeStart := time.Now()

    fmt.Println("Initial Score: ", currentScore)
    res := class.NewSolution()
    res.SetType("Simulated Annealing")
    res.AddSolutionItem(0, currentScore,
currentCube.GetCurrentState())

    i := 1

    stuck := 0

    for {
        successor := currentCube.GetRandomSuccessor()
        successorScore := successor.GetCurrentScore()

        scoreDifference := successorScore - currentScore

        currentTemperature := getCurrentTemperature(i)

        if currentTemperature < TEMPERATURE_THRESHOLD ||
currentScore == 0 {
            break
        }
    }
}
```

```

        probability := getProbability(scoreDifference,
currentTemperature)
        if scoreDifference > 0 {
            currentCube = successor.CopyCube()
            currentScore = successorScore

        } else {
            stuck++

            if isAcceptance(probability) {
                currentCube = successor.CopyCube()
                currentScore = successorScore
            }
        }

        if i%10000 == 0 {
            res.AddSolutionItem(i, currentScore,
currentCube.GetCurrentState(), probability)
            fmt.Println("Iteration: ", i, " Score: ",
currentScore)
        }

        i++
    }

    if i%10000 != 0 {
        res.AddSolutionItem(i, currentScore,
currentCube.GetCurrentState())
    }

    elapsedTime := time.Since(timeStart).Milliseconds()
    res.AddElapsedTime(float64(elapsedTime))
    res.AddLastScore(currentScore)
    res.AddAdditionalInfo("Stuck in Local Optimum Frequence",
float64(stuck))

    return *res
}

```

Simulated Annealing digunakan untuk menemukan solusi yang mendekati optimal dengan menggunakan pendekatan peluang. Beberapa hal yang dibutuhkan dalam penyelesaian ini adalah sebuah nilai suhu awal , fungsi

schedule, dan acceptance probability. fungsi *schedule* kami akan menggunakan fungsi sebagai berikut.

$$T(k) = \frac{T_0}{1 + \alpha \cdot \ln(1 + k)}$$

Dengan  $\alpha$  adalah faktor pendinginan yang mengontrol laju penurunan suhu,  $k$  adalah banyak iterasi, dan  $T(k)$  adalah suhu pada iterasi ke- $k$ .

Kami memilih pendekatan penurunan suhu secara logaritmik karena pada awalnya, ketika  $k$  masih kecil, suhu turun secara perlahan sehingga memungkinkan eksplorasi solusi yang lebih banyak, menerima solusi yang lebih buruk dengan probabilitas yang lebih tinggi. Seiring meningkatnya  $k$ , suhu menurun lebih cepat, dan algoritma mulai beralih dari ke eksplorasi solusi yang lebih baik. Selain itu kami juga akan memilih nilai  $\alpha=0.9$  karena kami ingin suhu turun lebih lambat sehingga pencarian solusi akan dilakukan ke lebih banyak suksesor.

## 8. Genetic Algorithm

*Genetic algorithm* memiliki mekanisme yang berbeda dari algoritma *local search* lainnya dalam pencarian solusi *diagonal magic cube*. Algoritma ini tidak meninjau satu buah kubus saja yang dilakukan manipulasi berulang-ulang, melainkan meninjau sebuah populasi yang terdiri atas sejumlah kubus-kubus. Hal ini memperkenalkan adanya dinamika dan komunikasi antar kubus, serta memunculkan kapabilitas untuk komunikasi antar *state*.

Untuk memfasilitasi *genetic algorithm* pada pencarian solusi *diagonal magic cube*, perlu dirancang sebuah kelas Population terlebih dahulu, untuk mengatur perilaku dari sebuah populasi.

```
type Population struct {
    Cubes      []*class.Cube
    BestCubeIndex int
    AvgScore   int
    PopulationNum int
```

```

    // Attributes for fitness function calculation
    MinScore int
    MaxFitness int
}

```

Aspek utama pada kelas populasi yang diimplementasikan adalah disimpannya sebuah *list of cubes* pada atribut Cubes. Data terkait seluruh individu pada populasi disimpan disini. Atribut sisanya hanya berupa nilai-nilai yang dapat diturunkan dari populasi tersebut. Contohnya adalah atribut BestCubeIndex yang menyimpan index untuk kubus dengan score tertinggi pada populasi, serta AvgScore untuk menyimpan skor rata-rata dari populasi.

Untuk penentuan dua individu yang akan dikawinkan, dilakukan pengambilan acak secara berbobot atas populasi, dengan individu dengan *fitness function* lebih tinggi lebih mungkin untuk dipilih dan dikawinkan. Penentuan bobot ini digunakan menggunakan atribut MaxFitness yang menyimpan nilai *fitness* tertinggi pada populasi.

Untuk perhitungan *fitness function* sendiri didasarkan pada *objective function* saja, karena individu yang ingin didominasikan pada populasi adalah individu dengan objective yang tinggi. Namun perlu dilakukan normalisasi atas *objective function* tersebut, karena mekanisme pembobotan mewajibkan *fitness function* bernilai positif. Sehingga akan digunakan nilai MinScore yang berfungsi untuk menggeser seluruh nilai *objective function* agar nilai skor paling rendah berada di 0.

Untuk metode-metode yang tersedia pada kelas populasi sendiri, selain metode-metode umum berupa *getter* dan *setter* data, antara lain:

No,	Metode	Penjelasan
1.	GetWeightedRandomCube()	Mendapatkan sebuah kubus acak dari populasi. Kubus dengan nilai <i>fitness</i> lebih tinggi akan lebih mungkin untuk dipilih.
2.	NewPopulation(populationNum int) <b>returns</b> *Population	Melakukan sebuah inisiasi Population baru, dengan ukuran populasi populationNum.
3..	GeneratePopulation()	Melakukan generasi awal atas sebuah populasi. Akan dibuat sebanyak

		populationNum jumlah individu, masing-masing diberikan sebuah <i>state</i> awal random.
4.	BreedPopulation() <b>returns</b> *Population	Mengembalikan sebuah populasi baru yang merupakan hasil perkawinan atas individu-individu dari populasi awal.

Metode GeneratePopulation akan membuat populasi awal dari algoritma. Metode ini hanya sekedar membuat sejumlah kubus dengan state random dan menambahkannya pada populasi.

```

func (p *Population) BreedPopulation() *Population {
    newPopulation := NewPopulation(p.GetPopulationNum())
    SumScore := 0

    for i := 0; i < p.GetPopulationNum(); i++ {
        parent1 := p.GetWeightedRandomCube()
        parent2 := p.GetWeightedRandomCube()

        child := Crossover(parent1, parent2)
        child = Mutate(child)

        newPopulation.Cubes[i] = child

        // Sum scores to get average
        Score := child.GetCurrentScore()
        SumScore += Score

        // Set best cube of population
        if Score >
            newPopulation.GetBestCube().GetCurrentScore() {
                newPopulation.SetBestCubeIndex(i)
            }

        // Set min score of population
        if Score < newPopulation.GetMinScore() {
            newPopulation.SetMinScore(Score)
        }
    }
}

```

```

        // Set average score of population
        newPopulation.SetAvgScore(SumScore /
newPopulation.GetPopulationNum())

        // Set max fitness of population
        maxFitness :=
newPopulation.GetBestCube().GetCurrentScore() -
newPopulation.GetMinScore()
        newPopulation.SetMaxFitness(maxFitness)

        return newPopulation
}

```

Metode BreedPopulation akan menghasilkan sebuah populasi baru berdasarkan populasi sebelumnya. Pembuatan populasi baru ini dilakukan dengan melakukan iterasi sebanyak populationNum, dan pada setiap iterasi dibuat sebuah *child state* yang merupakan hasil perkawinan dari dua buah state yang dipilih secara acak dari populasi sebelumnya.

Pada tahap pembuatan anak, terdapat dua tahap utama, yaitu tahap crossover dan tahap mutasi.

```

func Crossover(HigherHalfParent, LowerHalfParent *class.Cube)
*class.Cube {
    // Create new cube
    child := class.NewCube(5)

    // Get random index to split the DNA
    splitIndex := WeightedRandom(125)

    // Create map of swapped indices
    swappedIndices := make(map[int]bool)
    for i := splitIndex; i < 125; i++ {
        swappedIndices[i] = false
    }

    // Combine DNA of parents
    for i := 0; i < 125; i++ {
        if i < splitIndex {
            child.SetSmallCubeValue(i,
HigherHalfParent.GetSmallCubeValue(i))
        } else {
            child.SetSmallCubeValue(i,
LowerHalfParent.GetSmallCubeValue(i))
        }
    }
}

```

```

    } else {
        // Check if i has been swapped
        if swappedIndices[i] {
            continue
        }

        // Get values of the two parents
        HPValue :=
HigherHalfParent.GetSmallCubeValue(i) // HP = Higher Half Parent
LPValue :=
LowerHalfParent.GetSmallCubeValue(i) // LP = Lower Half Parent

        // Check if LowerParentValue in
HigherParent's second half
        HPIIndex :=
FindNumberInCube(HigherHalfParent, splitIndex, LPValue)

        // If it is, swap i with HPIIndex. Else, set
i to HPValue
        if HPIIndex != -1 &&
!swappedIndices[HPIIndex] {
            child.SetSmallCubeValue(i, LPValue)
            child.SetSmallCubeValue(HPIIndex,
HPValue)
            swappedIndices[i] = true
            swappedIndices[HPIIndex] = true
        } else {
            child.SetSmallCubeValue(i, HPValue)
        }
    }
}

child.SetCurrentScore()
return child
}

```

State *child* akan dihasilkan pada tahap *crossover* melalui hasil perkawinan dua buah *parent state*, yaitu dengan mengambil bagian atas dari *parent* pertama (*higher half parent*), dan mengambil bagian bawah dari *parent* kedua (*lower half parent*).

Akan diambil sebuah indeks acak antara 0 sampai 125, yang disimpan pada variabel splitIndex. Indeks ini akan menjadi pembatas antara bagian atas dan bagian bawah. Bagian atas, yaitu indeks sebelum splitIndex, akan mengambil nilai-nilai dari *higher half parent*.

Bagian bawah, yaitu indeks setelah splitIndex, akan menggunakan nilai dari *lower half parent*. Namun terdapat *constraint* pada *magic cube*, yaitu sebuah kubus harus memiliki semua nilai antara 1 sampai 125. Sehingga proses penggabungan ini harus memperhatikan bahwa tidak terhasilkan kubus-kubus duplikat pada *child state*.

Sehingga alur untuk menentukan nilai dari sebuah kubus kecil pada *lower half* adalah sebagai berikut:

- Ambil nilai dari *lower half parent* terlebih dahulu.
- Terdapat dua kemungkinan dari nilai yang diambil:
  - Jika nilai tersebut tidak berada pada *lower half* dari *higher half parent*, maka nilai tersebut berada di *higher half* dari *child* (karena sudah dimasukkan pada tahap sebelumnya) dan akan menghasilkan nilai duplikat. Sehingga pada kasus ini, cukup gunakan nilai pada *higher half parent* di indeks tersebut.
  - Jika nilai tersebut belum ada pada *lower half* dari *higher half parent*, maka penambahan nilai tersebut ke *child* tidak akan menghasilkan duplikat. Sehingga pada kasus ini, masukkan nilai tersebut ke indeks tersebut, dan pada indeks awal di *higher half parent*, masukkan dengan nilai *higher half* pada indeks tersebut. Kedua indeks tersebut dimasukkan ke map *swappedIndices* dan tidak akan disentuh lagi oleh algoritma crossover.

```
func Mutate(cube *class.Cube) *class.Cube {
    // Get two random indices to swap
    index1 := rand.Intn(125)
    index2 := rand.Intn(125)

    // Determine if mutation will occur
    // Mutation rate is 0.3
    rand.Seed(time.Now().UnixNano())
    mutationRate := rand.Float64()

    if mutationRate < 0.3 {
        // Swap the values at the two indices
    }
}
```

```

        cube.SwitchState(index1, index2)
    }

    return cube
}

```

Tahap kedua adalah tahap mutasi. Fungsi akan melakukan generasi float acak dari 0 sampai 1. Persentase untuk terjadi mutasi dibuat 30%. Jika terjadi mutasi, maka akan terjadi penukaran acak antara dua buah kubus kecil pada kubus.

```

func GeneticAlgorithm(populationNum int, iteration int)
class.Solution {
    population := NewPopulation(populationNum)
    population.GeneratePopulation()

    timeStart := time.Now()

    res := class.NewSolution()
    res.SetType("Genetic Algorithm")
    res.AddSolutionItem(0,
population.GetBestCube().GetCurrentScore(),
population.GetBestCube().GetCurrentState())
    res.AddAdditionalInfo(strconv.Itoa(0),
float64(population.GetAvgScore()))

    for i := 1; i <= iteration; i++ {
        newPopulation := population.BreedPopulation()

        population = newPopulation
        res.AddSolutionItem(i,
population.GetBestCube().GetCurrentScore(),
population.GetBestCube().GetCurrentState())
        res.AddAdditionalInfo(strconv.Itoa(i),
float64(population.GetAvgScore()))
    }

    // Add additional info
    elapsedTime := time.Since(timeStart).Milliseconds()
    res.AddElapsedTime(float64(elapsedTime))

    res.AddLastScore(population.GetBestCube().GetCurrentScore())
}

```

```
        res.AddAdditionalInfo("Iteration count",
float64(iteration))
        res.AddAdditionalInfo("Population count",
float64(populationNum))

    return *res
}
```

Fungsi yang menjalankan algoritma *genetic algorithm* hanya sekedar menggabungkan fungsi-fungsi yang sudah disebutkan sebelumnya. Akan dilakukan pembuatan dan generasi populasi awal, dan akan dilakukan sebanyak n iterasi, dimana pada setiap iterasi akan dijalankan metode BreedPopulation. Metode ini akan membuat populasi baru dengan mengambil dua individu random dan menghasilkan sebuah *child* melalui proses *crossover* dan *mutation*, sebanyak populationNum kali.

## C. Hasil Eksperimen dan Analisis

Semua screenshot state awal dan akhir pada hasil eksperimen akan menggunakan mode *display flatten X*.

### 1. Hill-Climbing Steepest Ascent

#### a. Eksperimen 1

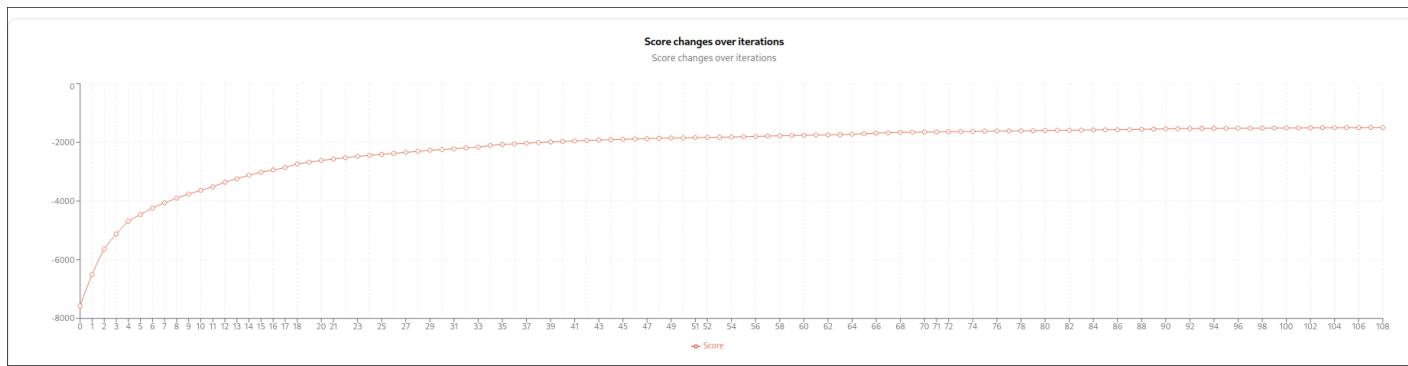
##### - State Awal

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

##### - State Akhir

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

##### - Plot Objective Function

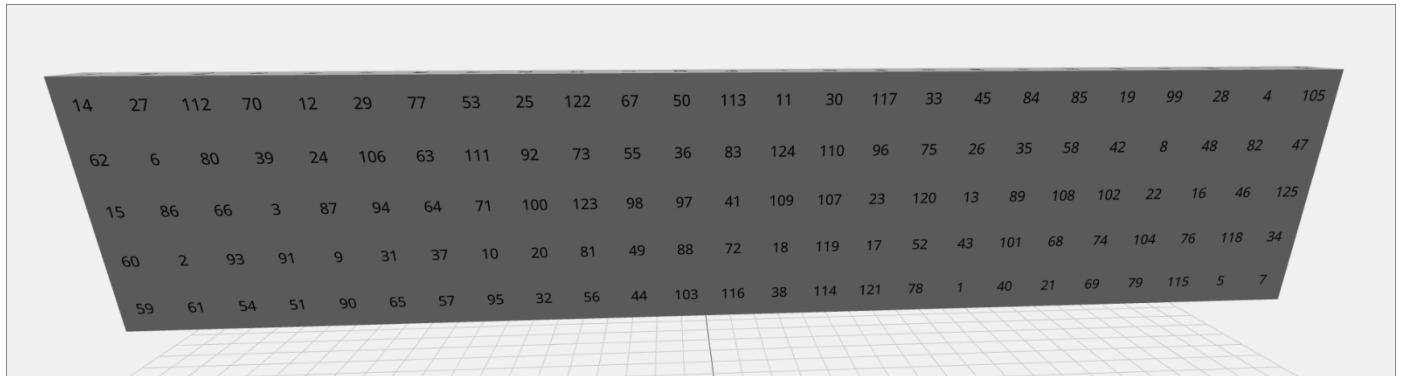


Nilai <b>objective function</b> terakhir	-1492
Durasi pencarian	570 ms

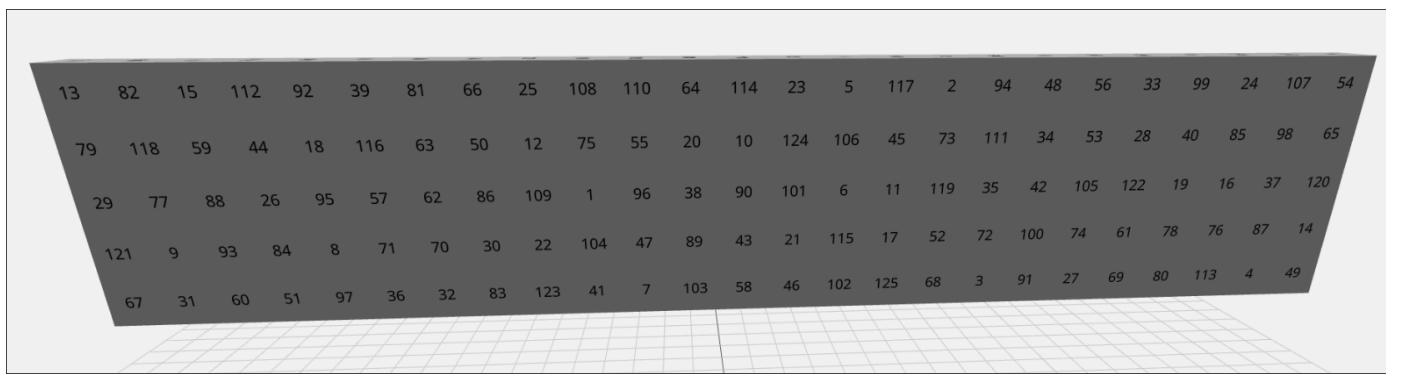
<b>Banyak iterasi</b>	108 iterasi
-----------------------	-------------

b. Eksperimen 2

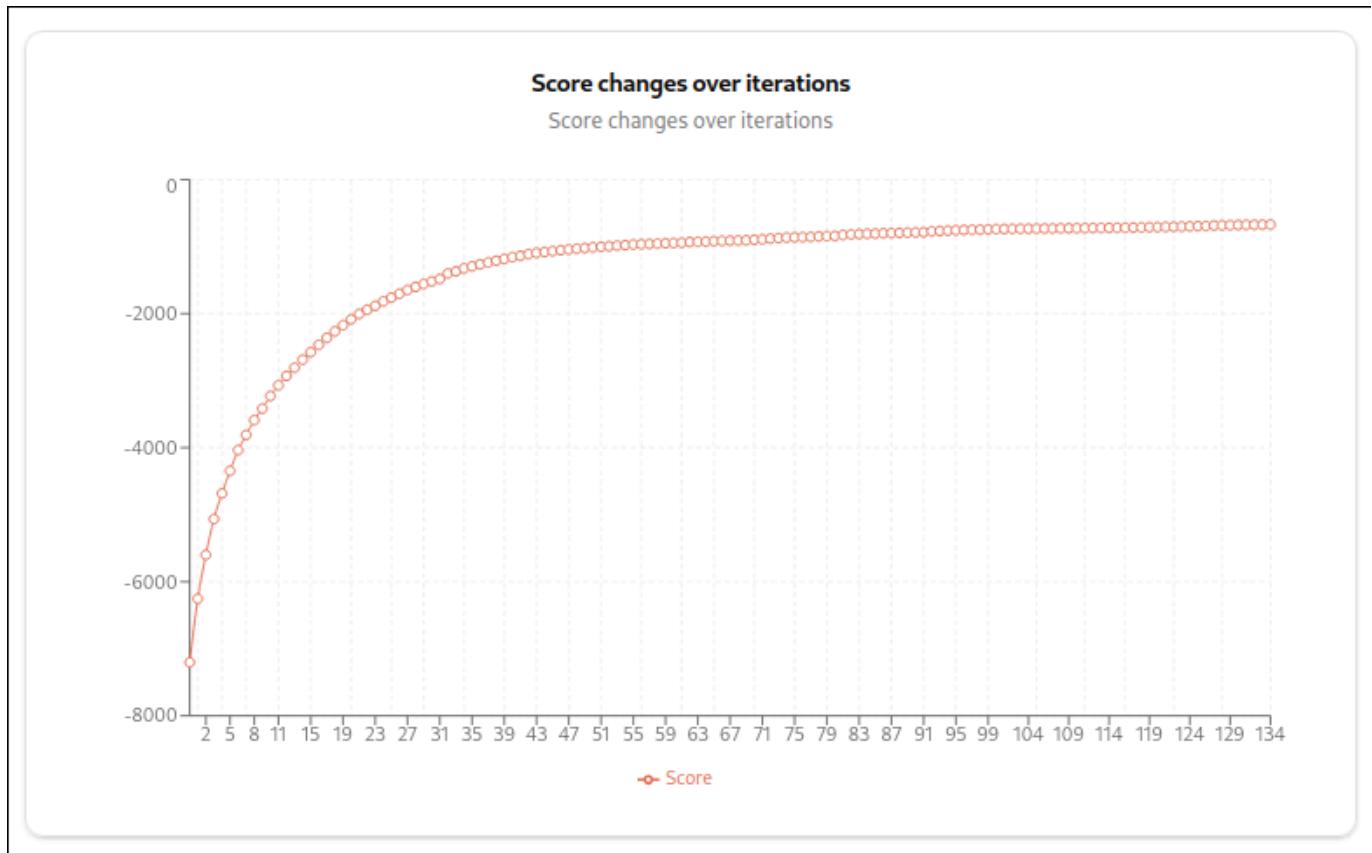
- State Awal



- State Akhir



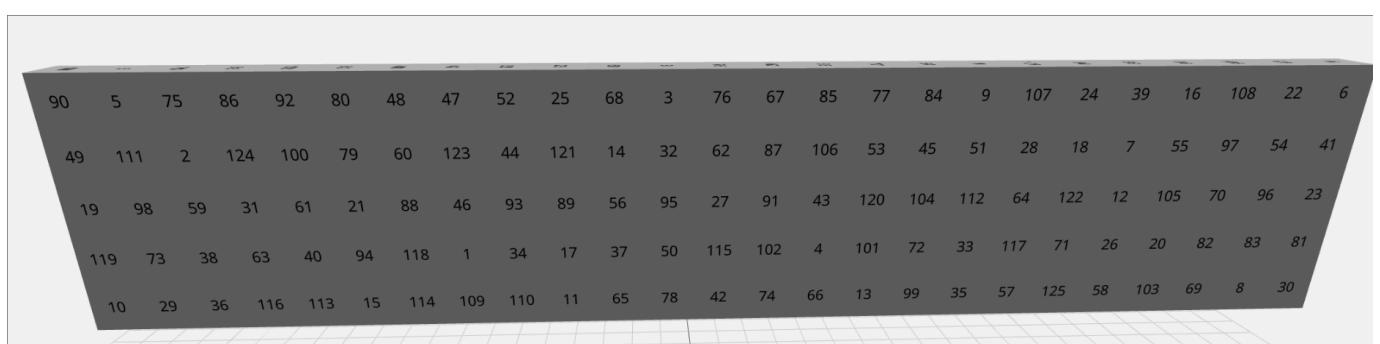
- Plot Objective Function



<b>Nilai <i>objective function</i> terakhir</b>	-671
<b>Durasi pencarian</b>	698 ms
<b>Banyak iterasi</b>	134 iterasi

### c. Eksperimen 3

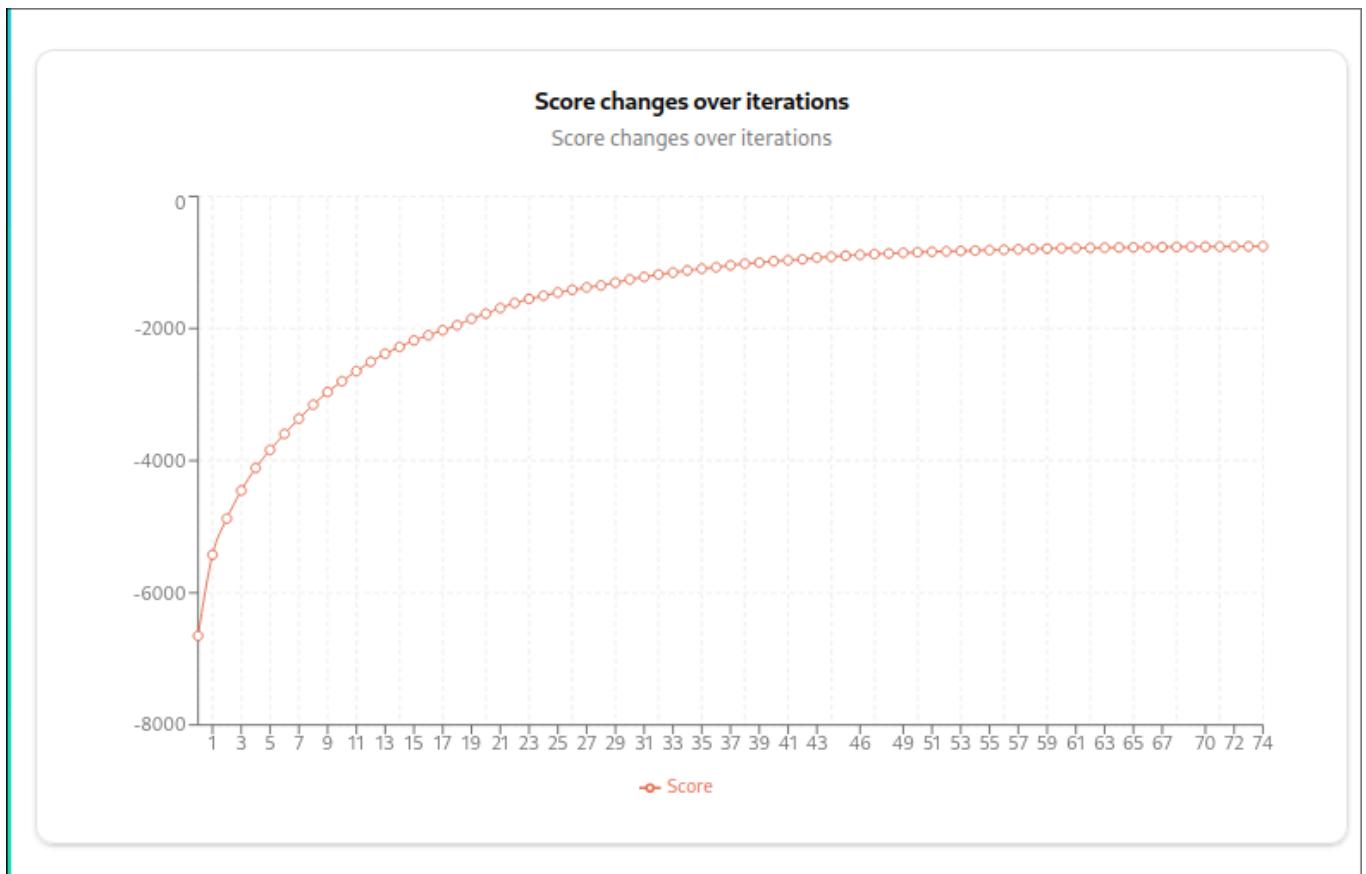
- State Awal



- State Akhir

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

- Plot Objective Function



<b>Nilai objective function terakhir</b>	-756
<b>Durasi pencarian</b>	391 ms
<b>Banyak iterasi</b>	74 iterasi

## 2. Hill-Climbing Sideways Move

a. Eksperimen 1

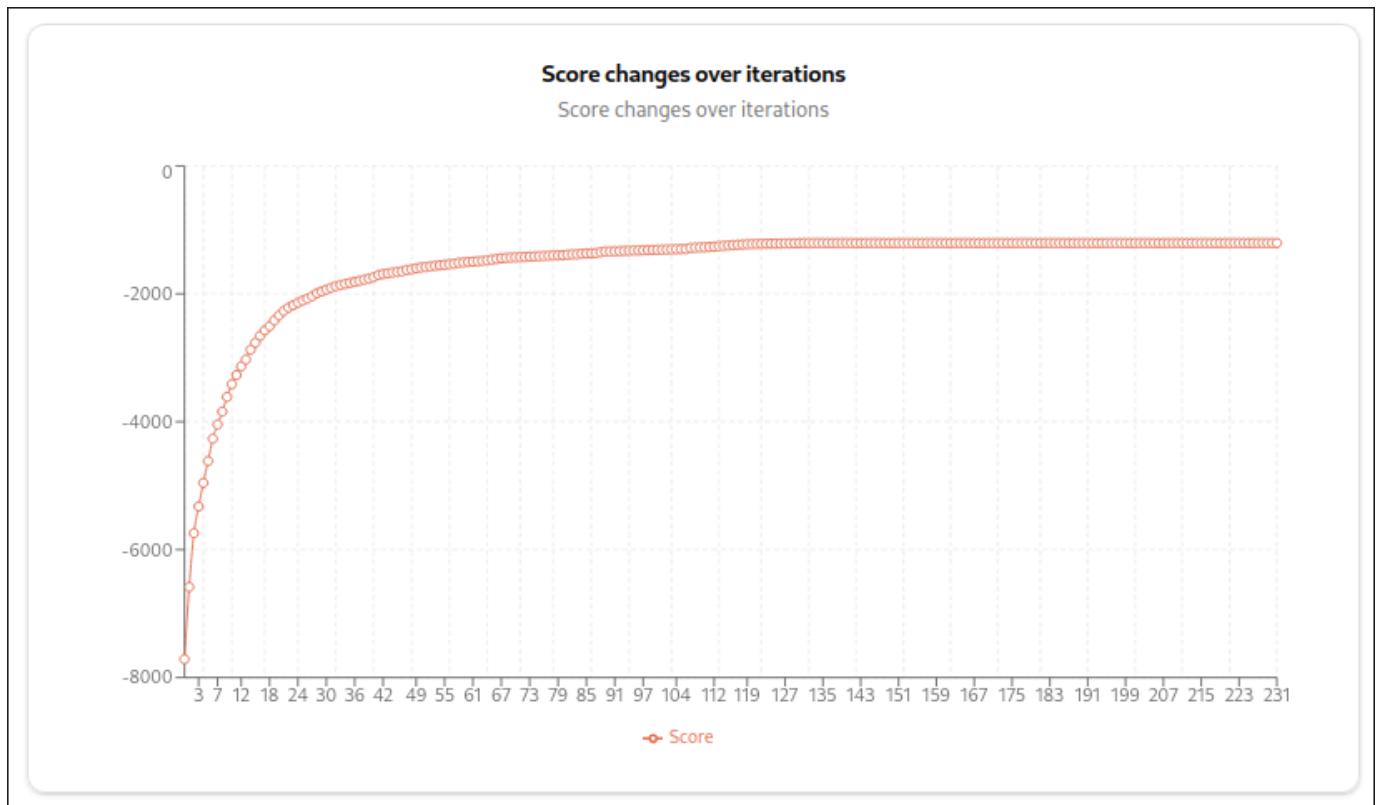
- State Awal

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

- State Akhir

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

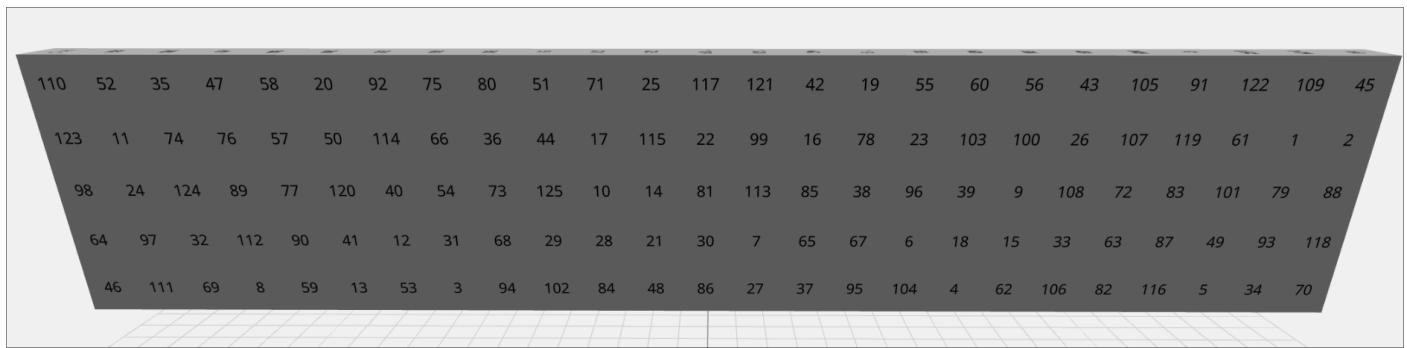
- Plot Objective Function



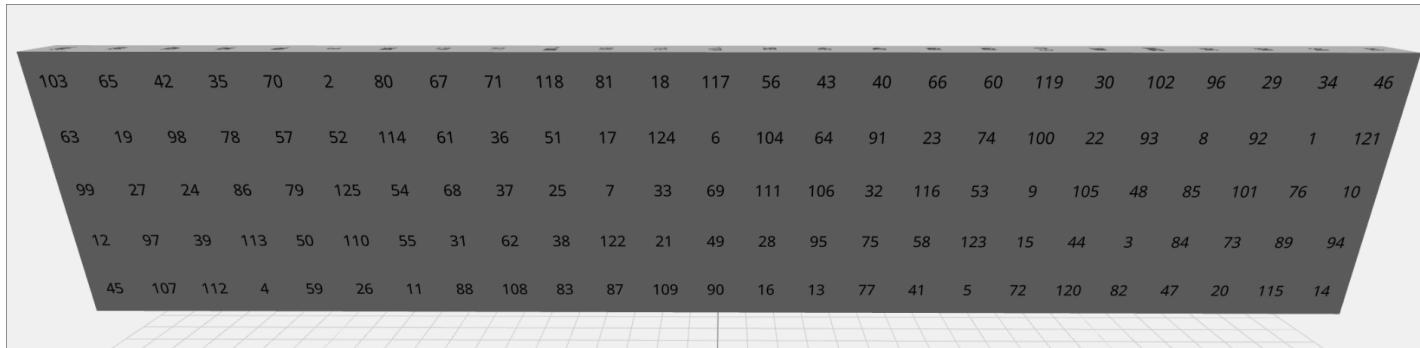
<b>Nilai <i>objective function</i> terakhir</b>	-1205
<b>Durasi pencarian</b>	1187 ms
<b>Banyak iterasi</b>	231
<b>Sideways move maksimum</b>	100

b. Eksperimen 2

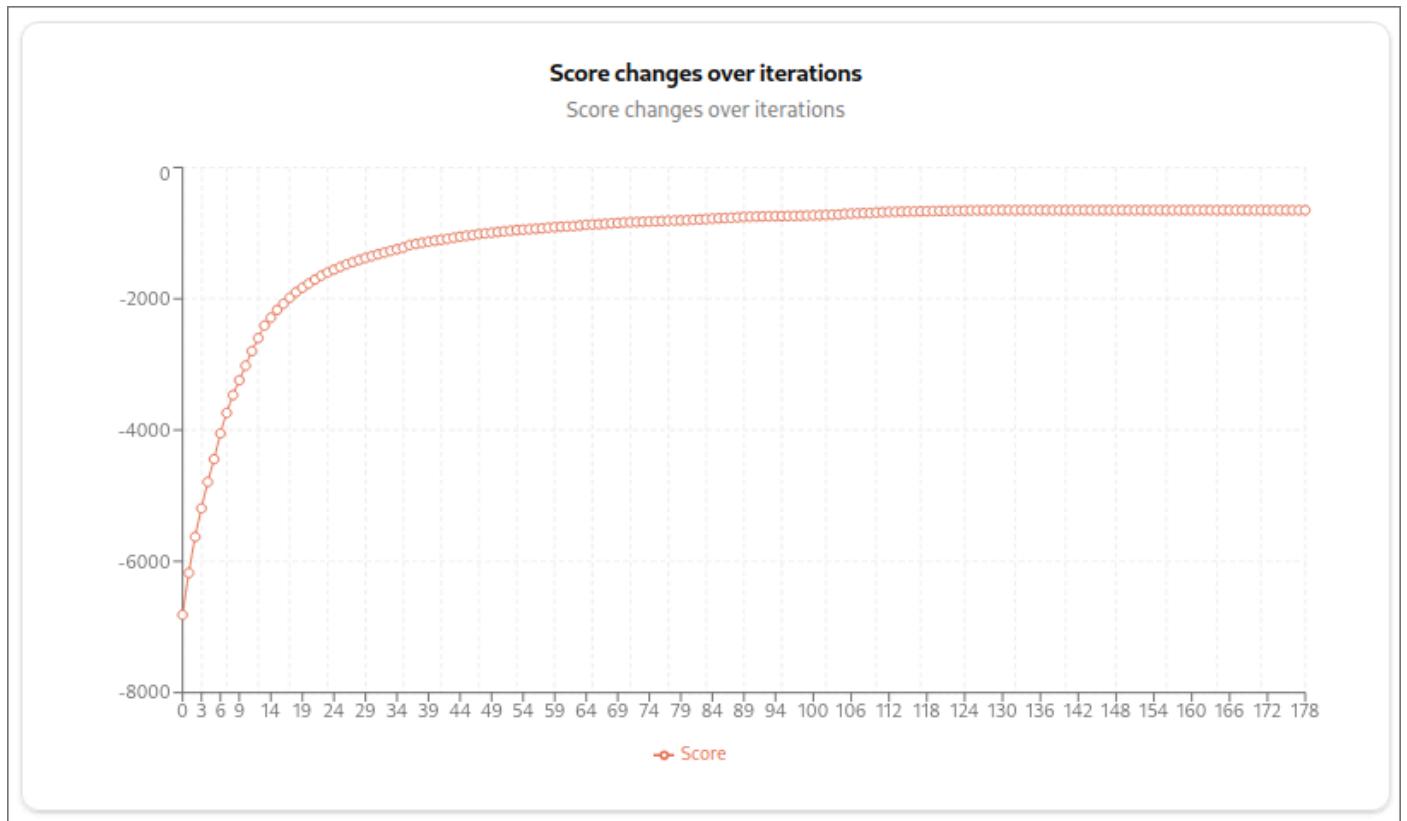
- State Awal



- State Akhir



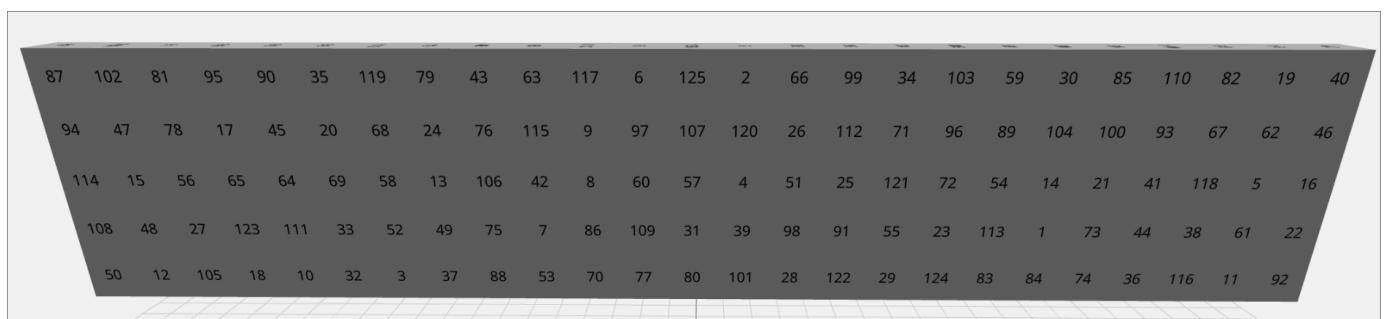
- Plot *Objective Function*



<b>Nilai objective function terakhir</b>	-650
<b>Durasi pencarian</b>	920 ms
<b>Banyak iterasi</b>	178
<b>Sideways move maksimum</b>	50

### c. Eksperimen 3

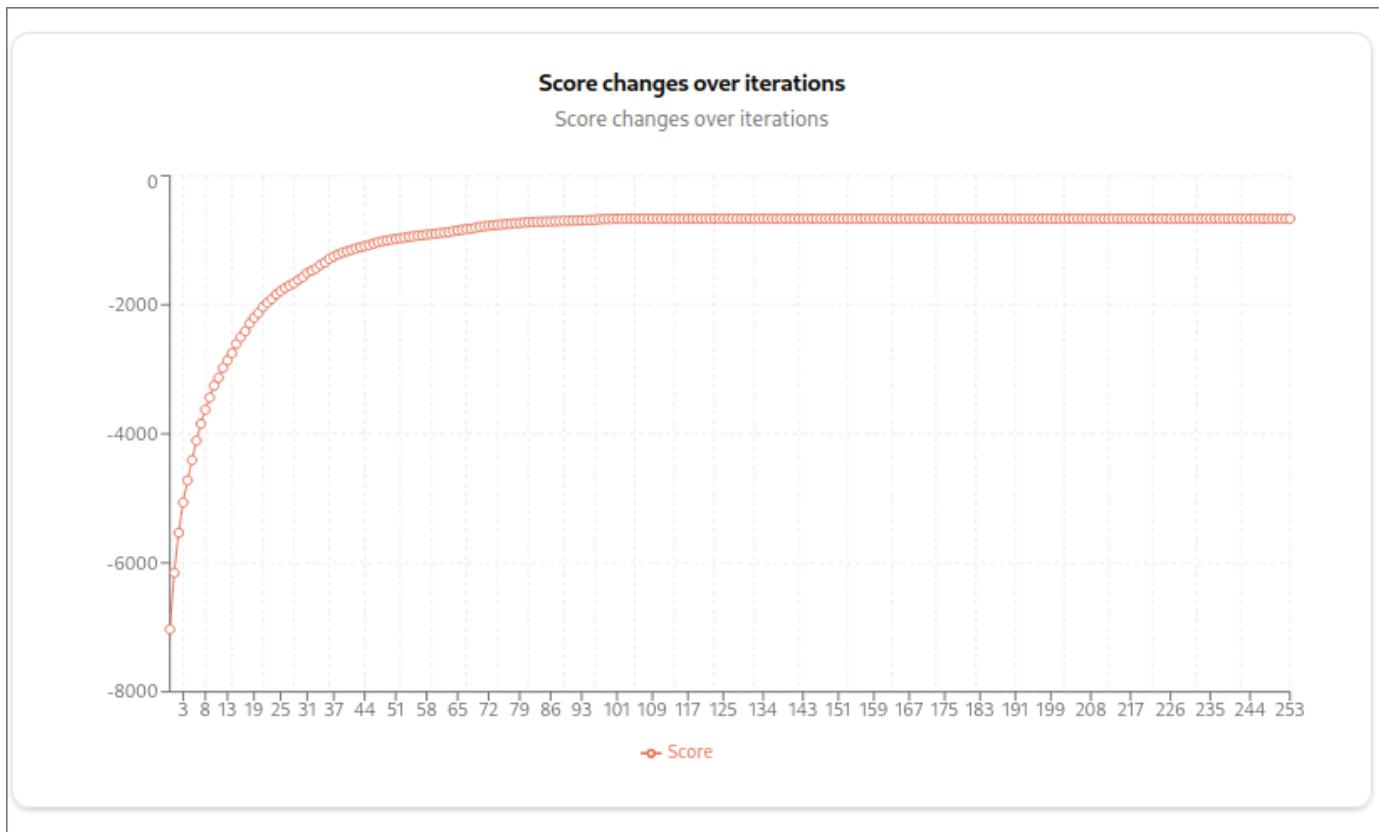
- State Awal



- State Akhir

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

- Plot Objective Function



<b>Nilai objective function terakhir</b>	-655
<b>Durasi pencarian</b>	1313 ms
<b>Banyak iterasi</b>	253 iterasi
<b>Sideways move maksimum</b>	150

### 3. Hill-Climbing Stochastic

a. Eksperimen 1 (max iterasi = 50)

- State Awal

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

Initial/Final State

Initial State

Final State

Controls

Separate X

Separate Y

Separate Z

Flatten X

Flatten Y

Flatten Z

- State Akhir

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

Initial/Final State

Initial State

Final State

Controls

Separate X

Separate Y

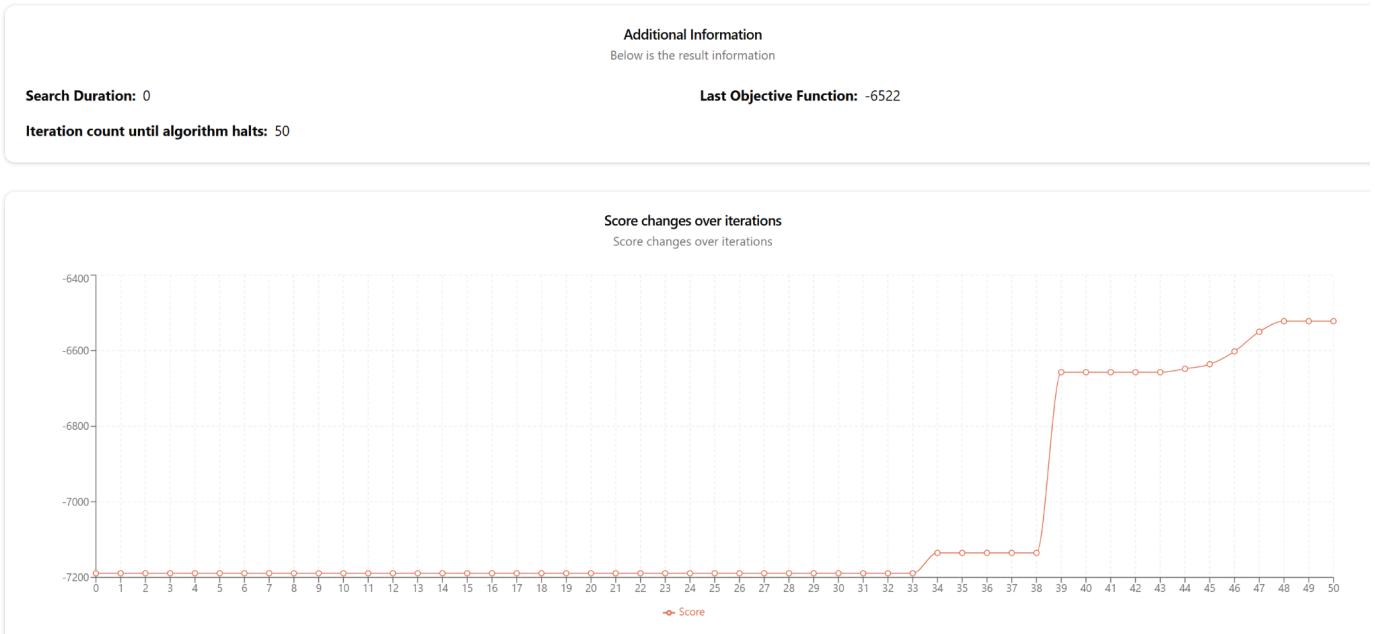
Separate Z

Flatten X

Flatten Y

Flatten Z

- Plot Objective Function



<b>Nilai objective function terakhir</b>	-6522
<b>Durasi pencarian</b>	0 ms
<b>Banyak iterasi</b>	50

### b. Eksperimen 2 (max iterasi = 500)

- State Awal

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

**Initial/Final State**

Initial State

Final State

**Controls**

Separate X
Separate Y
Separate Z

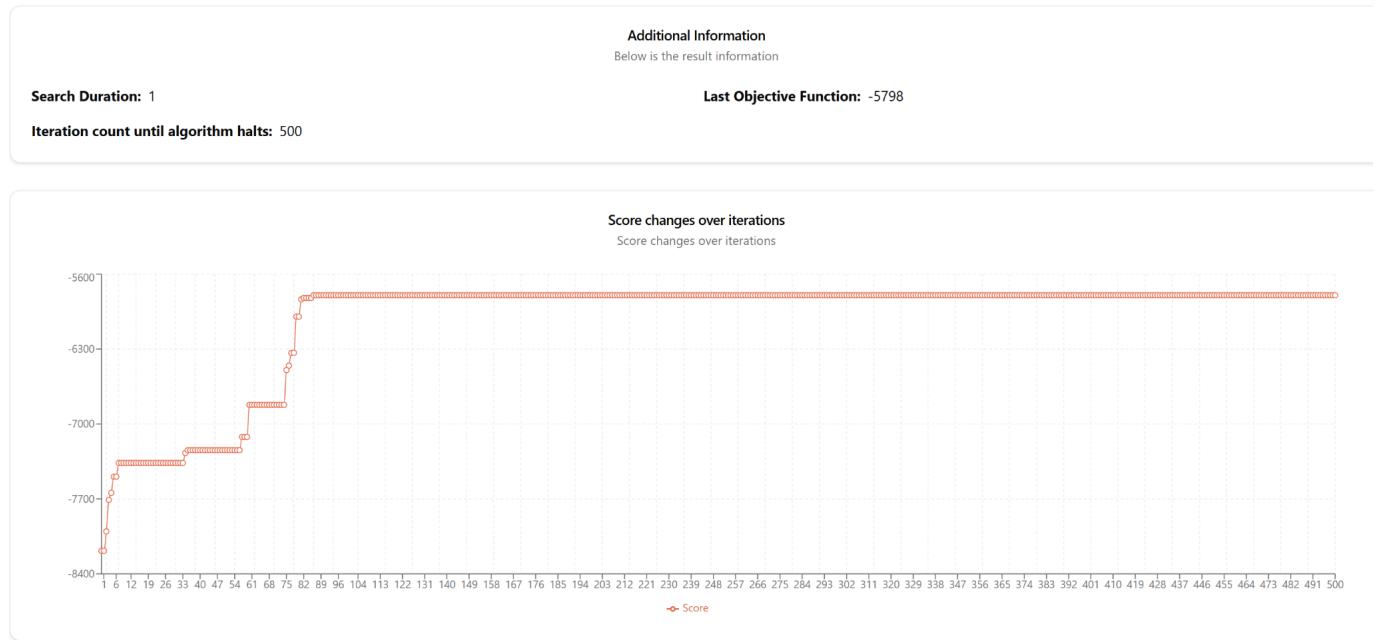
Flatten X
Flatten Y
Flatten Z

- State Akhir

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



### - Plot Objective Function



<b>Nilai objective function terakhir</b>	-5798
<b>Durasi pencarian</b>	1 ms
<b>Banyak iterasi</b>	500

### c. Eksperimen 3 (banyak iterasi = 5000)

#### - State Awal

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

### Initial/Final State

Initial State

Final State

### Controls

Separate X

Separate Y

Separate Z

Flatten X

Flatten Y

Flatten Z

### - State Akhir

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

### Initial/Final State

Initial State

Final State

### Controls

Separate X

Separate Y

Separate Z

Flatten X

Flatten Y

Flatten Z

### - Plot Objective Function

#### Additional Information

Below is the result information

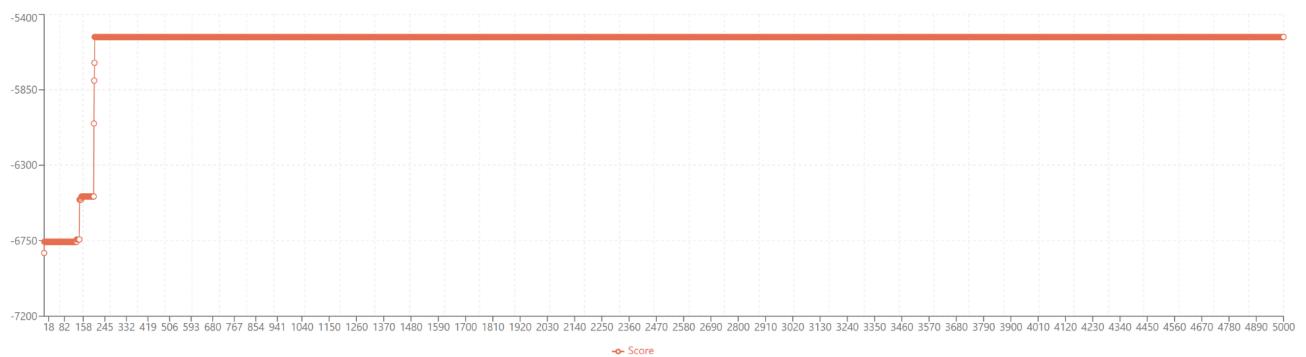
Search Duration: 11

Last Objective Function: -5535

Iteration count until algorithm halts: 5000

#### Score changes over iterations

Score changes over iterations

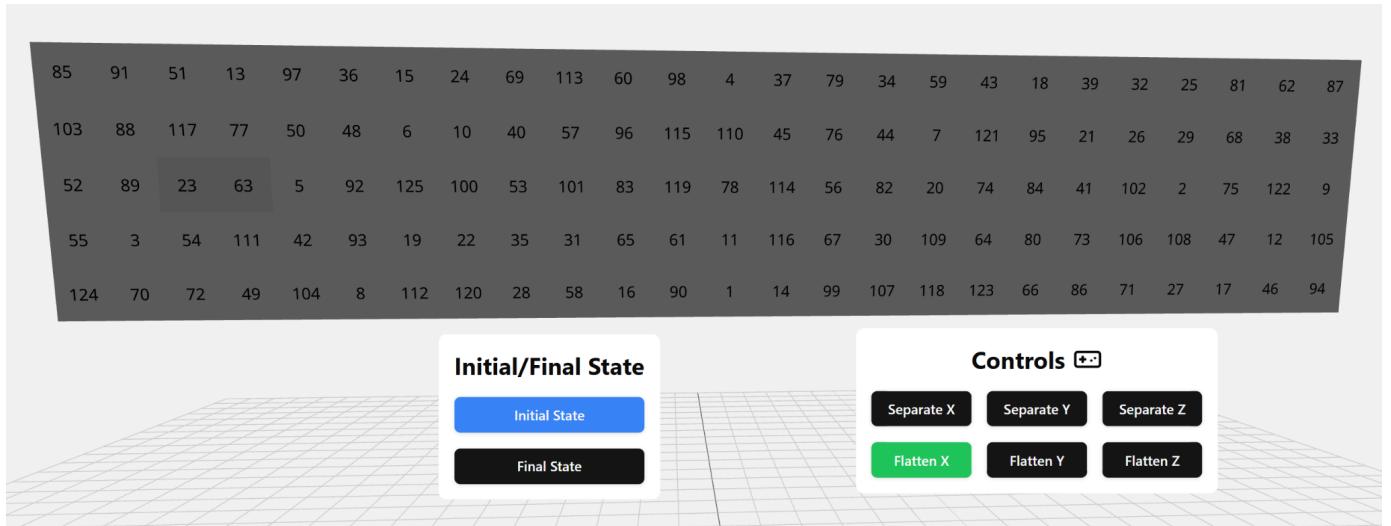


<b>Nilai <i>objective function</i> terakhir</b>	-5535
<b>Durasi pencarian</b>	11 ms
<b>Banyak iterasi</b>	5000

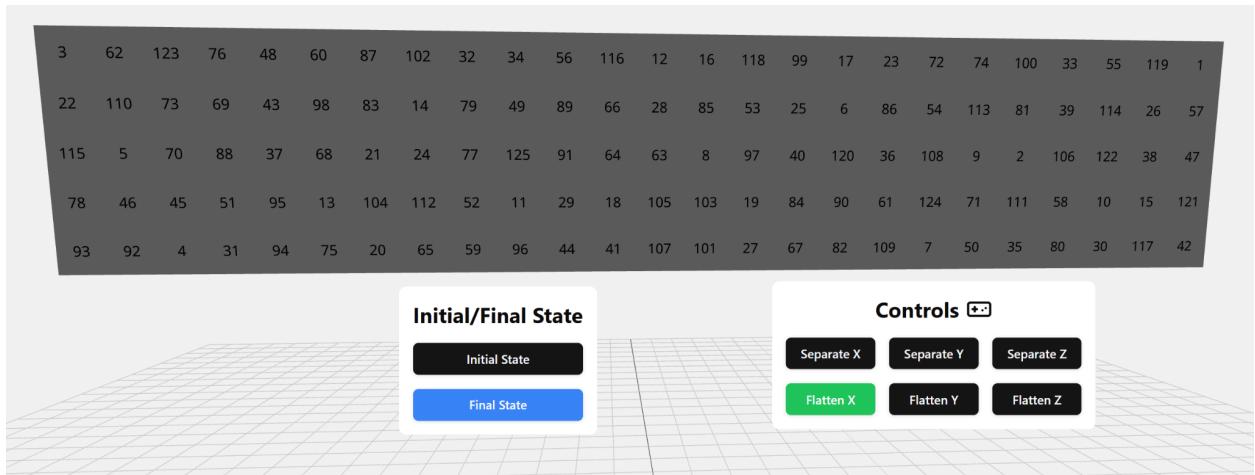
## 4. Random-Restart Hill Climbing

### a. Eksperimen 1 (max restart = 3)

- State Awal

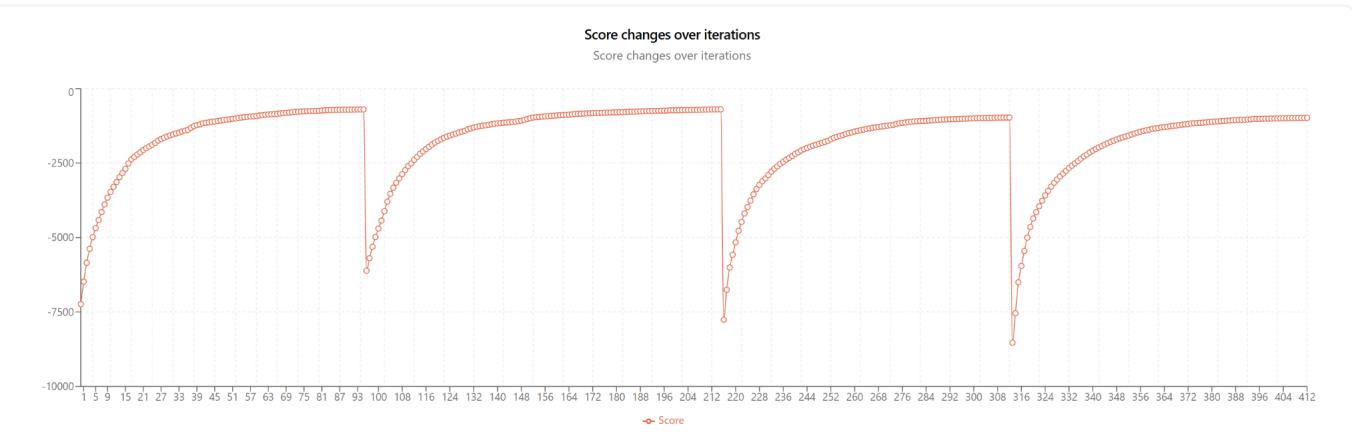


- State Akhir



- Plot Objective Function

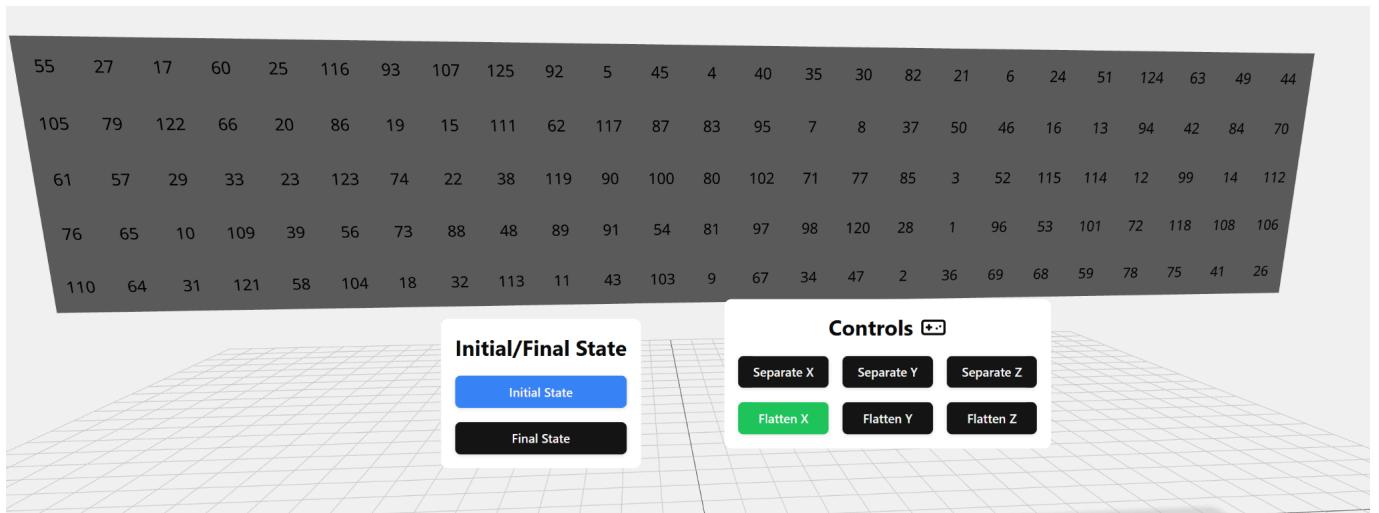
Additional Information	
Below is the result information	
<b>Banyak iterasi restart-0:</b> 96	<b>Banyak iterasi restart-1:</b> 120
<b>Banyak iterasi restart-2:</b> 97	<b>Banyak iterasi restart-3:</b> 100
<b>Banyak restart:</b> 3	<b>Banyak maximal restart:</b> 3
<b>Search Duration:</b> 3624	<b>Last Objective Function:</b> -981



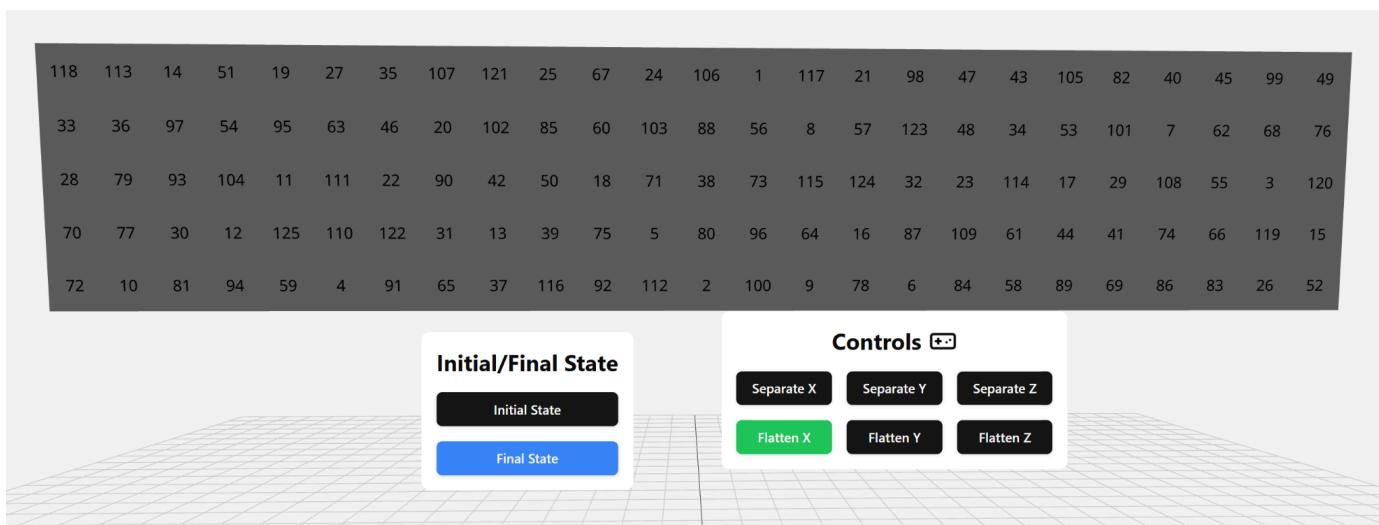
<b>Nilai <i>objective function</i> terakhir</b>	-981
<b>Durasi pencarian</b>	3624 ms
<b>Banyak restart</b>	3
<b>Banyak iterasi per <i>restart</i></b>	0 : 96 1 : 120 2 : 97 3: 100

b. Eksperimen 2 (max restart = 5)

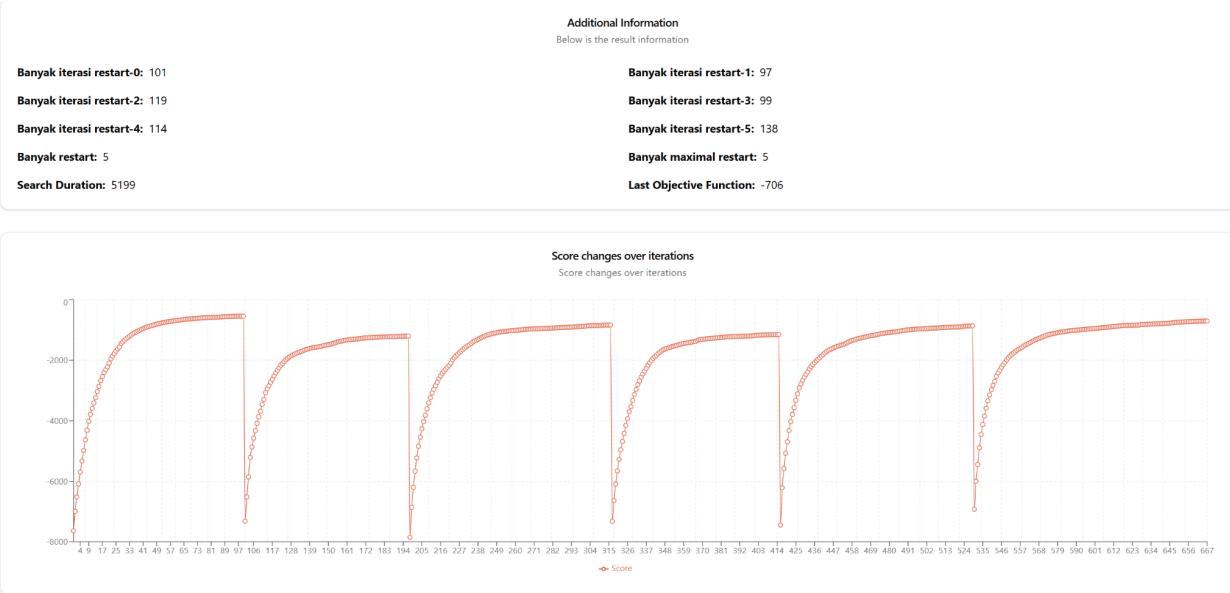
- State Awal



- State Akhir



- Plot Objective Function



<b>Nilai <i>objective function</i> terakhir</b>	-706
<b>Durasi pencarian</b>	5199 ms
<b>Banyak restart</b>	5
<b>Banyak iterasi per restart</b>	0: 101 1 : 97 2 : 119 3 : 99 4 : 114 5: 138

c. Eksperimen 3 (max restart = 6)

- State Awal

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

#### Initial/Final State

Initial State

Final State

#### Controls

Separate X Separate Y Separate Z

Flatten X Flatten Y Flatten Z

### - State Akhir

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

#### Initial/Final State

Initial State

Final State

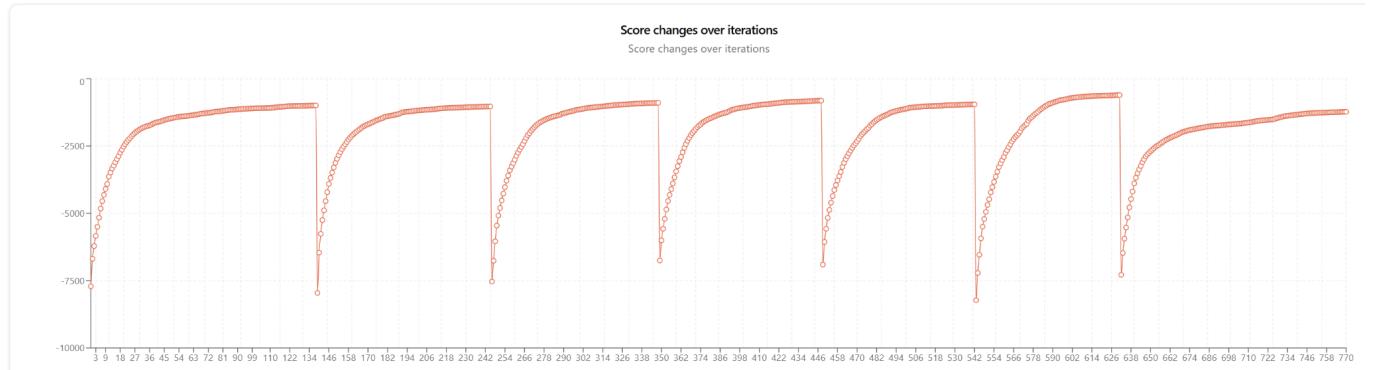
#### Controls

Separate X Separate Y Separate Z

Flatten X Flatten Y Flatten Z

### - Plot Objective Function

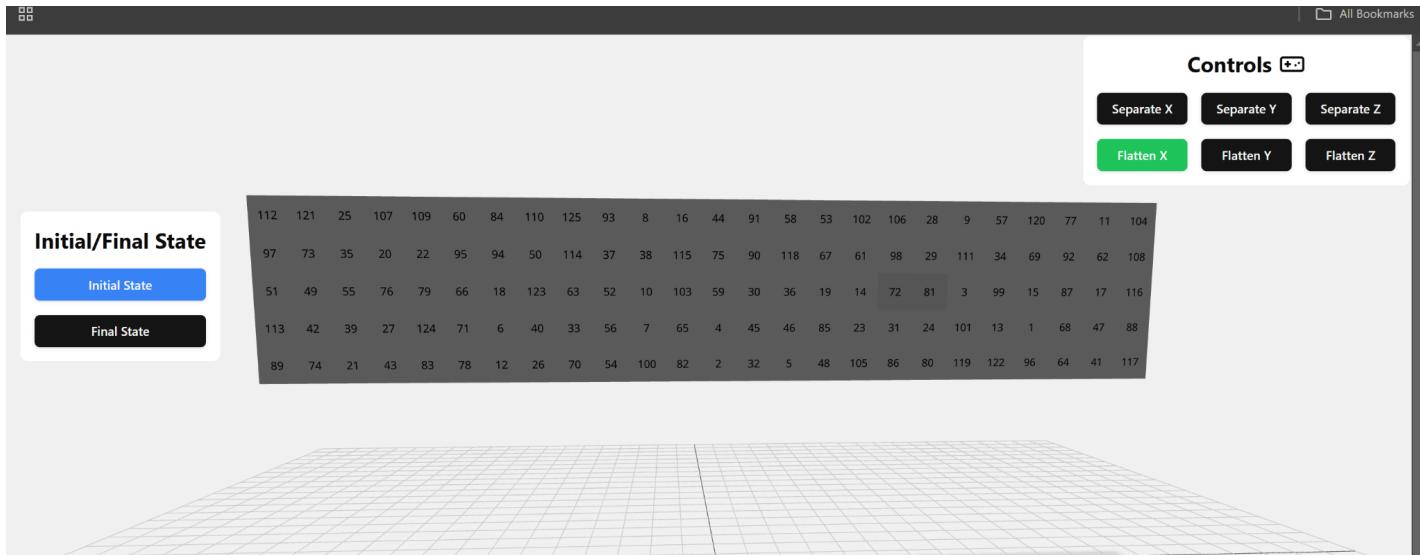
Additional Information	
Below is the result information	
Banyak iterasi restart-0:	139
Banyak iterasi restart-2:	103
Banyak iterasi restart-4:	94
Banyak iterasi restart-6:	139
Banyak maximal restart:	6
Last Objective Function:	-1227
Banyak iterasi restart-1:	107
Banyak iterasi restart-3:	100
Banyak iterasi restart-5:	89
Banyak restart:	6
Search Duration:	6373



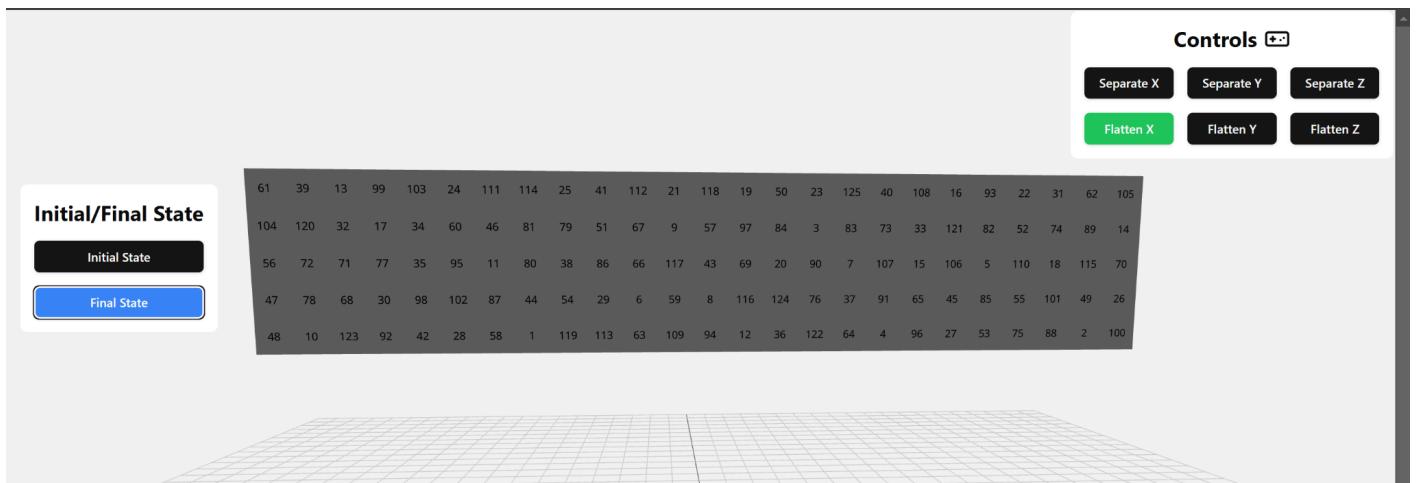
<b>Nilai objective function terakhir</b>	-1227
<b>Durasi pencarian</b>	6373 ms
<b>Banyak restart</b>	6
<b>Banyak iterasi per restart</b>	0:139 1:107 2: 103 3: 100 4: 94 5: 89 6: 139

## 5. Simulated Annealing

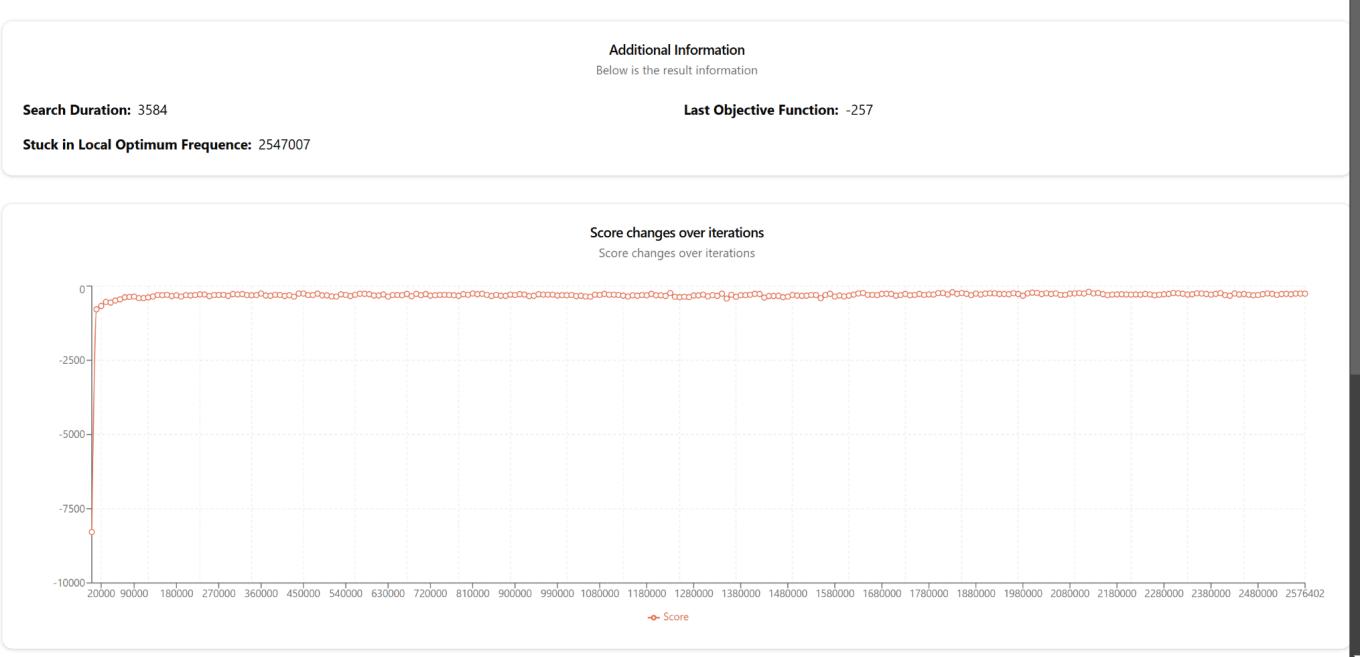
- a. Eksperimen 1
  - State Awal



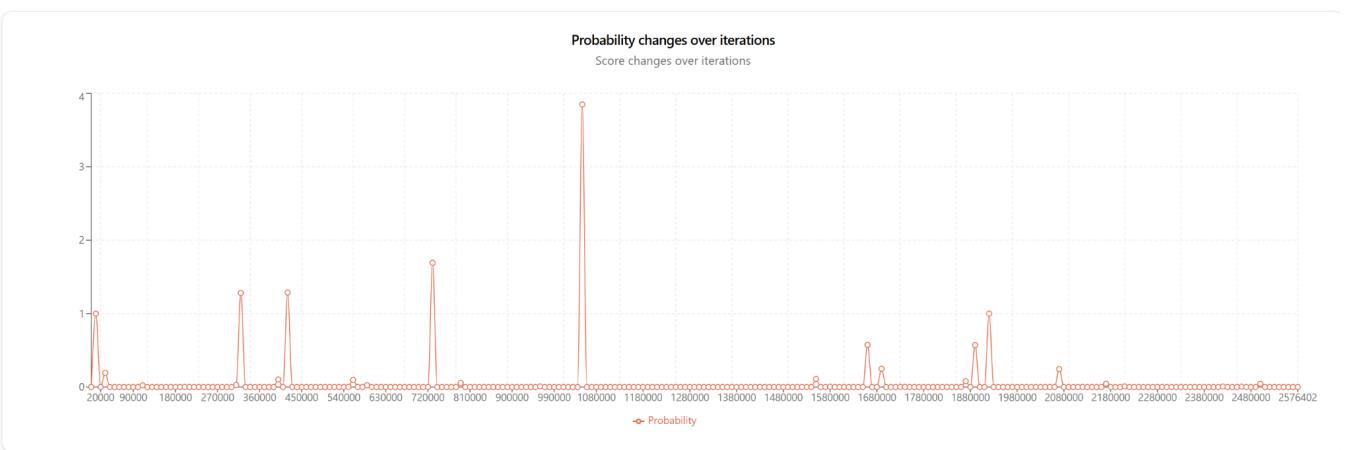
- State Akhir



- Plot Objective Function + informasi stuck



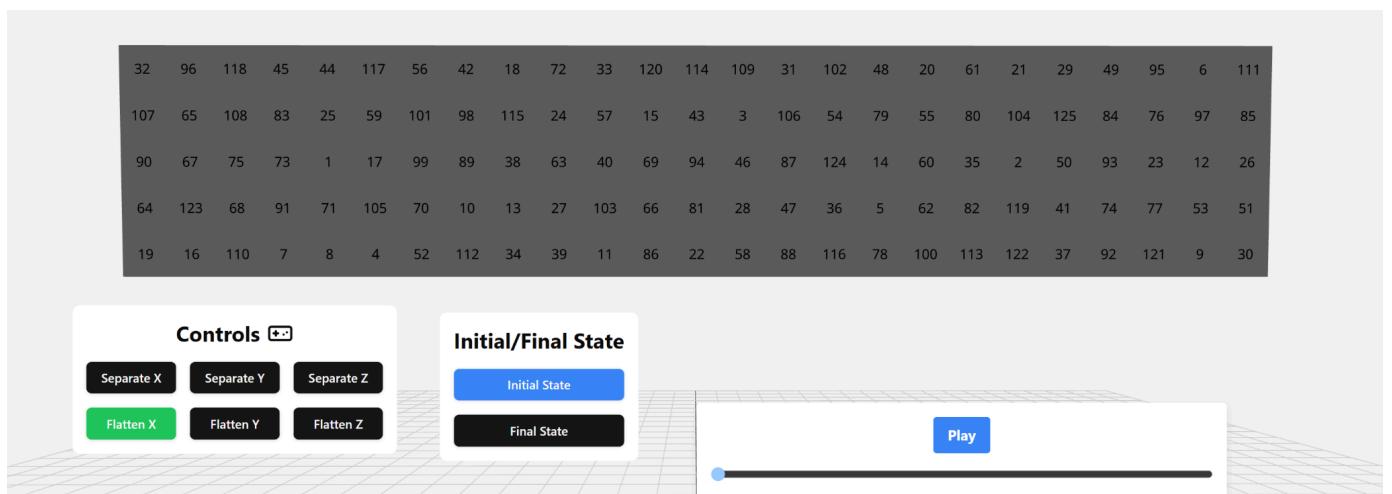
-  $\frac{\Delta E}{T}$



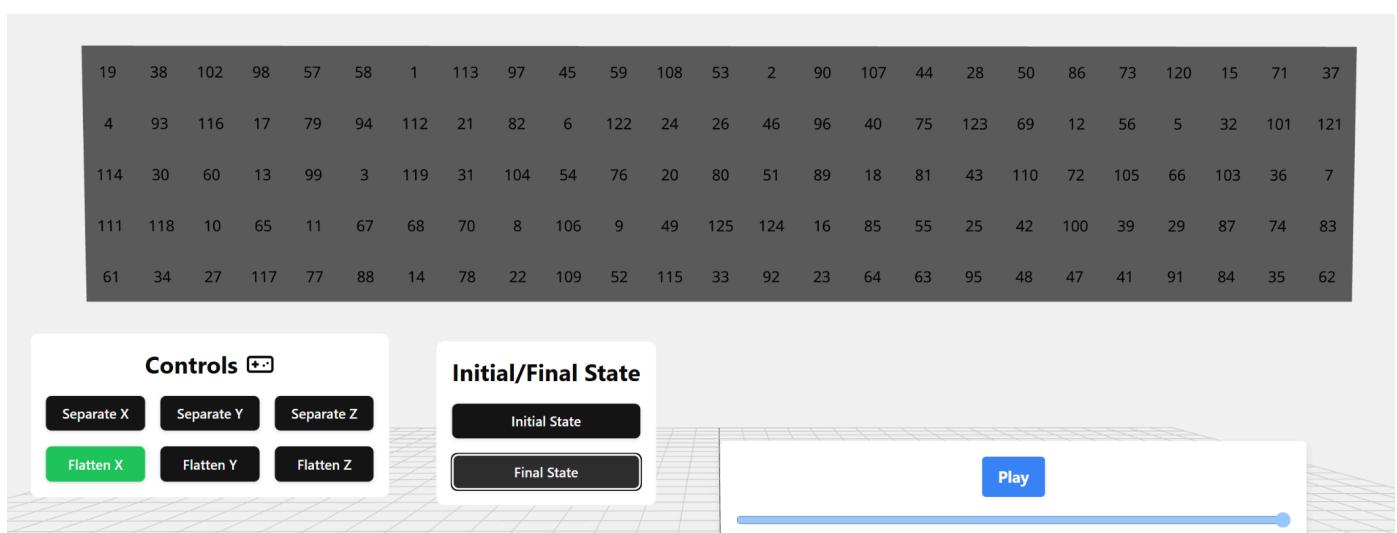
<b>Nilai objective function terakhir</b>	-257
<b>Durasi pencarian</b>	3584 ms
<b>Frekuensi stuck</b>	2547007

## b. Eksperimen 2

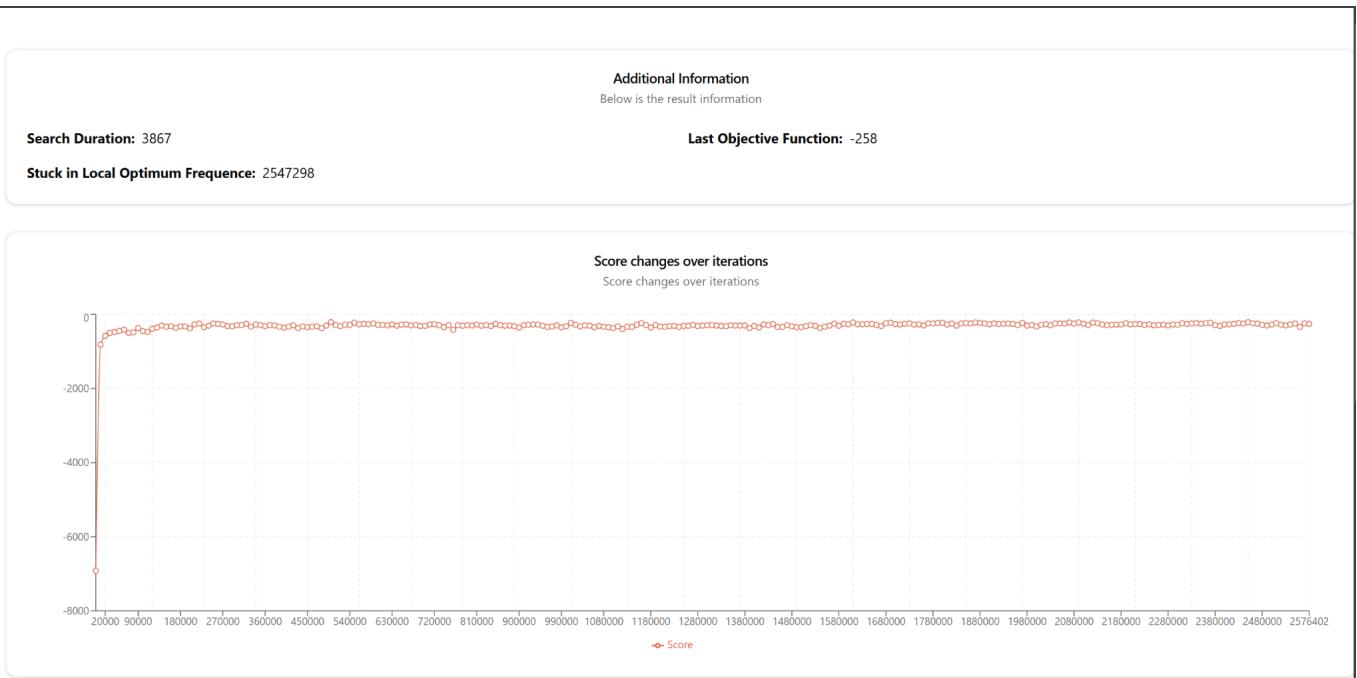
- State Awal



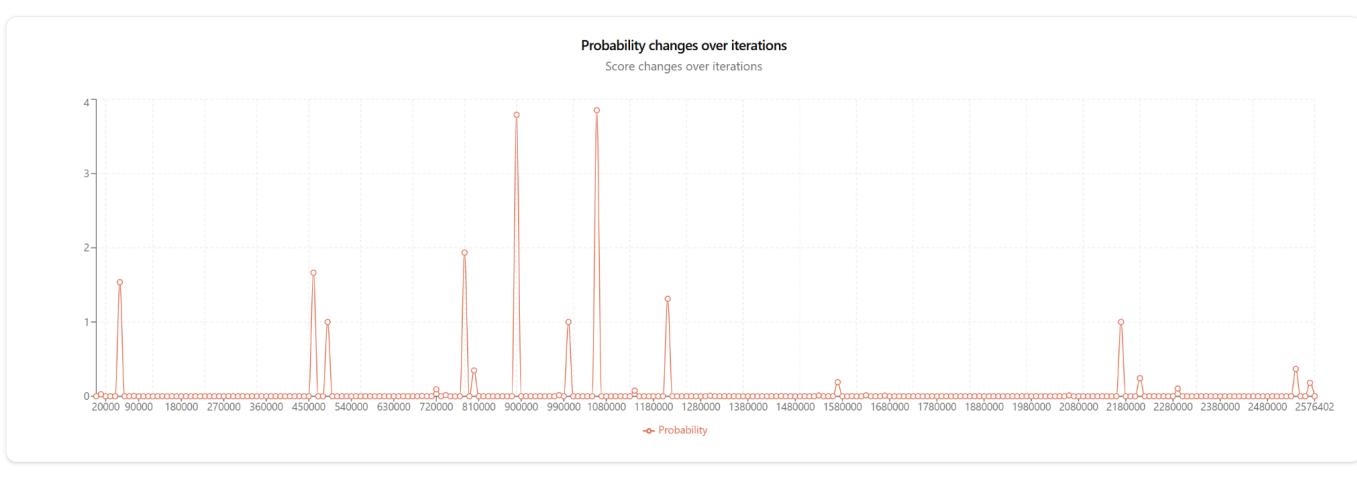
- State Akhir



- Plot Objective Function + informasi stuck



- Plot  $e^{\frac{\Delta E}{T}}$



<b>Nilai objective function terakhir</b>	-258
<b>Durasi pencarian</b>	3867 ms
<b>Frekuensi stuck</b>	2547298

c. Eksperimen 3  
- State Awal

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

**Initial/Final State**

Initial State

Final State

**Controls**

Separate X Separate Y Separate Z

Flatten X Flatten Y Flatten Z

Play

- State Akhir

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

**Initial/Final State**

Initial State

Final State

**Controls**

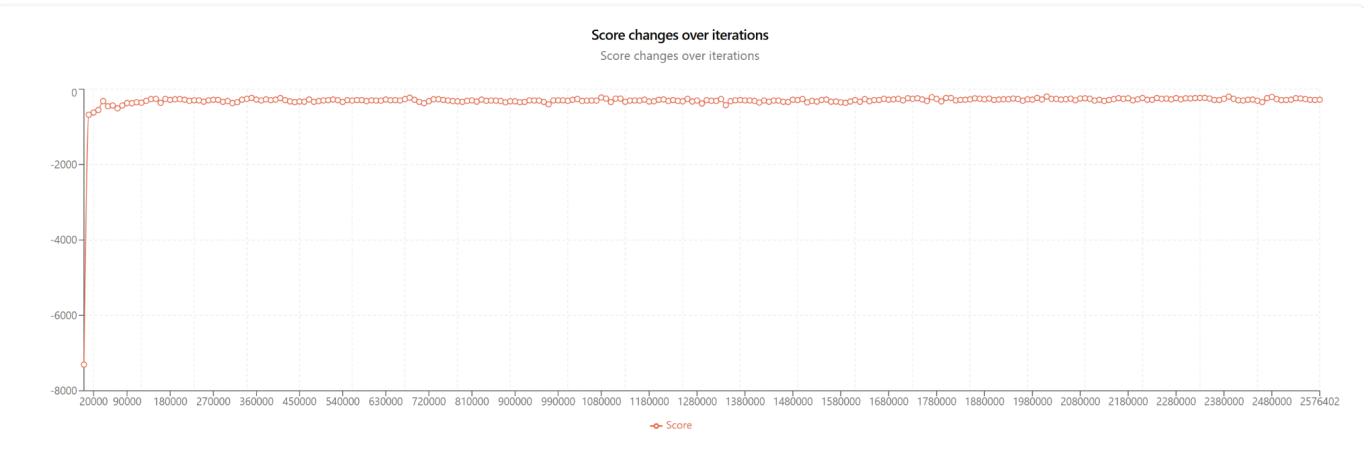
Separate X Separate Y Separate Z

Flatten X Flatten Y Flatten Z

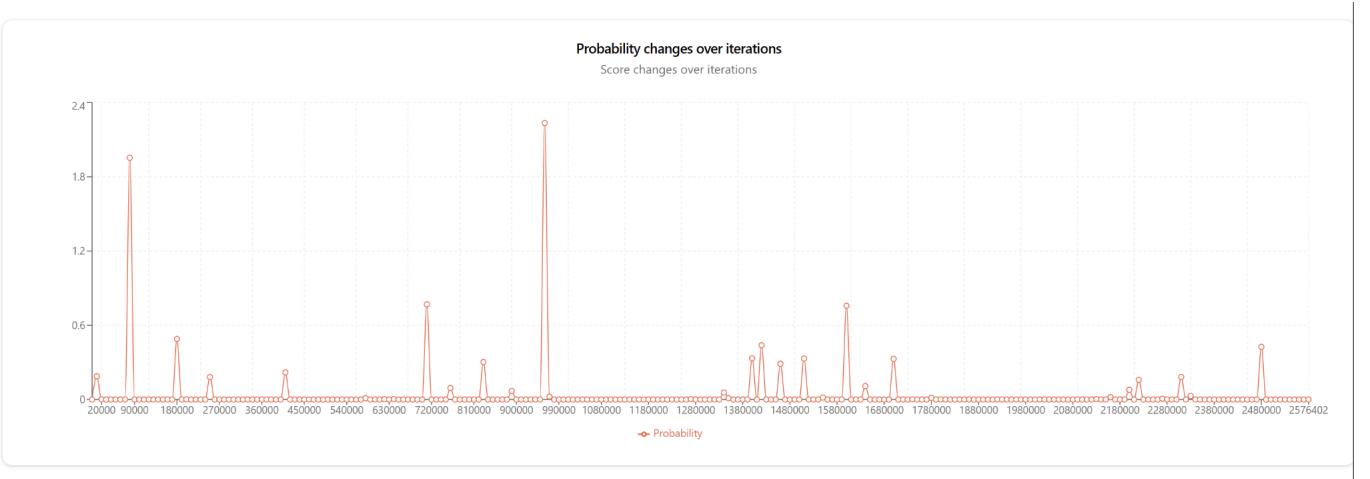
Play

- Plot Objective Function + informasi stuck

Additional Information	
Below is the result information	
<b>Search Duration:</b> 3861	<b>Last Objective Function:</b> -278
<b>Stuck in Local Optimum Frequency:</b> 2547591	



- Plot  $e^{\frac{\Delta E}{T}}$



<b>Nilai objective function terakhir</b>	-278
<b>Durasi pencarian</b>	3861 ms
<b>Frekuensi stuck</b>	2547591

## 6. Genetic Algorithm

a. Jumlah Populasi sebagai kontrol  
 i. **Populasi = 10, Iterasi = 10**

1. Eksperimen 1

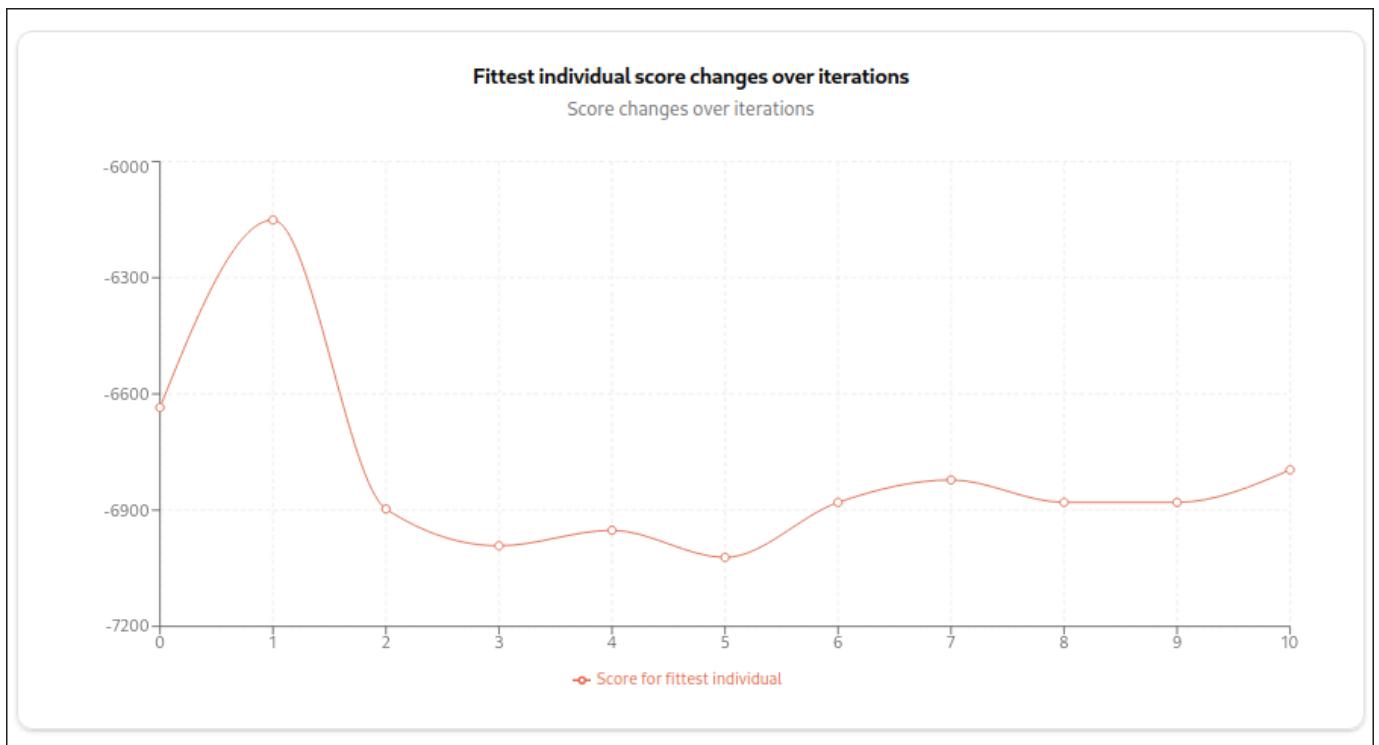
- State awal

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

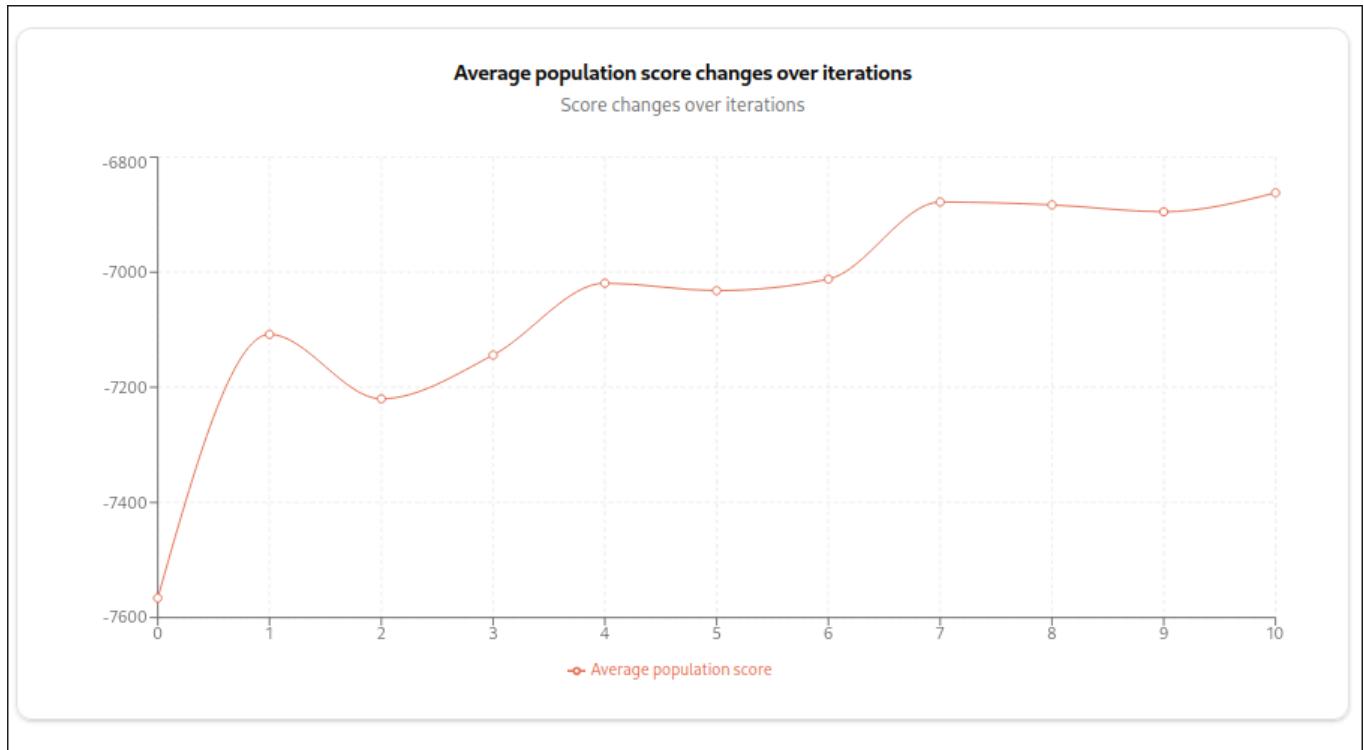
- State akhir

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

- Plot objective function (best score from population)



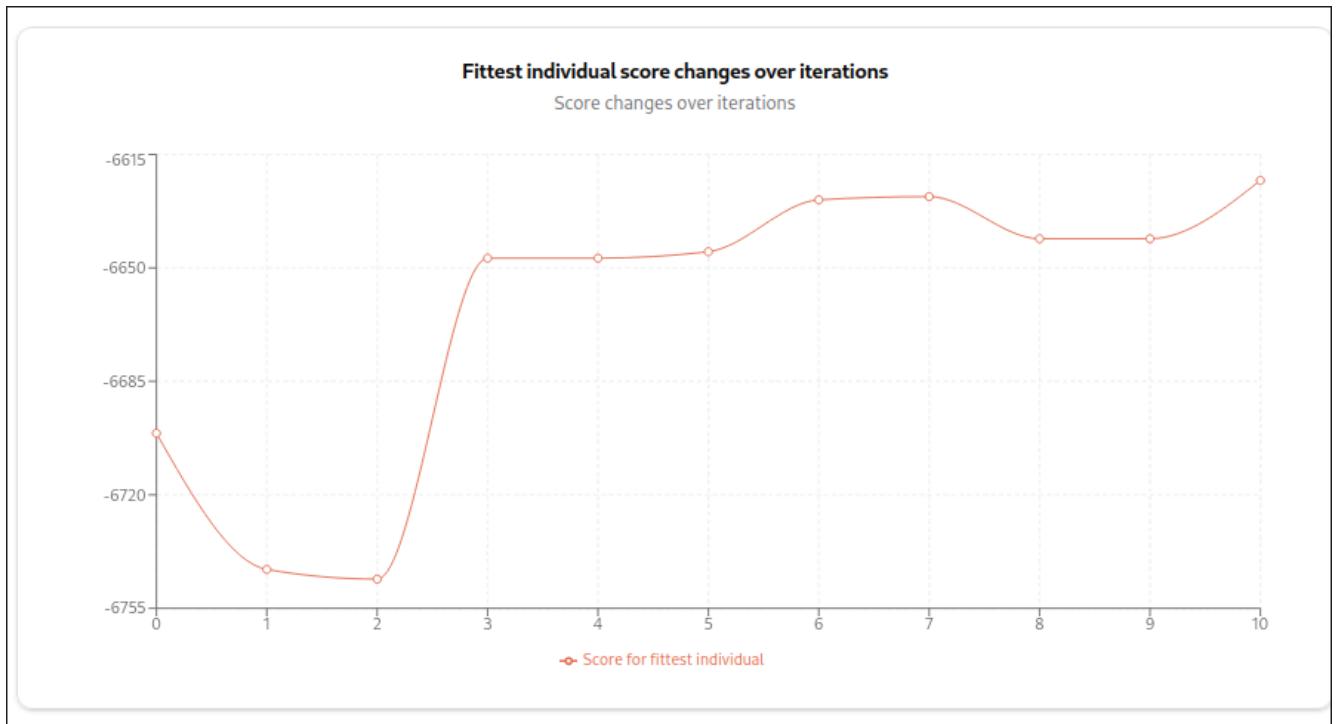
- Plot *objective function* (average score from population)



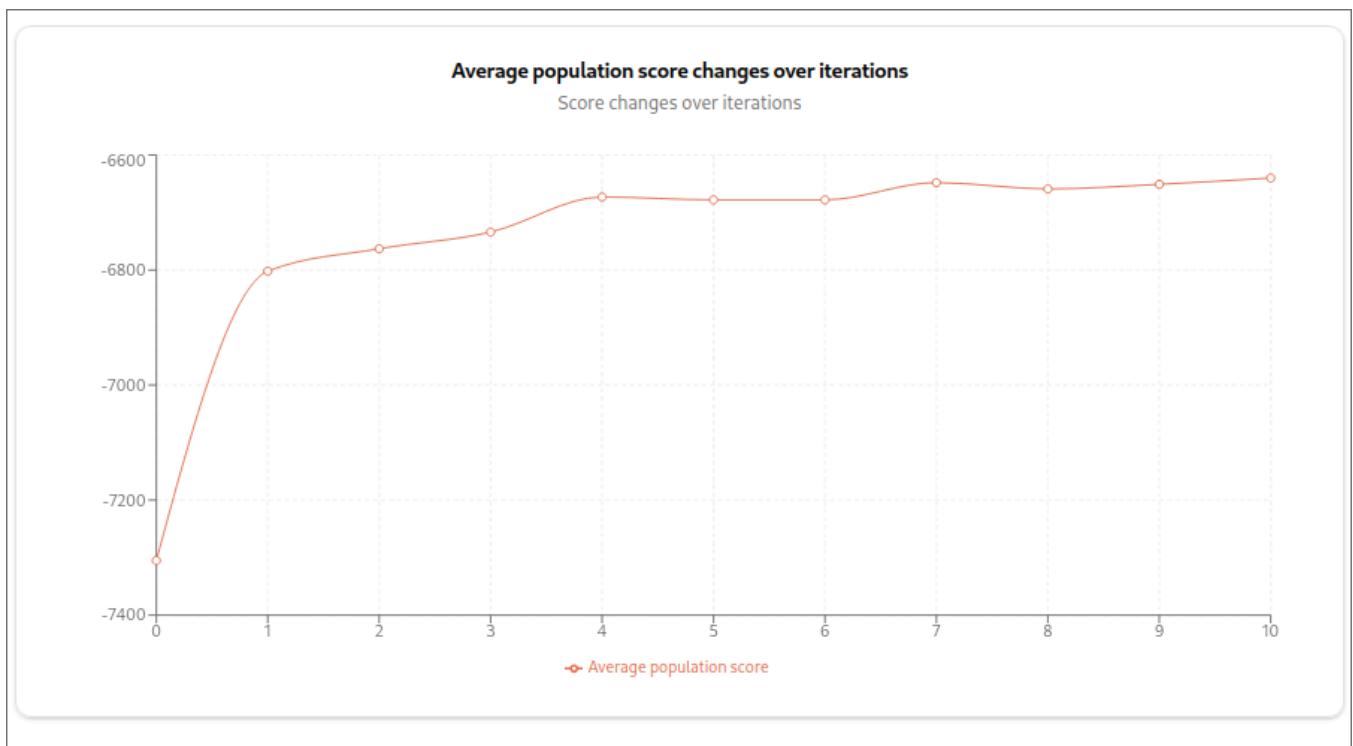
Nilai <i>objective function</i> terakhir	-6796
Durasi pencarian	3 ms
Jumlah populasi	10
Banyak iterasi	10

## 2. Eksperimen 2

- State awal
- State akhir
- Plot *objective function* (best score from population)



- Plot *objective function* (average score from population)

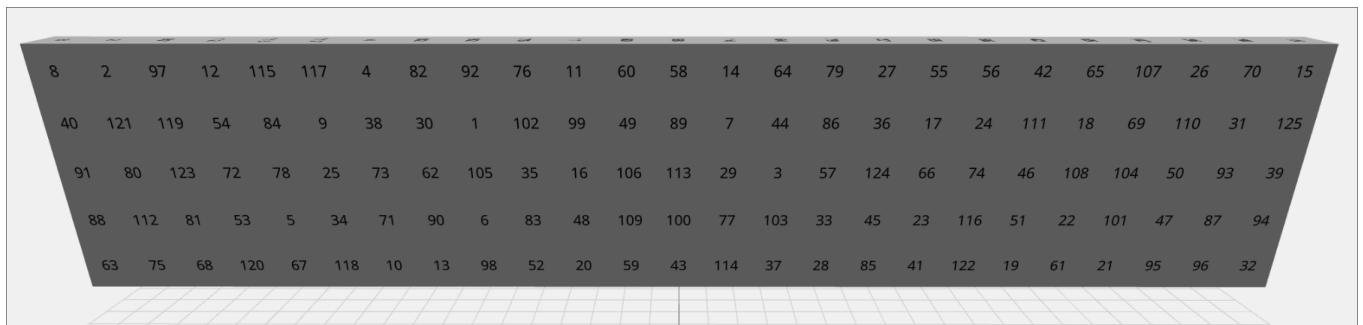


<b>Nilai <i>objective function</i> terakhir</b>	-6623
---	-------

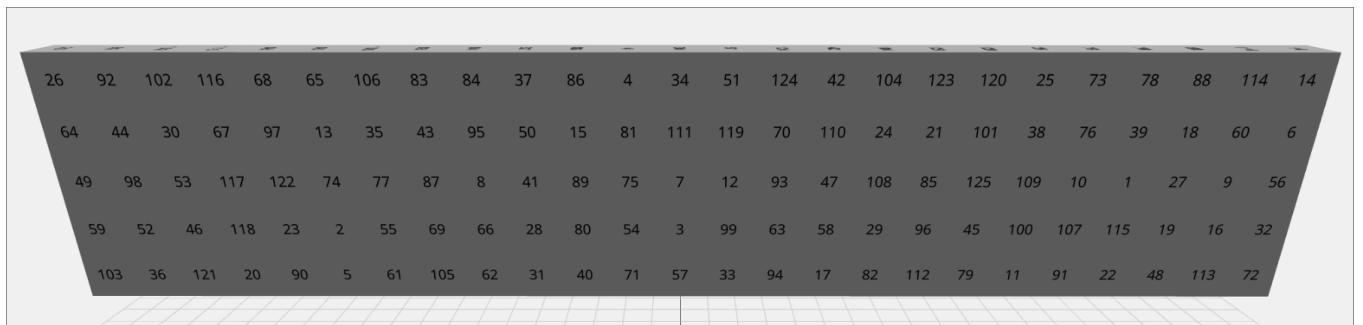
<b>Durasi pencarian</b>	3 ms
<b>Jumlah populasi</b>	10
<b>Banyak iterasi</b>	10

### 3. Eksperimen 3

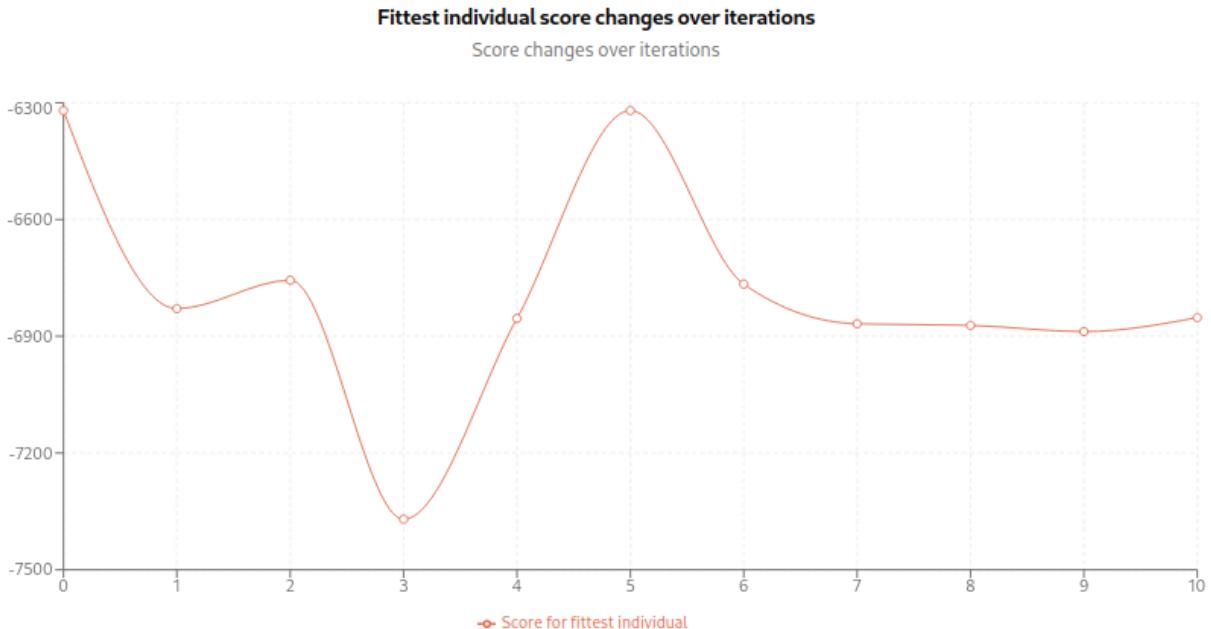
- State awal



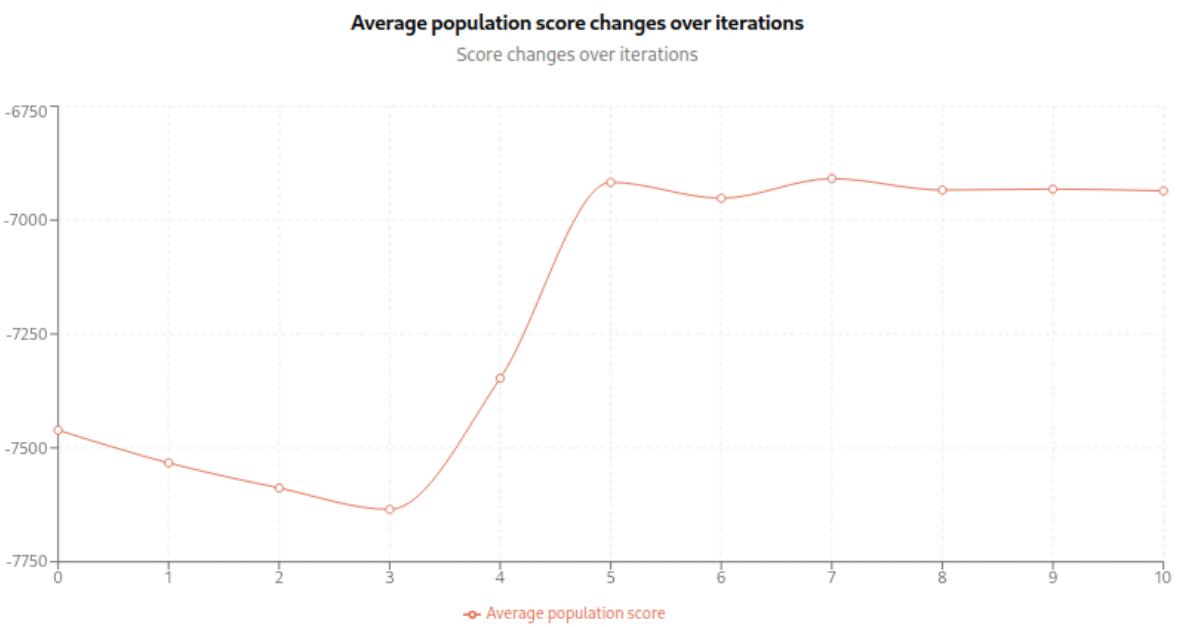
- State akhir



- Plot *objective function* (best score from population)



- Plot *objective function* (average score from population)



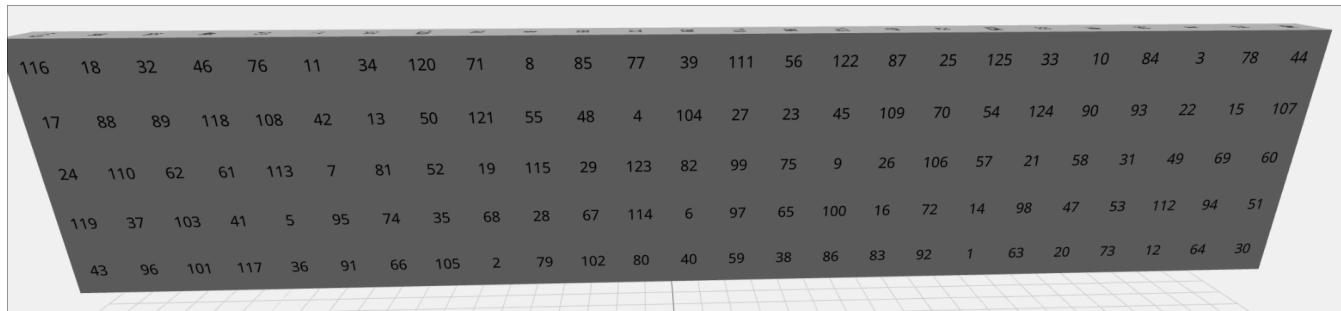
Nilai <b>objective function</b> terakhir	-6852
--	-------

<b>Durasi pencarian</b>	3 ms
<b>Jumlah populasi</b>	10
<b>Banyak iterasi</b>	10

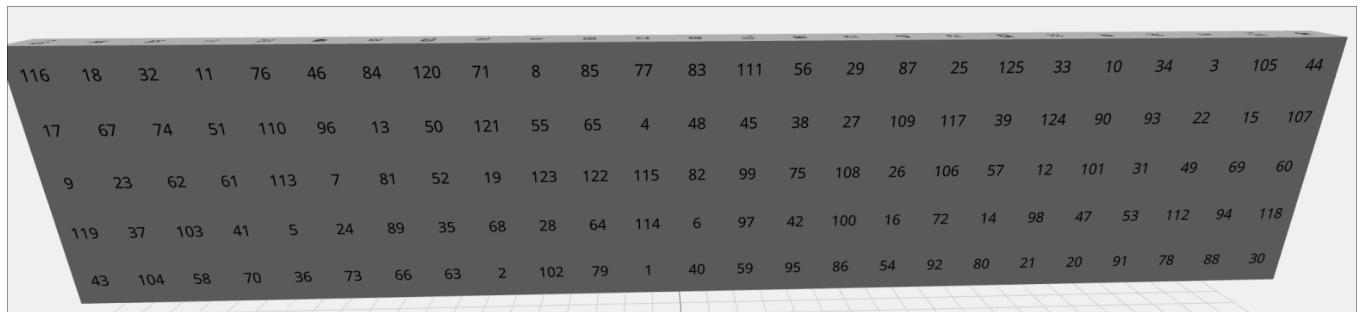
## ii. Populasi = 10, Iterasi = 100

### 1. Eksperimen 1

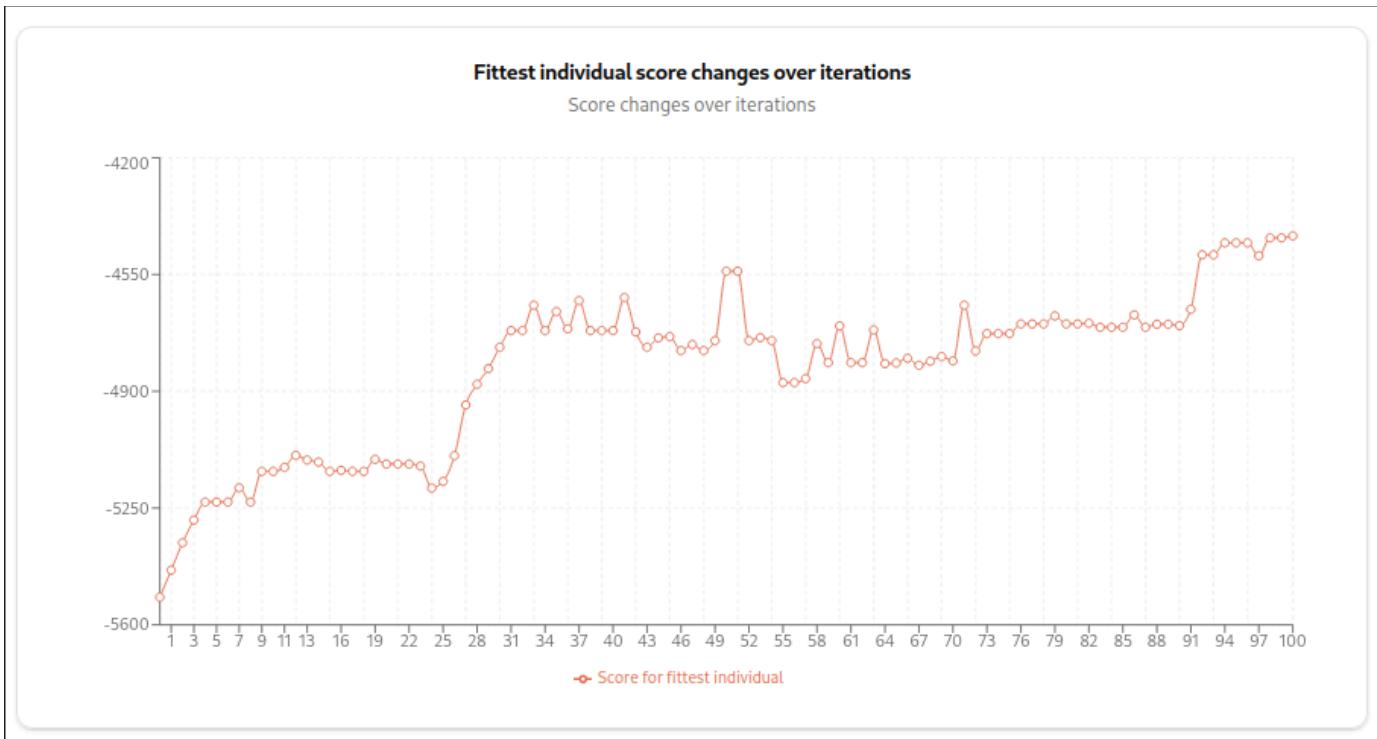
- State awal



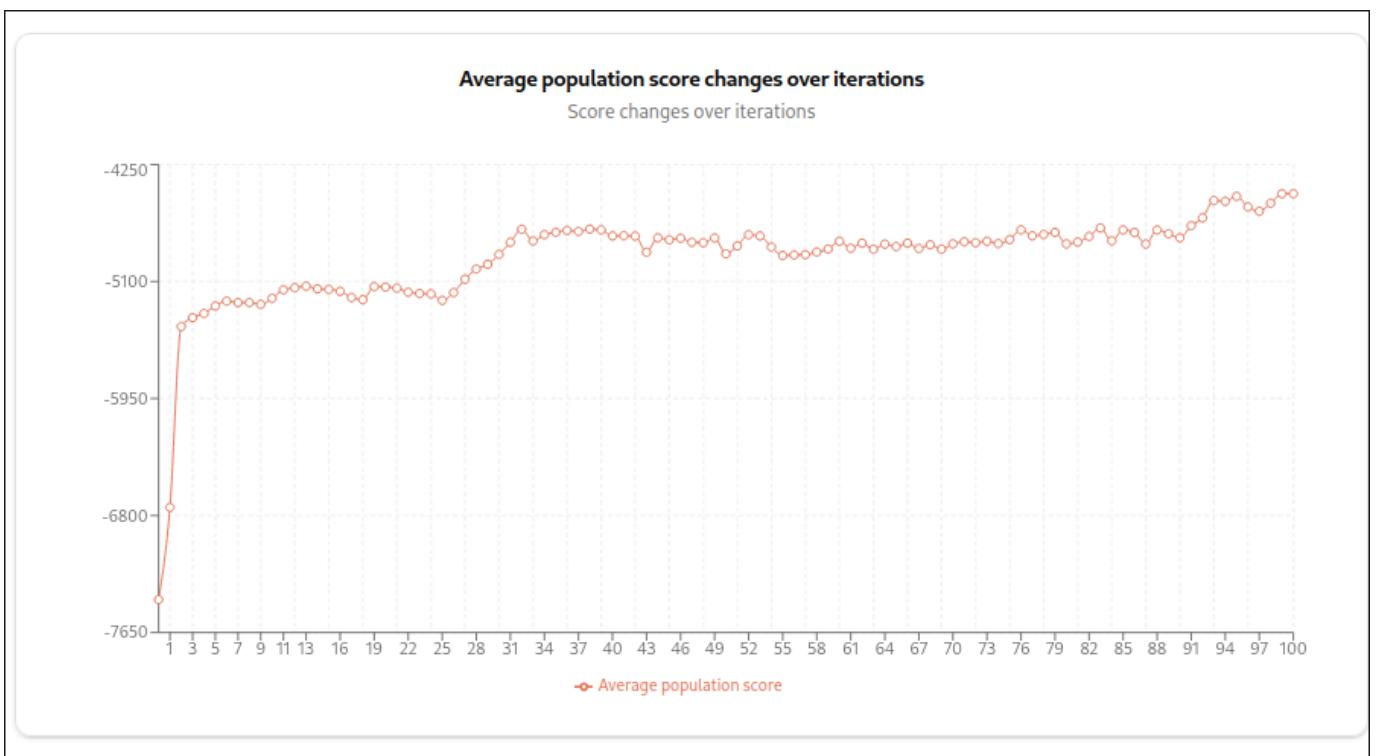
- State akhir



- Plot *objective function* (best score from population)



- Plot *objective function* (average score from population)



Nilai <b>objective function</b> terakhir	-4434
--	-------

<b>Durasi pencarian</b>	39 ms
<b>Jumlah populasi</b>	10
<b>Banyak iterasi</b>	100

## 2. Eksperimen 2

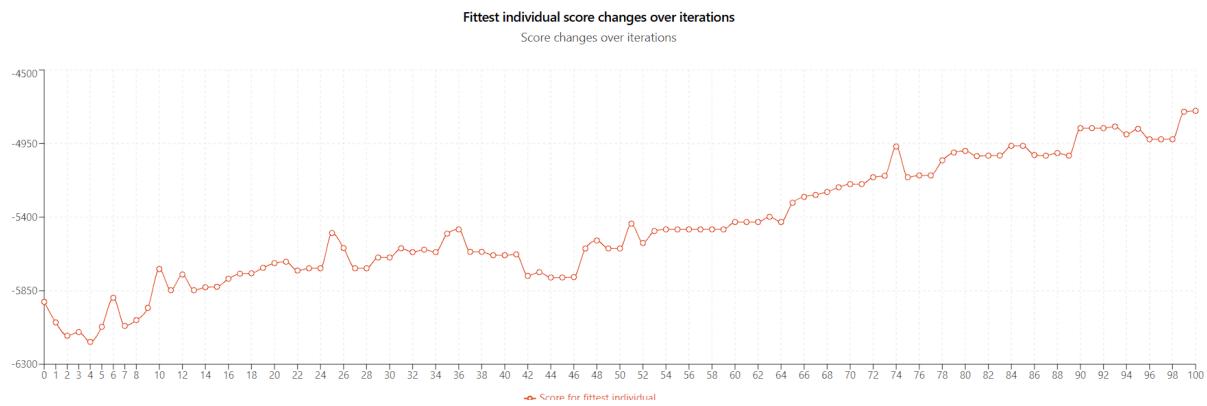
### - State Awal

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

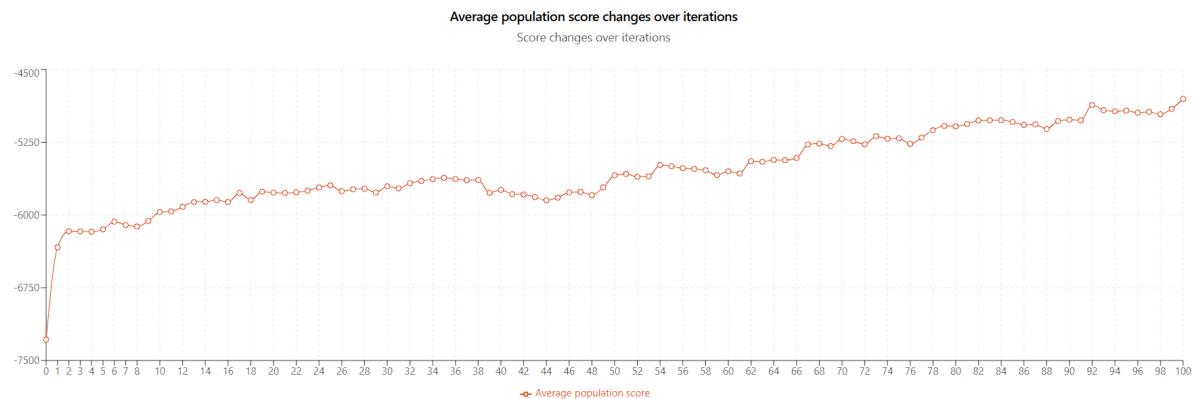
### - State Akhir

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

### - Plot *objective function* (best score from population)



- Plot *objective function*



<b>Nilai objective function terakhir</b>	-4750
<b>Durasi pencarian</b>	51ms
<b>Jumlah populasi</b>	10
<b>Banyak iterasi</b>	100

### 3. Eksperimen 3

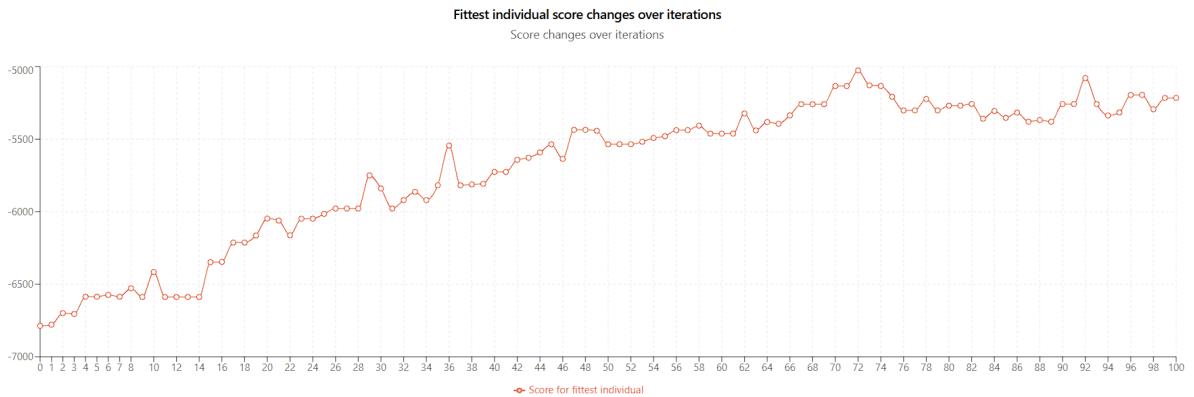
- State awal

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

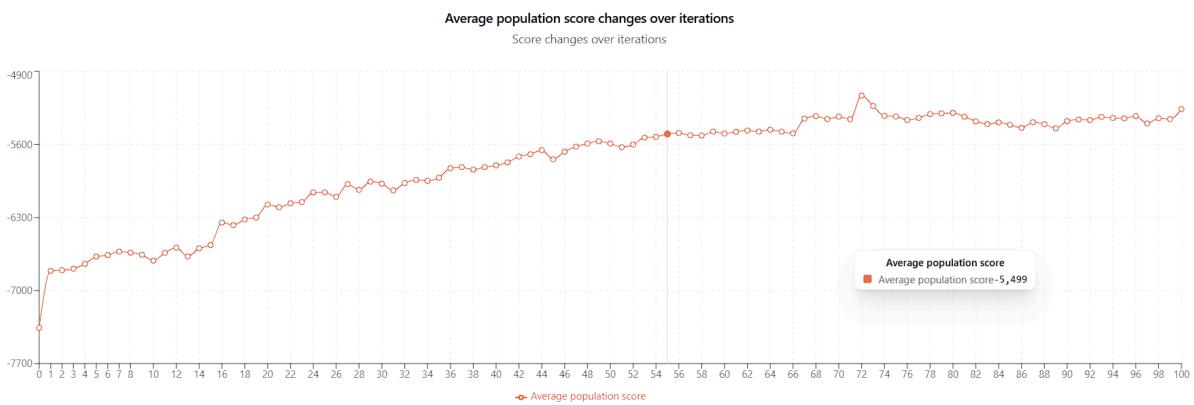
- State akhir

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

- Plot *objective function* (best score from population)



- Plot objective function (average score from population)



<b>Nilai objective function terakhir</b>	-5216
<b>Durasi pencarian</b>	73ms
<b>Jumlah populasi</b>	10
<b>Banyak iterasi</b>	100

### iii. Populasi = 10, Iterasi = 200

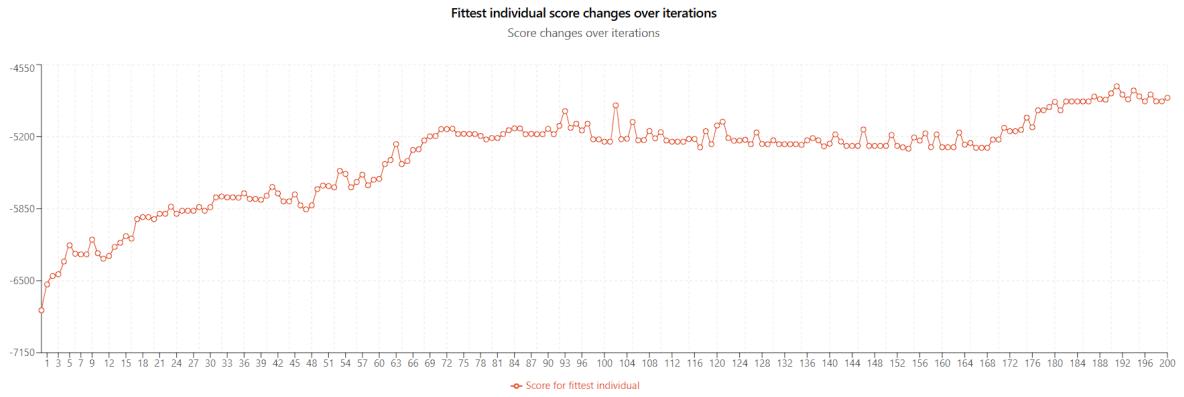
1. Eksperimen 1
  - State awal

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

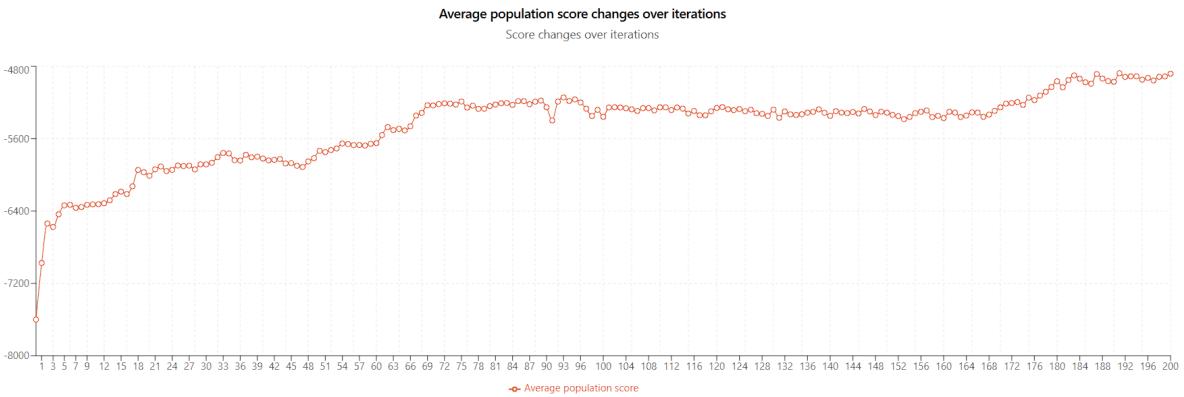
- State akhir

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

- Plot *objective function* (best score from population)



- Plot *objective function* (average score from population)



<b>Nilai objective function terakhir</b>	-4849
<b>Durasi pencarian</b>	105
<b>Jumlah populasi</b>	10
<b>Banyak iterasi</b>	200

## 2. Eksperimen 2

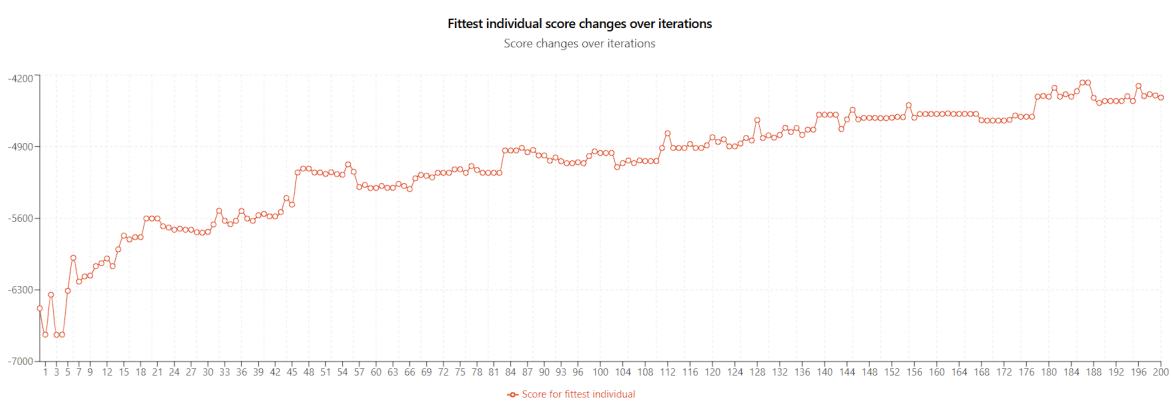
- State awal

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

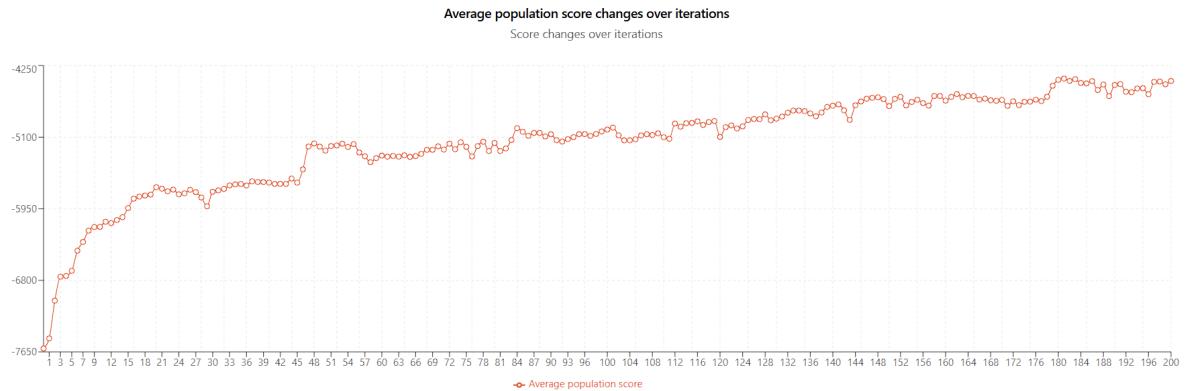
- State akhir

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

- Plot objective function (best score from population)



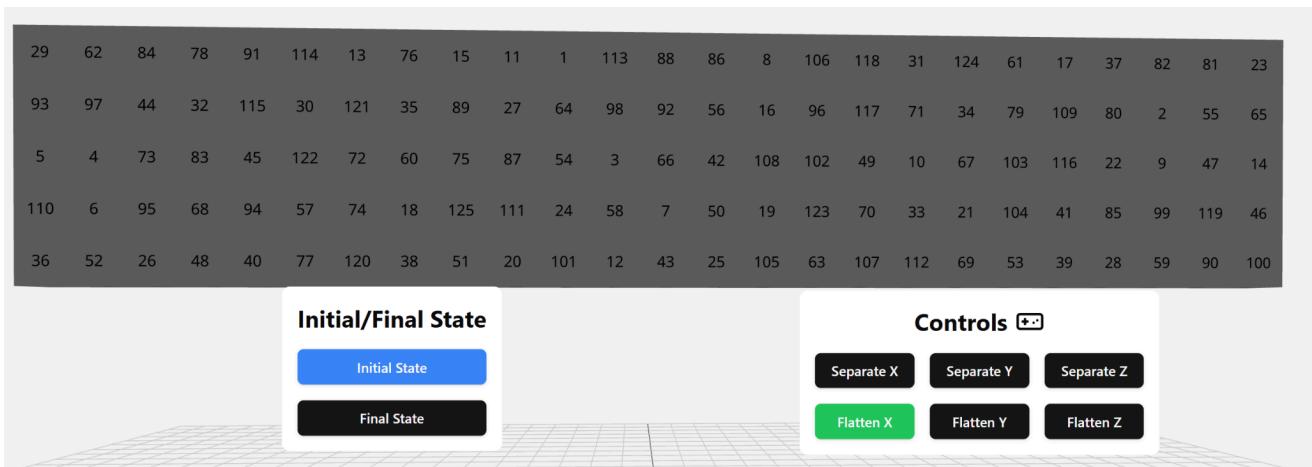
- Plot objective function (average score from population)

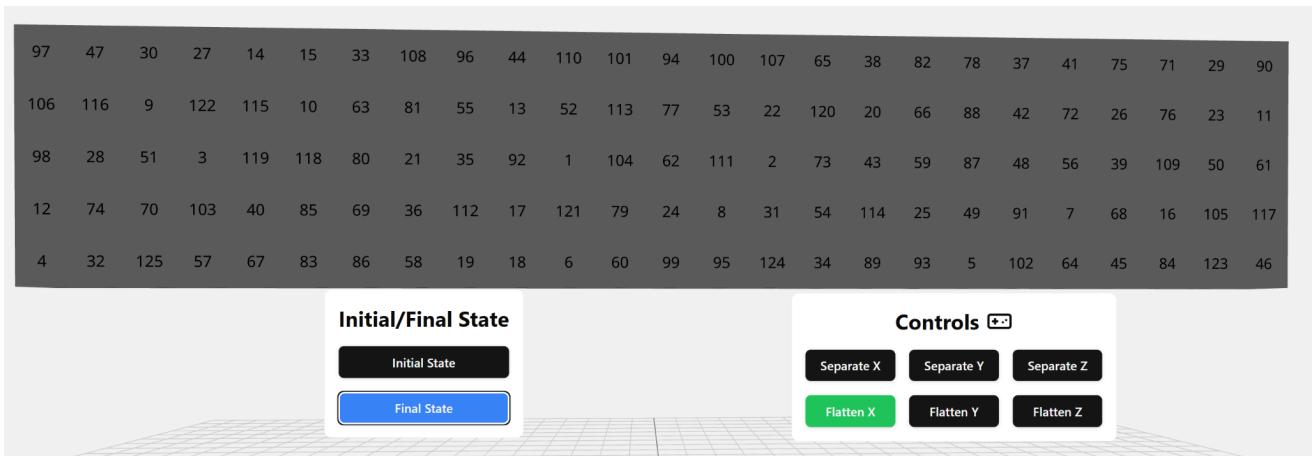


<b>Nilai objective function terakhir</b>	-4420
<b>Durasi pencarian</b>	198ms
<b>Jumlah populasi</b>	10
<b>Banyak iterasi</b>	200

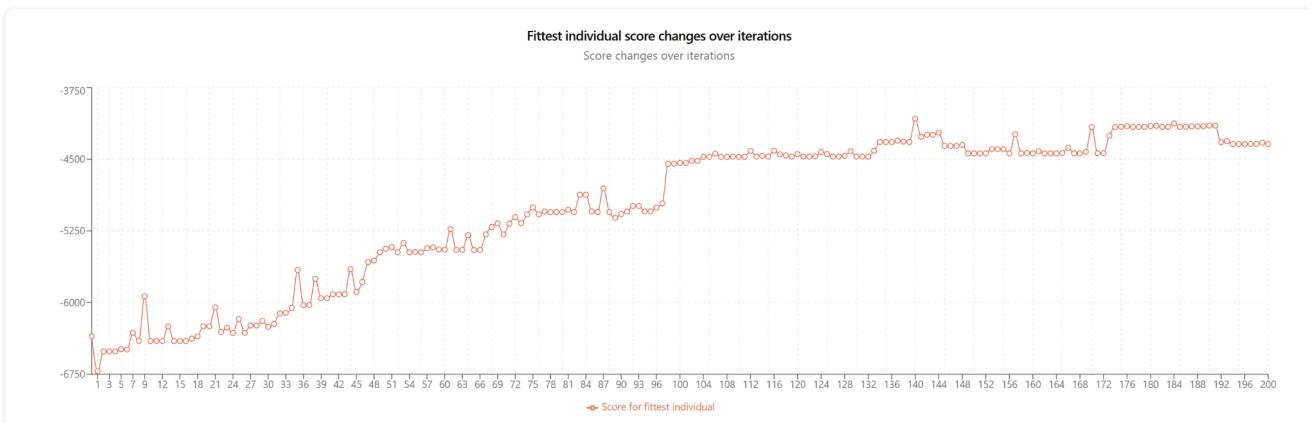
### 3. Eksperimen 3

- State awal

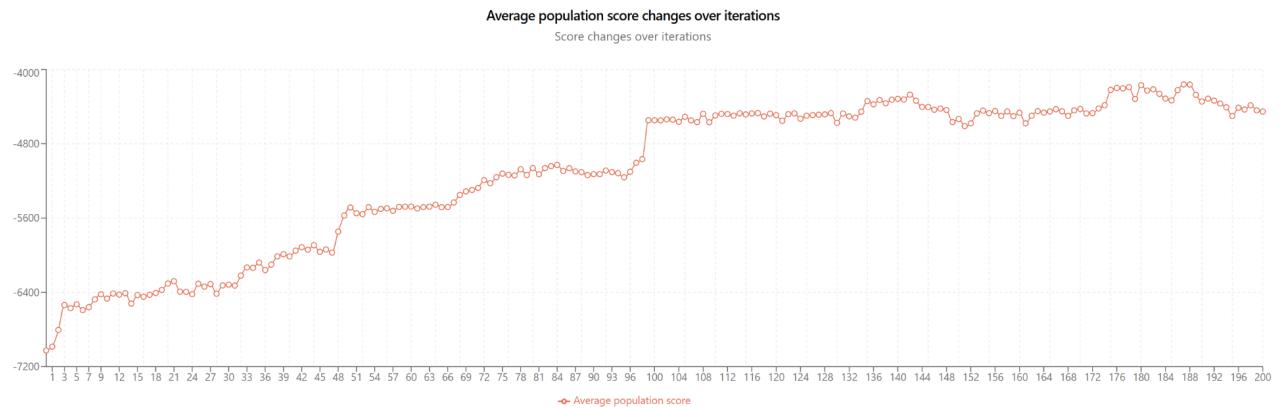




- Plot *objective function* (best score from population)



- Plot *objective function* (average score from population)



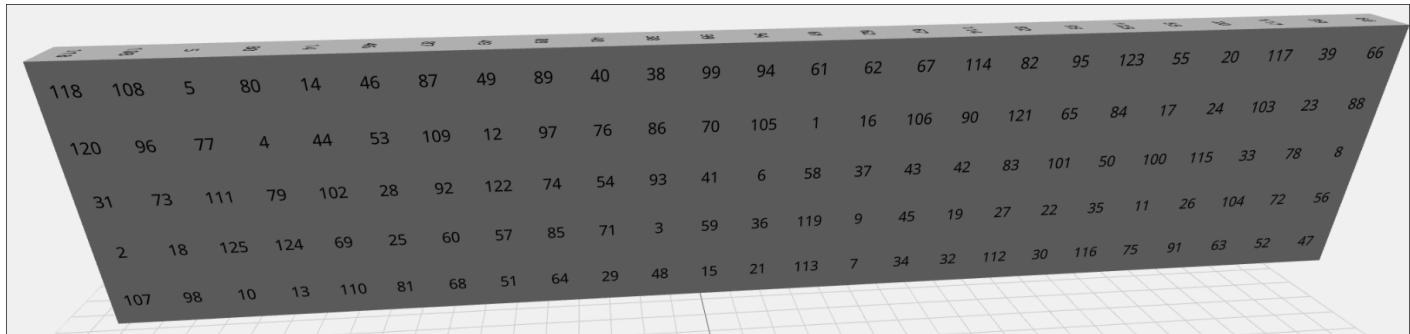
<b>Nilai objective function terakhir</b>	-4341
<b>Durasi pencarian</b>	207 ms
<b>Jumlah populasi</b>	10
<b>Banyak iterasi</b>	200

b. Jumlah Banyak Iterasi sebagai kontrol

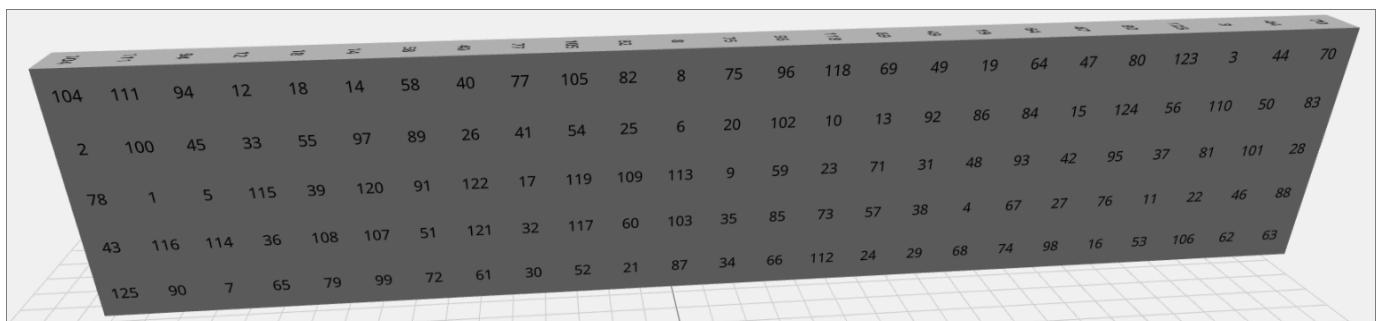
i. **Populasi = 10, Iterasi = 20**

1. Eksperimen 1

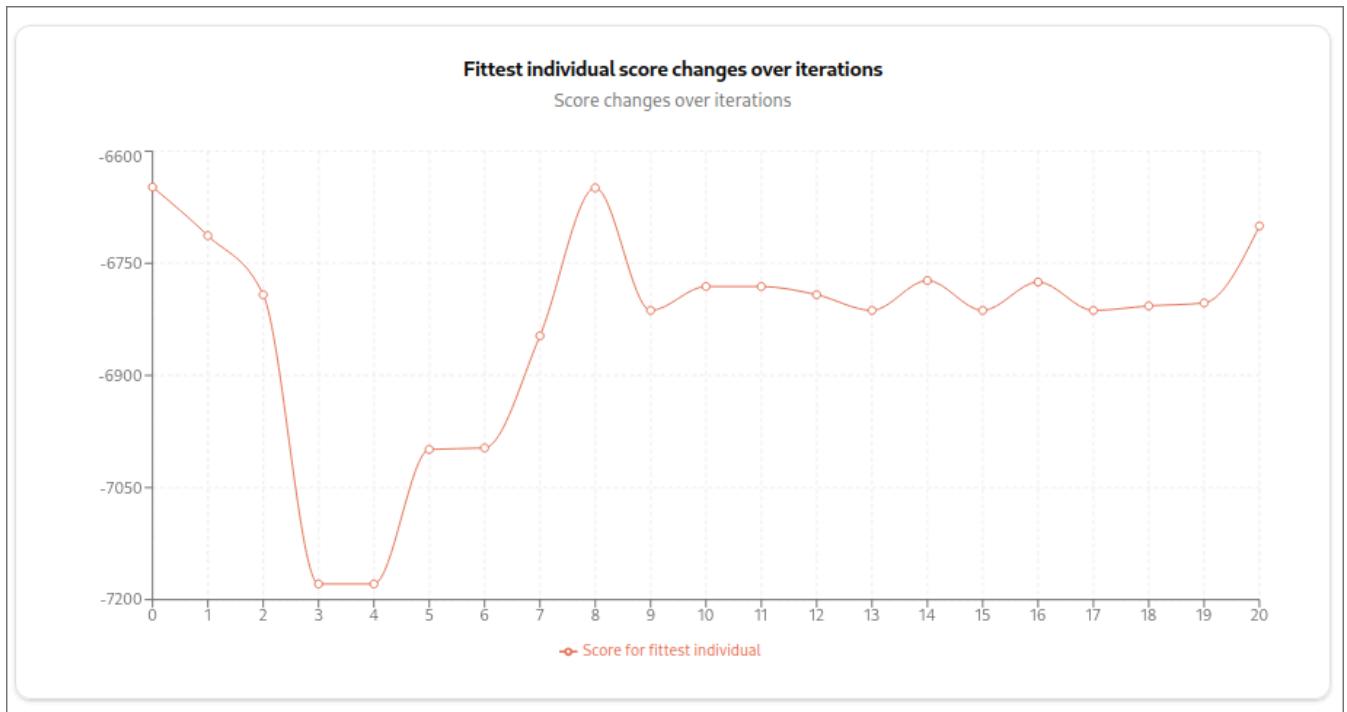
- State awal



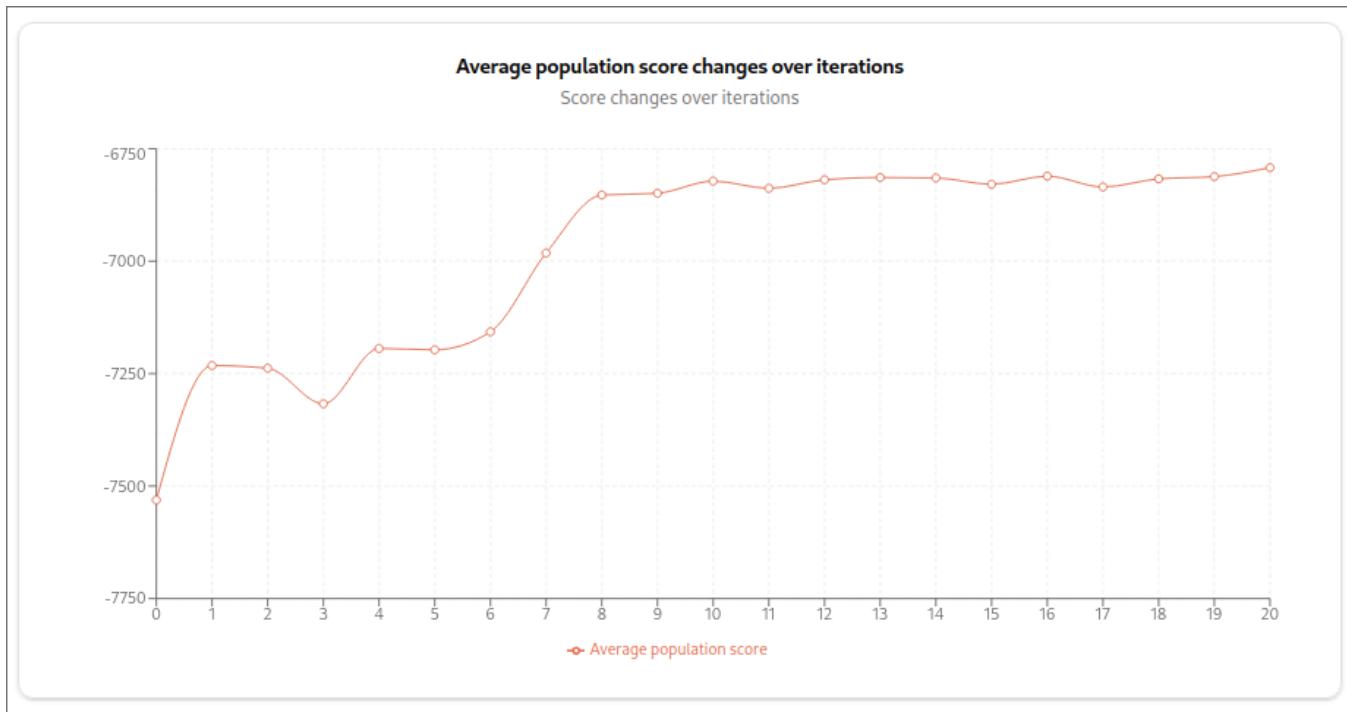
- State akhir



- Plot *objective function* (best score from population)



- Plot *objective function* (average score from population)

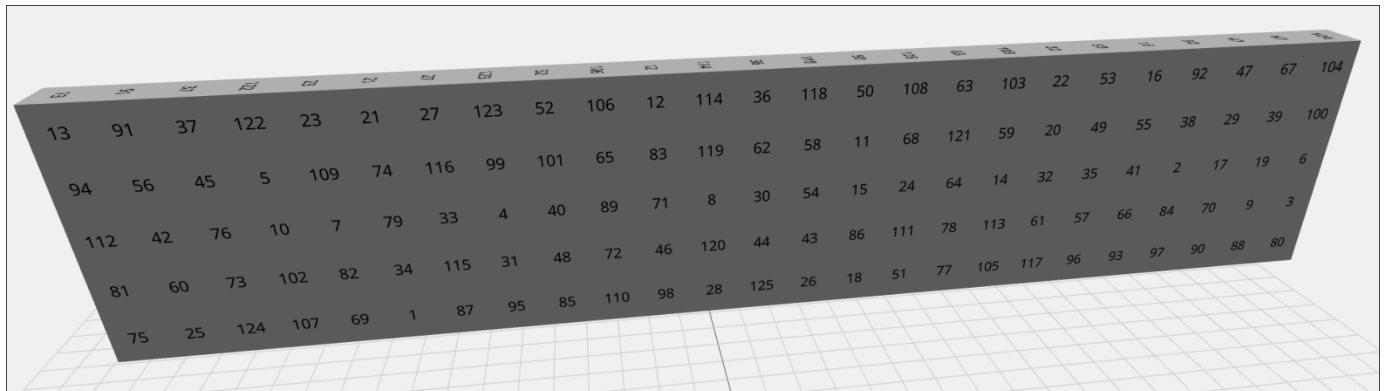


Nilai <b>objective function</b> terakhir	-6700
--	-------

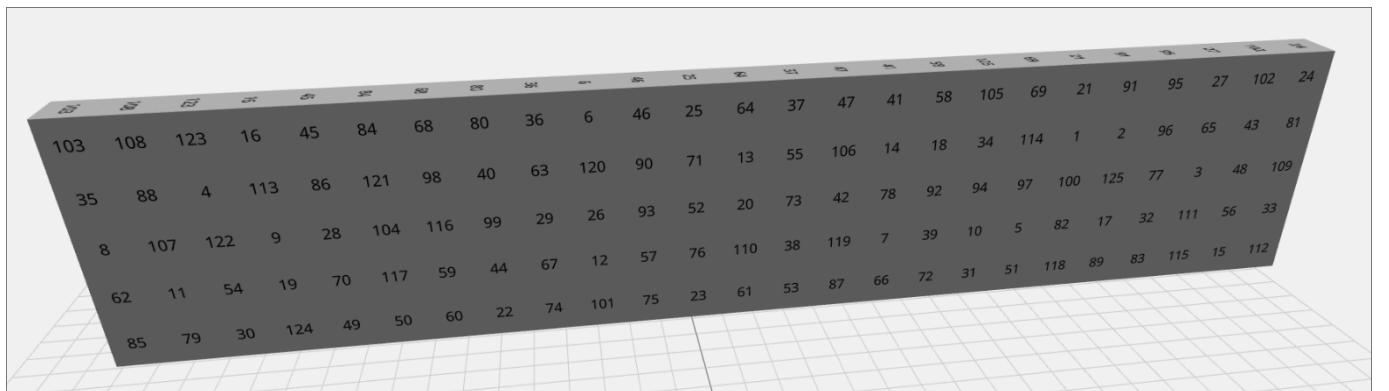
<b>Durasi pencarian</b>	7 ms
<b>Jumlah populasi</b>	10
<b>Banyak iterasi</b>	20

## 2. Eksperimen 2

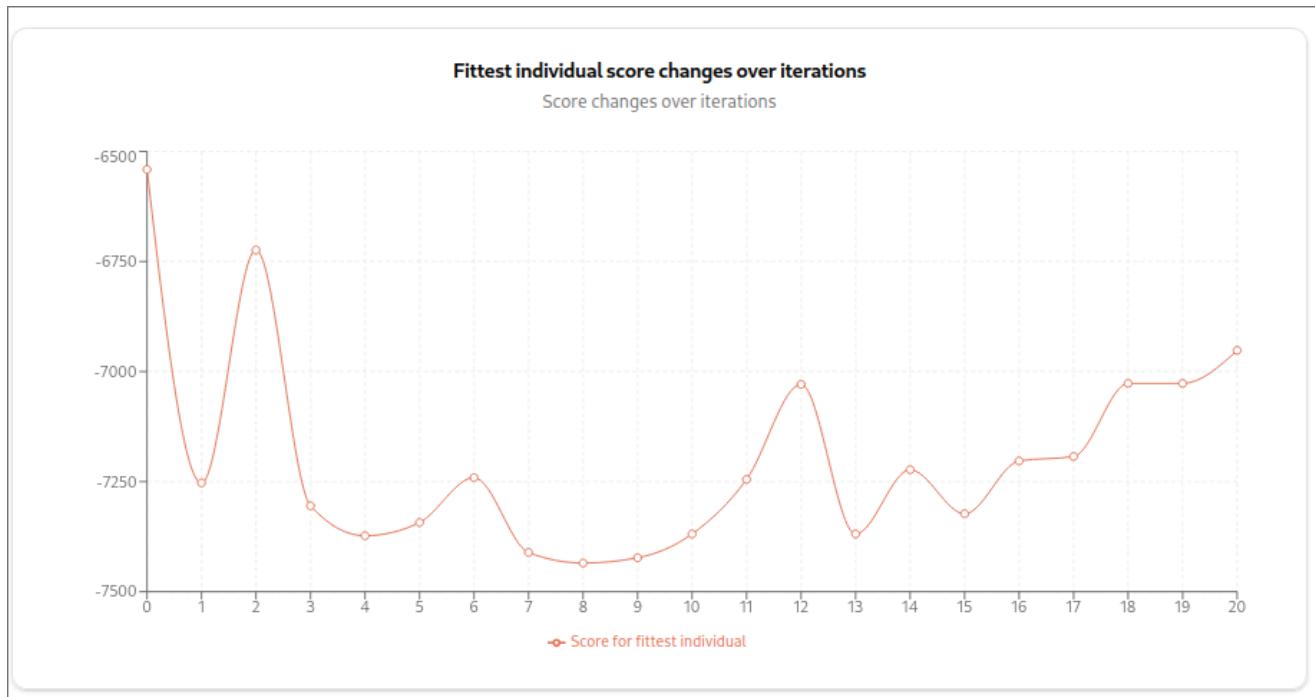
- State awal



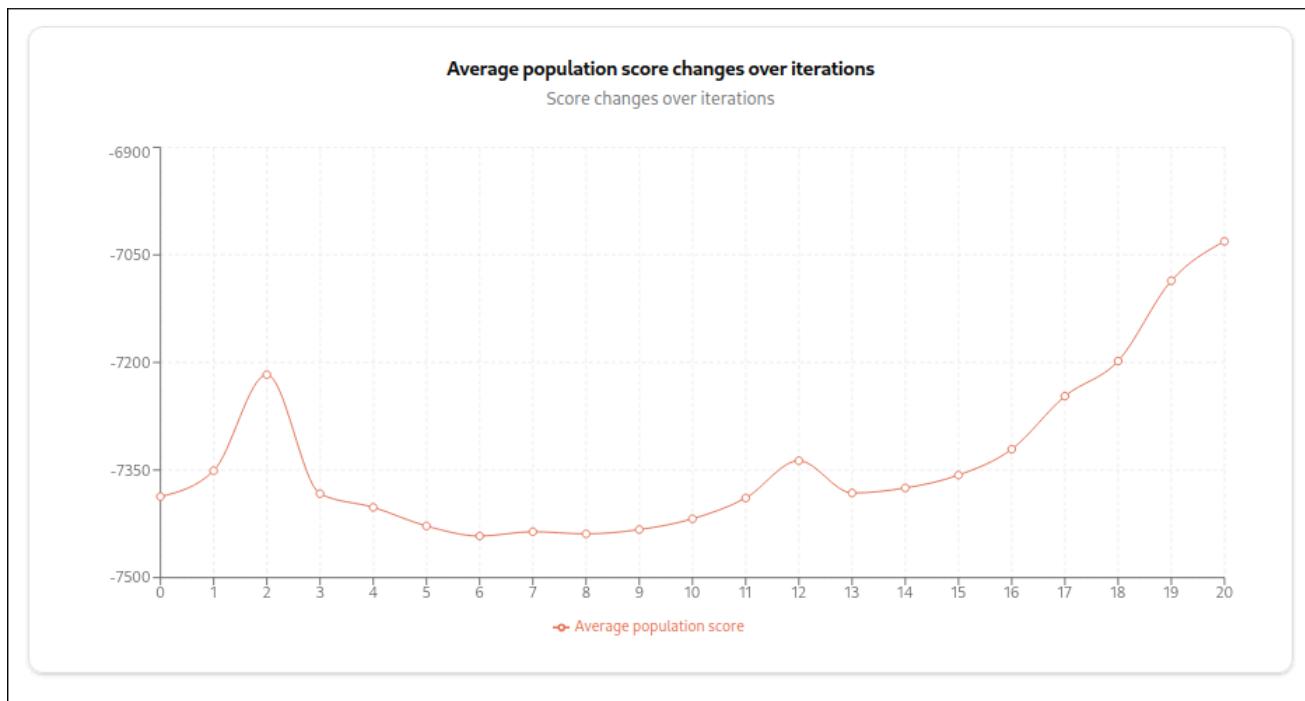
- State akhir



- Plot *objective function* (best score from population)



- Plot *objective function* (average score from population)



Nilai <b>objective function</b> terakhir	-6952
Durasi pencarian	7 ms

Jumlah populasi	10
Banyak iterasi	20

### 3. Eksperimen 3

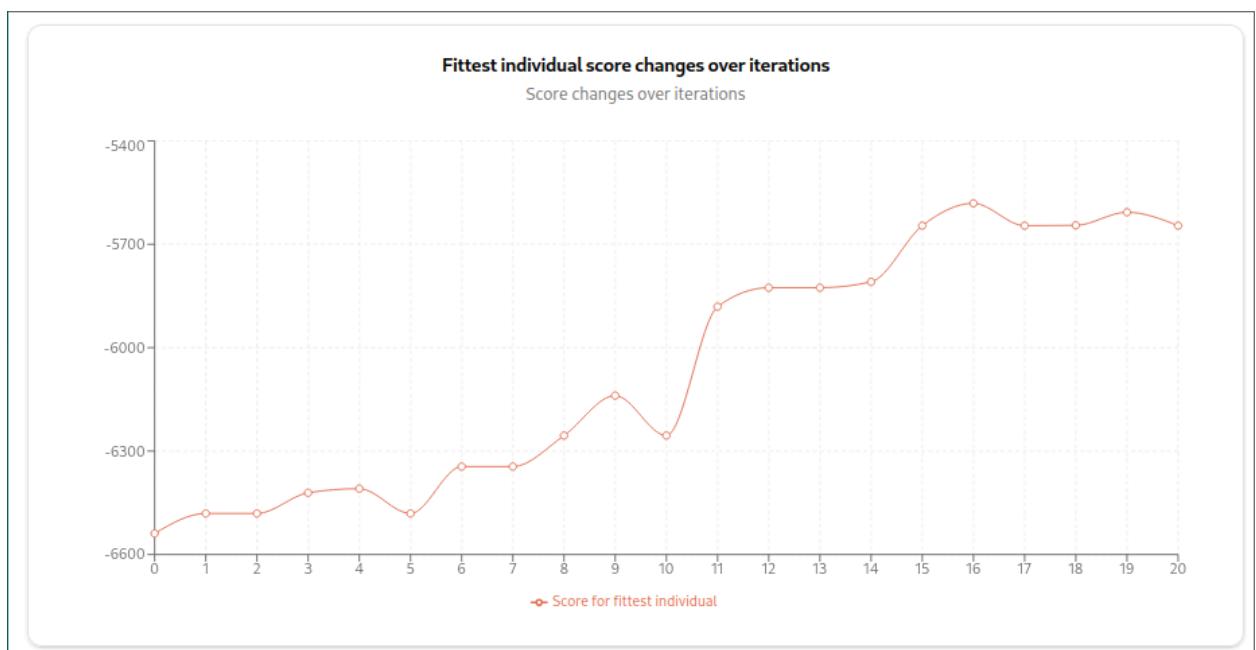
#### - State awal

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

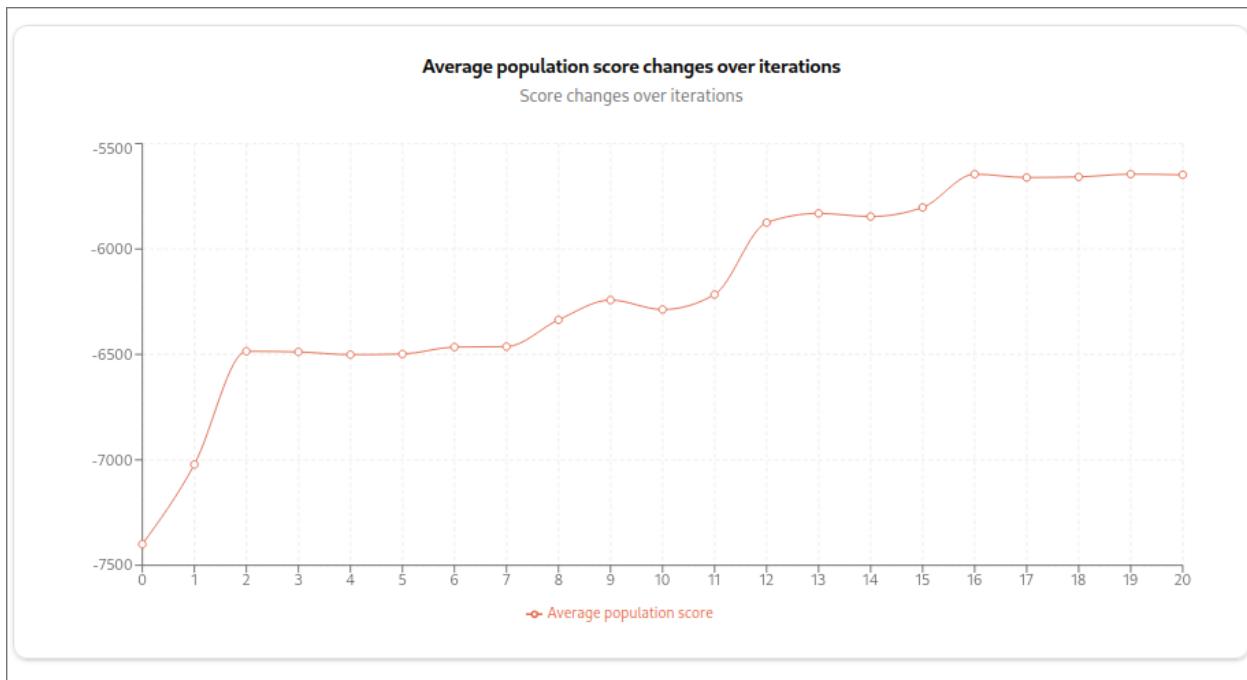
#### - State akhir

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

#### - Plot objective function (best score from population)



- Plot *objective function* (average score from population)

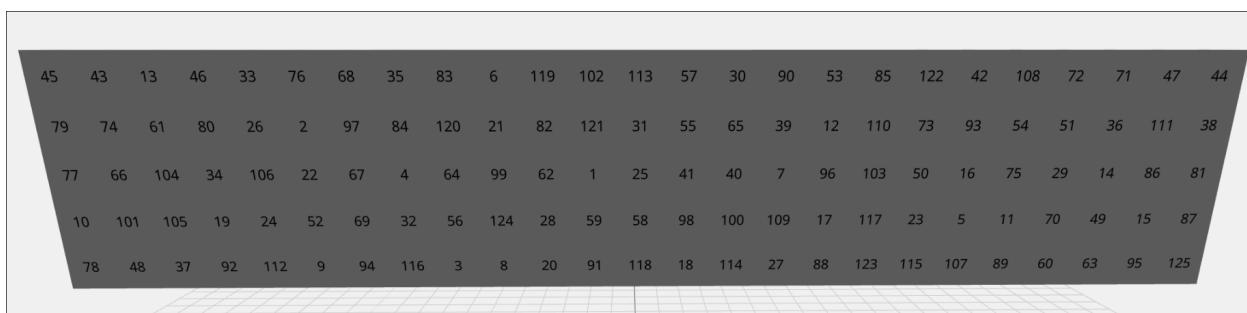


Nilai <i>objective function</i> terakhir	-5646
Durasi pencarian	7 ms
Jumlah populasi	10
Banyak iterasi	20

## ii. Populasi = 100, Iterasi = 20

### 1. Eksperimen 1

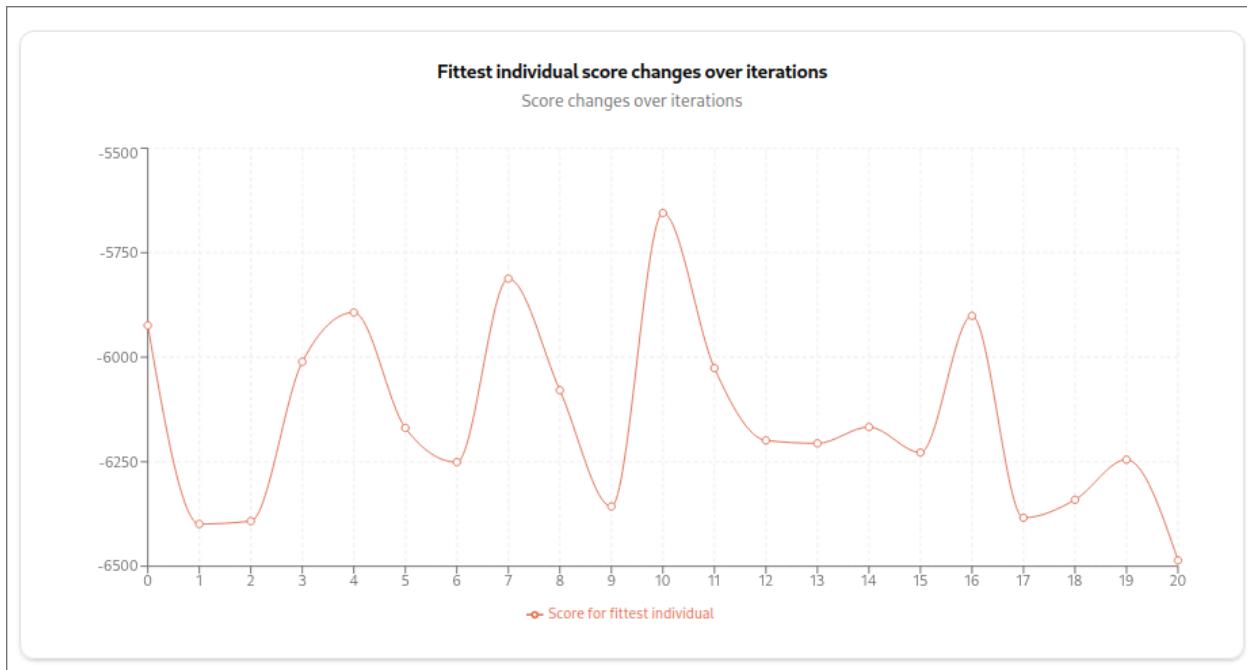
- State awal



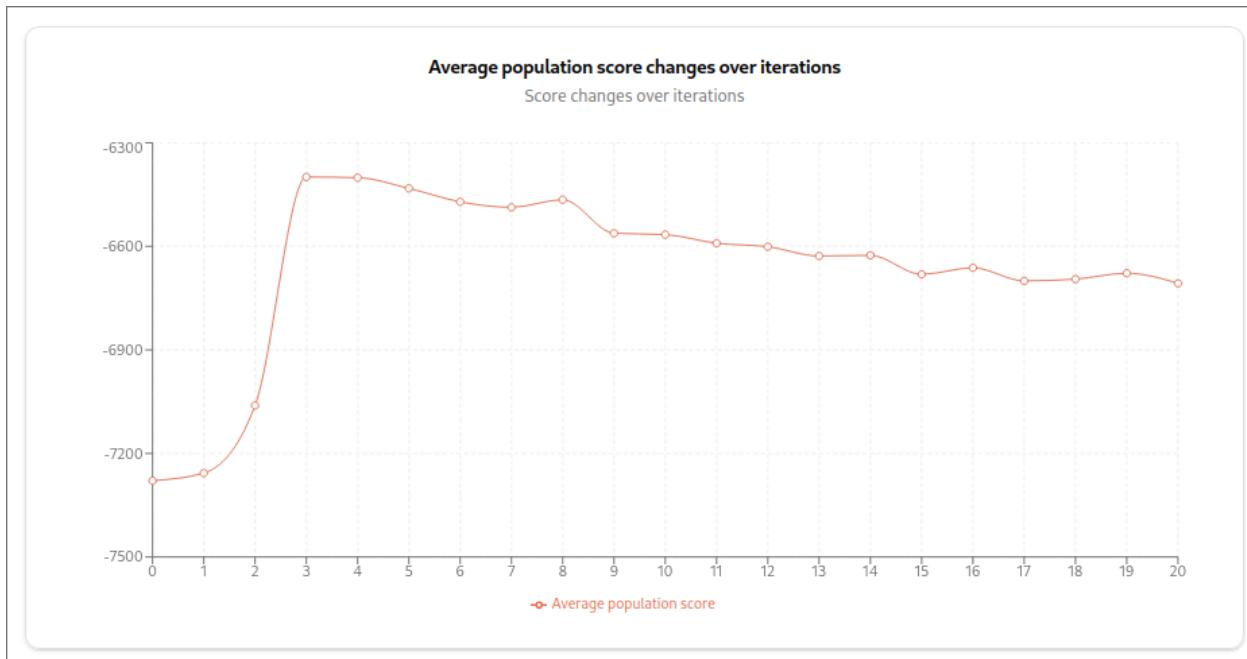
- State akhir

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

- Plot *objective function* (best score from population)



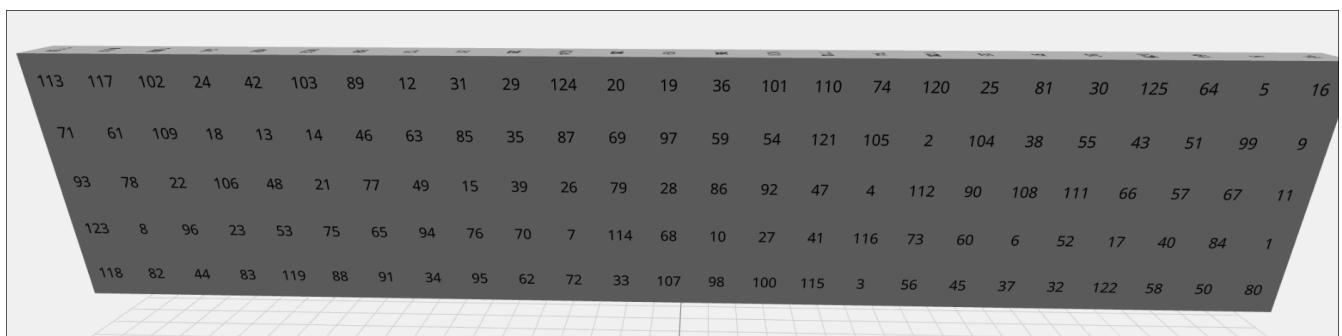
- Plot *objective function* (average score from population)



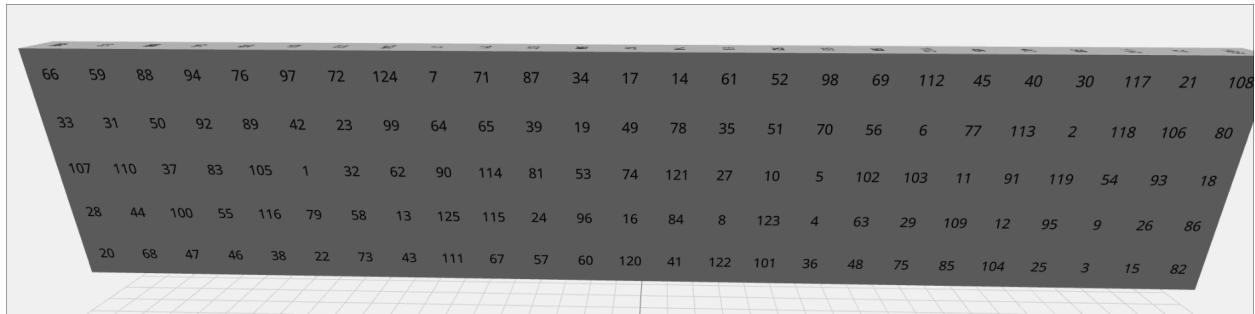
<b>Nilai objective function terakhir</b>	-6486
<b>Durasi pencarian</b>	76 ms
<b>Jumlah populasi</b>	100
<b>Banyak iterasi</b>	20

## 2. Eksperimen 2

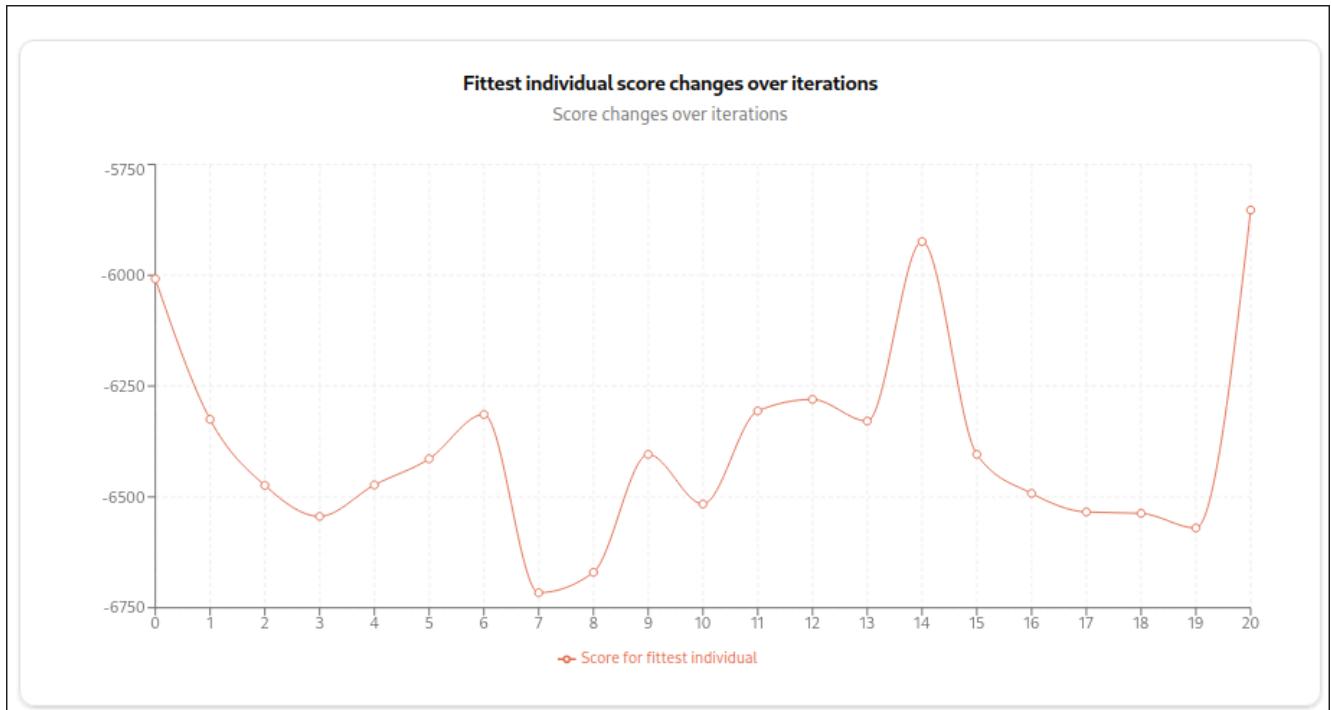
- State awal



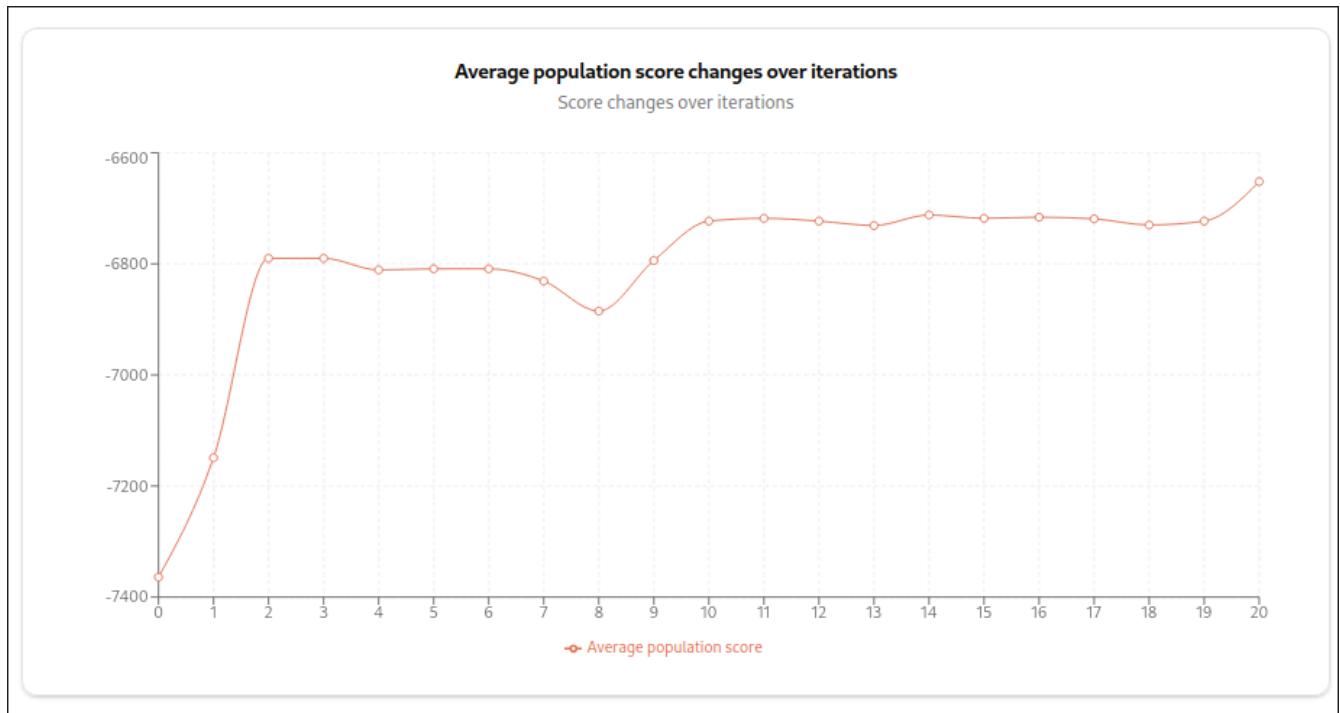
- State akhir



- Plot objective function (best score from population)



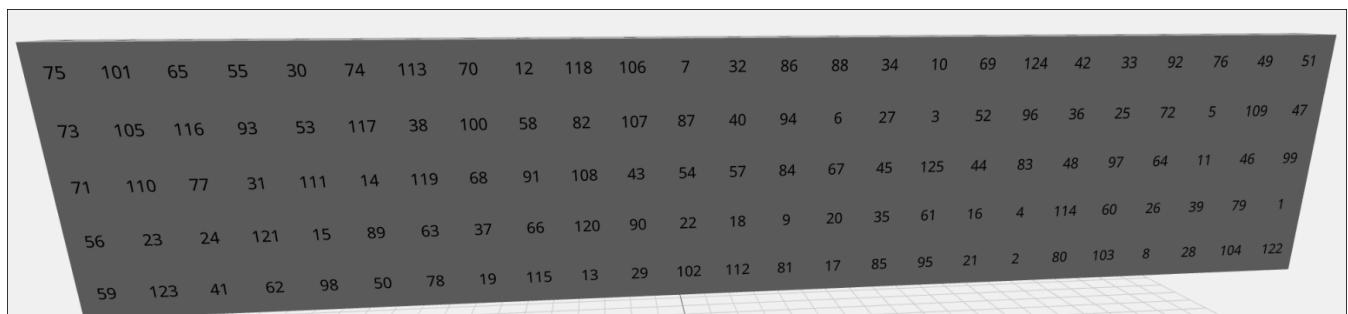
- Plot objective function (average score from population)



<b>Nilai objective function terakhir</b>	-5853
<b>Durasi pencarian</b>	77 ms
<b>Jumlah populasi</b>	100
<b>Banyak iterasi</b>	20

### 3. Eksperimen 3

- State awal



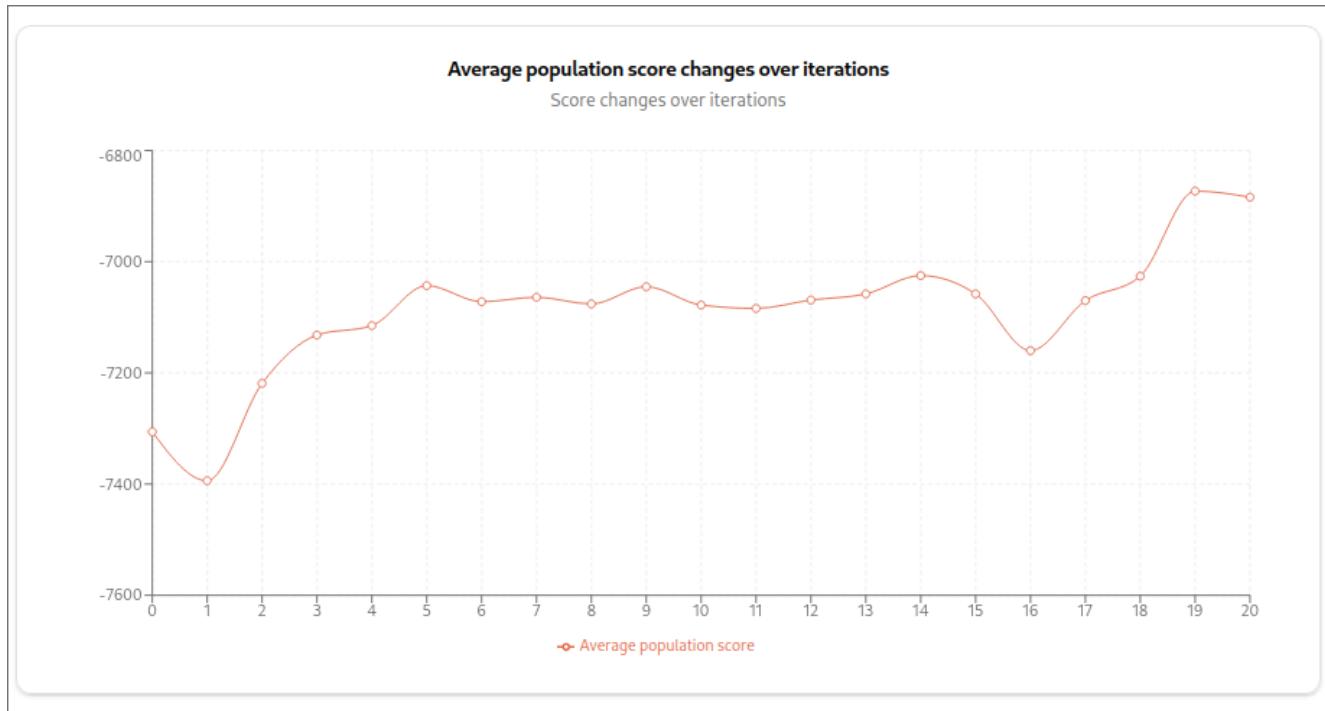
- State akhir

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

- Plot objective function (best score from population)



- Plot objective function (average score from population)



<b>Nilai objective function terakhir</b>	-6481
<b>Durasi pencarian</b>	76 ms
<b>Jumlah populasi</b>	100
<b>Banyak iterasi</b>	20

### iii. Populasi = 200, Iterasi = 20

#### 1. Eksperimen 1

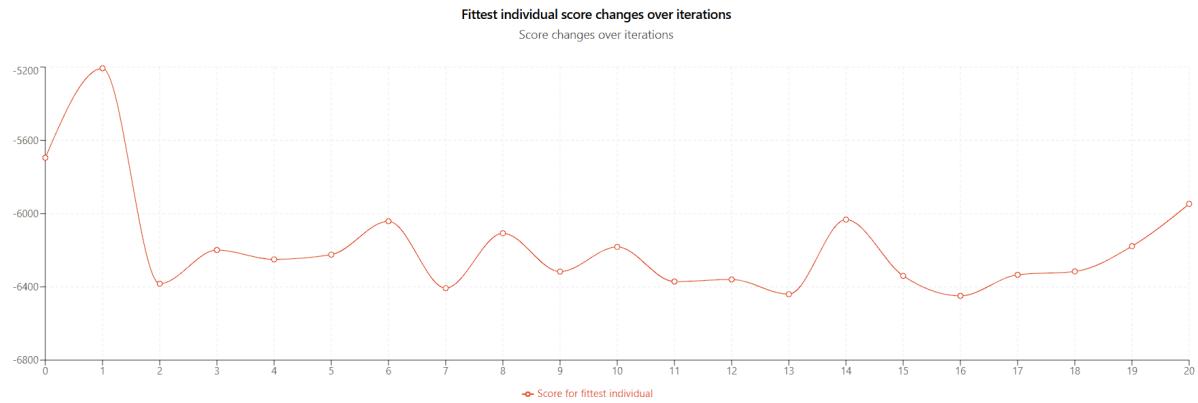
- State awal

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

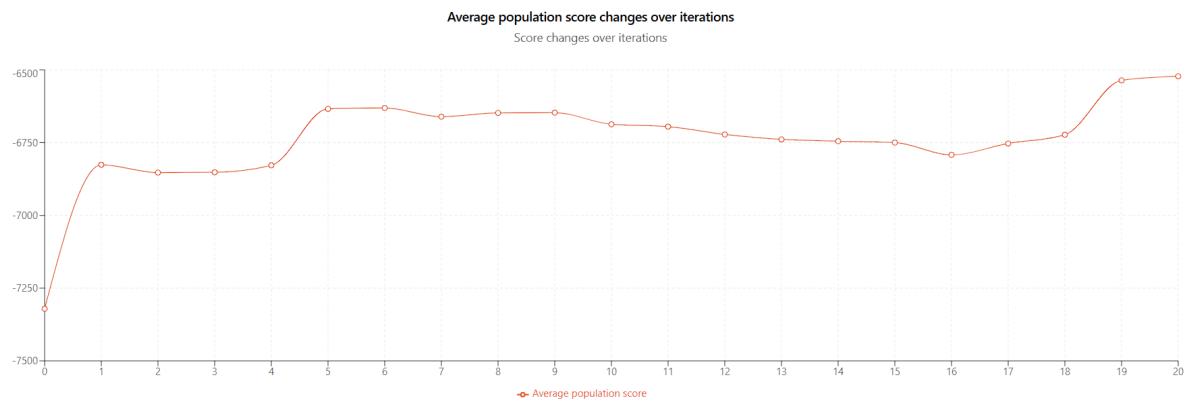
- State akhir

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

- Plot *objective function* (best score from population)



- Plot *objective function* (average score from population)



<b>Nilai <i>objective function</i> terakhir</b>	<b>-5947</b>
<b>Durasi pencarian</b>	<b>266ms</b>
<b>Jumlah populasi</b>	<b>200</b>
<b>Banyak iterasi</b>	<b>20</b>

## 2. Eksperimen 2

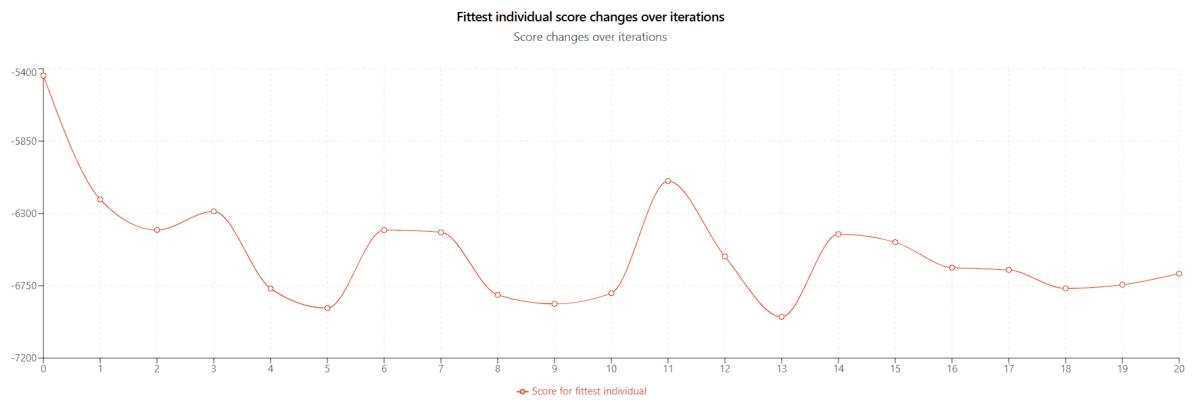
- State Awal

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

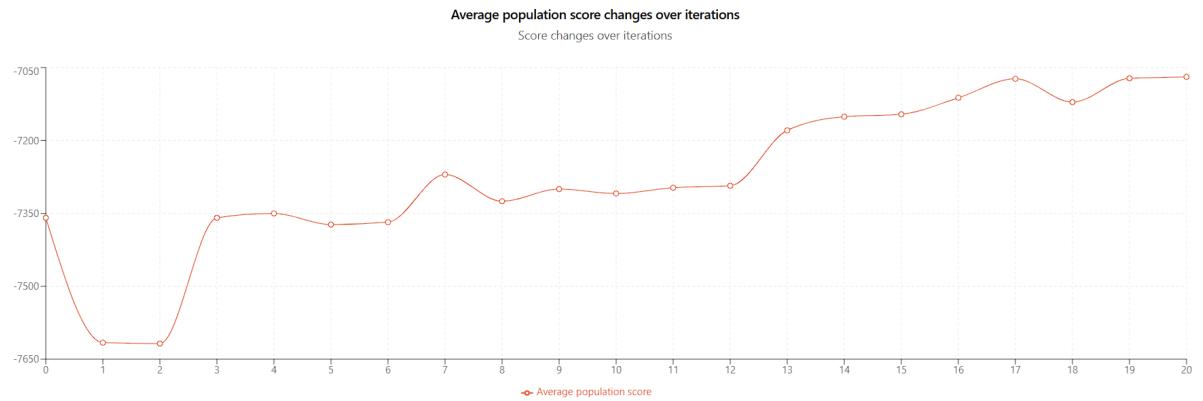
- State Akhir

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

- Plot *objective function* (best score from population)



- Plot *objective function* (average score from population)



<b>Nilai <i>objective function</i> terakhir</b>	-6674
<b>Durasi pencarian</b>	226ms
<b>Jumlah populasi</b>	200
<b>Banyak iterasi</b>	20

### 3. Eksperimen 3

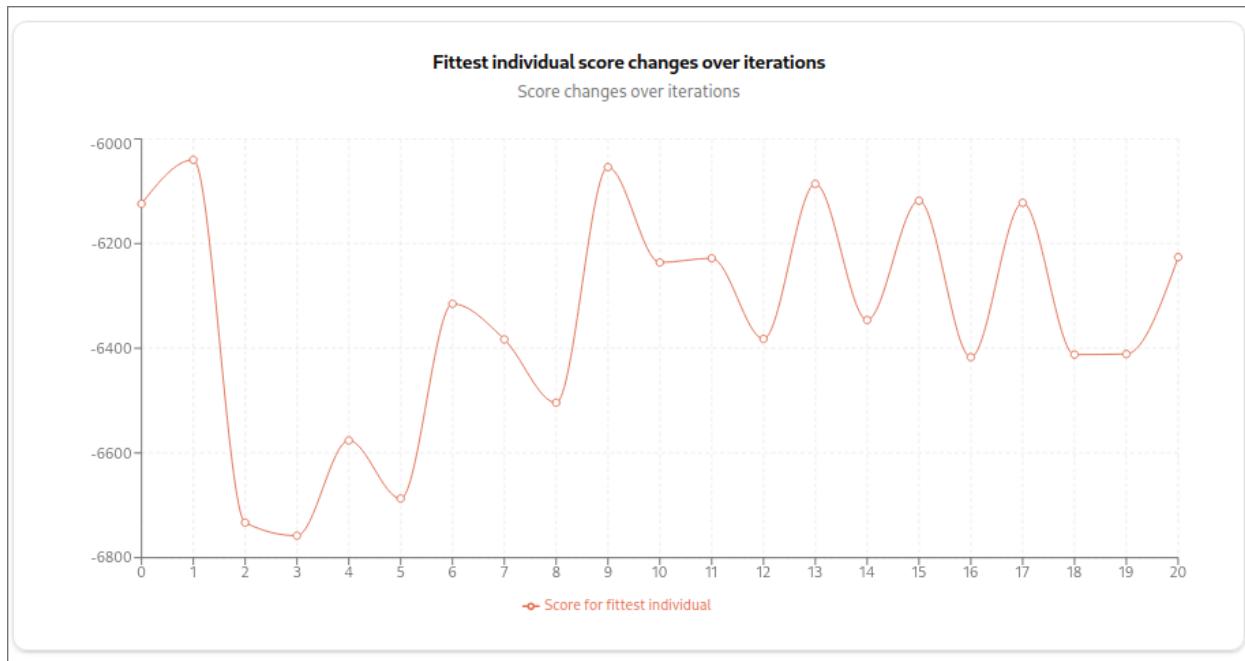
- State awal

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

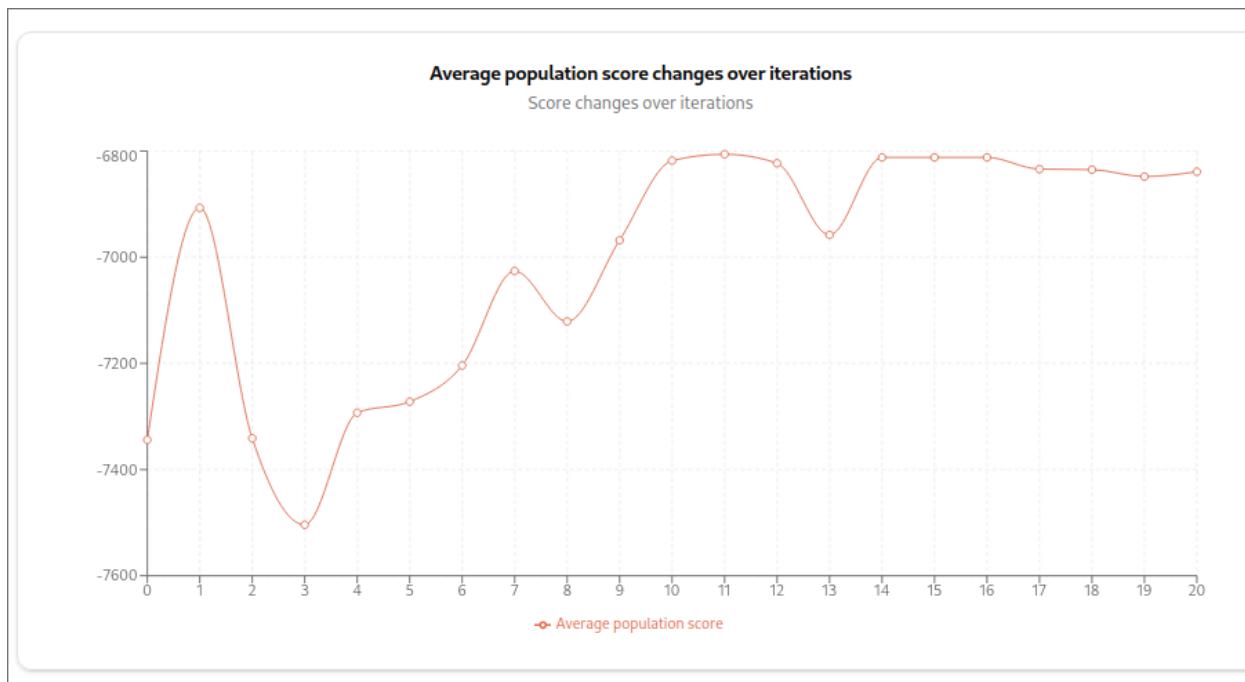
- State akhir

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

- Plot *objective function* (best score from population)



- Plot *objective function* (average score from population)



<b>Nilai <i>objective function</i> terakhir</b>	-6226
<b>Durasi pencarian</b>	151 ms
<b>Jumlah populasi</b>	200

## ANALISIS

- Seberapa dekat tiap-tiap algoritma bisa mendekati global optima dan mengapa hasilnya demikian?
  - **Hill-Climbing Steepest Ascent** mendapatkan hasil yang cukup dekat dengan global optima, mendapatkan sekitar -600 sampai -1400 berdasarkan hasil eksperimen. Ini dikarenakan sifat dari algoritma yang sederhana dan *strict* sehingga hanya mengambil successor yang naik saja.
  - **Hill-Climbing with Sideways Move** mendapatkan hasil yang kurang lebih sama dengan steepest ascent secara objective function.
  - **Hill-Climbing Stochastic** memiliki hasil yang sangat buruk dari segi objective function, mendapatkan hasil sekitar -6000 dan -5000 berdasarkan hasil eksperimen. Ini dikarenakan algoritma ini tidak memiliki basis pemilihan dan hanya mengambil successor secara random.
  - **Random Restart Hill Climbing** mendapatkan hasil yang cukup bagus, sekitar -1200 dan -700, tidak jauh beda dengan *steepest ascent*. Ini dikarenakan algoritma ini secara dasar hanyalah *steepest ascent* yang dijalankan berulang-ulang.
  - **Simulated Annealing** mendapatkan hasil paling bagus dari segi objective function, konsisten mendapatkan nilai objective function di atas -300. Ini dikarenakan algoritma ini memperbolehkan pergerakan menurun secara objective function, sehingga membuka kemungkinan untuk menemukan jalan pintas ke path yang lebih tinggi.
  - **Genetic Algorithm** mendapatkan hasil yang cukup buruk, umumnya sekitar -6000 sampai -4000. Ini dikarenakan sifatnya yang sangat *random* dan bergantung pada banyak kriteria, serta *constraint* pada magic cube yang membuat crossover tidak benar-benar menggabungkan dua state.
- Bagaimana perbandingan hasil pencarian tiap-tiap algoritma dengan algoritma local search yang lain?
  - **Hill-Climbing Steepest Ascent**  
Hill Climbing memiliki hasil yang cukup memuaskan meskipun dia rentan untuk *stuck* pada *local maximum*. Selain itu, hasil *objective function* dari *state* akhir terbilang cukup baik dibanding dengan *Stochastic* dan *Genetic*, yakni dari range -1492 dan -756.
  - **Hill-Climbing with Sideways Move**  
Hill Climbing with *Sideway Move* memiliki hasil yang cukup memuaskan meskipun dia rentan untuk *stuck* pada *local maximum*. Selain itu, hasil *objective*

*function* dari *state* akhir terbilang cukup baik dibanding dengan *Stochastic* dan *Genetic*, yakni dari range -1205 dan -650.

- **Hill-Climbing Stochastic**

Hill Climbing Stochastic memiliki hasil yang kurang memuaskan karena *successor* yang dipilih random dan hanya bergantung kepada kebetulan meskipun akan tetap dibandingkan. Akan tetapi, dibanding dengan algoritma lainnya, algoritma ini tidak memberikan keunggulan yang baik. Nilai range *objective function* yang didapat dari eksperimen untuk algoritma ini adalah -6522 dan -5535.

- **Random Restart Hill Climbing**

Random restart memiliki hasil yang cukup memuaskan karena algoritma ini dapat menghindari local maximum dengan melakukan restart yang akan membuatnya kembali pada *state random*. Hal tersebut menguntungkan, tetapi kelemahan algoritma ini adalah ketidakpastian dari *state random* yang dipilih. Bisa jadi *state random* yang terpilih memiliki *successor* yang tidak menjanjikan. Nilai range *objective function* yang didapat dari eksperimen untuk algoritma ini adalah -1227 dan -706.

- **Simulated Annealing**

Simulated Annealing merupakan algoritma yang **paling menjanjikan** untuk persoalan magic cube dan terbukti melalui hasil eksperimen yang dijalankan. Hal tersebut disebabkan algoritma ini menggunakan pendekatan yang memanfaatkan peluang untuk menghindari *local maximum* sekaligus mengikuti aturan. Alhasil, algoritma ini menjadi algoritma yang memiliki hasil yang paling efektif dari keseluruhan. Untuk range hasil eksperimen pada algoritma Simulated Annealing adalah -278 dan -257 .

- **Genetic Algorithm**

Genetic Algorithm memiliki hasil yang kurang memuaskan karena algoritma ini memiliki ketidakpastian pada prosesnya yakni pada saat penukaran dan mutasi. Pada saat itu, proses yang terlibat tidak menjamin hasil yang memiliki *objective function value* lebih baik. Hasil eksperimen pada Genetic Algorithm adalah -6952 dan - 4432 .

- Bagaimana perbandingan durasi proses pencarian tiap algoritma relatif terhadap algoritma lainnya?

- **Hill-Climbing Steepest Ascent**

Hill climbing steepest ascent memiliki durasi hasil eksperimen dengan range 391 hingga 698 milliseconds. Hal ini bisa disebabkan karena hill climbing menggunakan loop untuk mencari best successor sehingga akan memakan waktu.

- **Hill-Climbing with Sideways Move**

Hill climbing sideways memiliki durasi hasil eksperimen dengan range 920 hingga 1313 milliseconds. Algoritma ini memiliki hasil durasi yang lebih lama dengan Hill-Climbing with Steepest Ascent dikarenakan program harus mengecek conditional untuk maksimum sideways move.

- **Hill-Climbing Stochastic**

Hill climbing stochastic memiliki durasi hasil eksperimen dengan range 0 hingga 11 milliseconds. Ini dikarenakan proses dalam pemilihan neighbor yang random sehingga tidak perlu dilakukan iterasi tambahan yang akan memakan waktu lama. **Hill Climbing Stochastic** memiliki durasi yang paling singkat dibanding dengan algoritma lainnya dari hasil eksperimen.

- **Random Restart Hill Climbing**

Random restart hill climbing memiliki durasi hasil eksperimen dengan range 2310 hingga 2634 milliseconds. Random restart memiliki durasi yang cukup tinggi hal ini dikarenakan algoritma **Random Restart Hill Climbing** mirip dengan steepest ascent tetapi dia akan menjalankan steepest ascent dengan masukan restart pengguna. Sehingga lama durasi dari simulated annealing  $\approx \text{maxRestart} * \text{Durasi Steepest Ascent}$ .

- **Simulated Annealing**

Simulated annealing memiliki durasi hasil eksperimen dengan range 3584 hingga 3861 milliseconds. Algoritma simulated annealing memiliki algoritma yang mirip dengan Stochastic Hill Climbing dalam mencari neighbor. Akan tetapi algoritma simulated annealing memiliki perhitungan tambahan dengan rumus probability. Hal ini akan menambah durasi secara signifikan dibanding dengan algoritma **Hill Climbing Stochastic**. **Simulated Annealing** memiliki durasi yang paling lama dibanding algoritma lainnya dari hasil eksperimen.

- **Genetic Algorithm**

Genetic Algorithm memiliki durasi yang beragam tergantung dengan jumlah populasi dan iterasinya. Genetic algorithm memiliki durasi dengan range 7 hingga 266 ms. Genetic algorithm memiliki durasi yang relatif kecil hal ini bisa dikarenakan pengujian menggunakan nilai populasi dan iterasi yang cukup kecil dan juga genetic algorithm mengambil populasi awal secara random sehingga dapat mengurangi durasi dibanding algoritma lainnya yang membutuhkan loop untuk mencari best successor

- Seberapa konsisten hasil akhir yang didapatkan dari tiap-tiap eksperimen yang dilakukan?
  - **Hill-Climbing Steepest Ascent** memiliki hasil akhir yang cukup variatif dan tidak konsisten, kadang mendapatkan hasil akhir sekitar -1200, dan kadang hasil akhirnya sampai -600.

- **Hill-Climbing with Sideways Move** juga memiliki hasil akhir yang tidak konsisten, sama seperti *steepest ascent* dan karena alasan yang sama, karena kedua algoritma tidak terlalu berbeda.
- **Hill-Climbing Stochastic** memiliki hasil yang sangat tidak konsisten, ini dikarenakan sifat algoritmanya yang benar-benar *random*.
- **Random Restart Hill Climbing** memiliki hasil yang tidak konsisten, mirip dengan *steepest ascent* karena pada dasarnya hanya *steepest ascent* yang dijalankan berulang-ulang.
- **Simulated Annealing** memiliki hasil eksperimen yang sangat konsisten dan bagus, selalu mendapatkan *objective function* akhir sekitar -250.
- **Genetic Algorithm** memiliki hasil yang variatif dan konsisten, namun ini dikarenakan banyaknya variasi parameter. Jika parameter yang digunakan sama, hasil akhirnya cukup konsisten namun masih dapat cukup bervariasi.
- Bagaimana pengaruh banyak iterasi dan jumlah populasi terhadap hasil akhir pencarian pada Genetic Algorithm?
  - Pada kasus ketika **populasi tetap dan iterasi divariasikan**, terdapat perubahan besar pada variasi iterasi. Pada eksperimen dengan iterasi = 10, didapatkan hasil akhir sekitar -6000. Ketika dibuat menjadi iterasi = 100 dan iterasi = 200, didapatkan **peningkatan besar** menjadi sekitar -4000 untuk hasil akhir.
  - Pada kasus **iterasi tetap dan populasi divariasikan**, tidak didapatkan perubahan besar pada masing-masing eksperimen. Didapatkan ketika iterasi = 20 dan populasi = 10, 100, 200, hasil akhir objective function selalu sekitar -6000.

Sehingga dapat disimpulkan bahwa **Jumlah iterasi jauh lebih berpengaruh pada kinerja algoritma genetic algorithm** dibanding jumlah populasi.

### **III. Kesimpulan dan Saran**

Berdasarkan implementasi dan hasil eksperimen yang dilakukan pada laporan ini, dapat disimpulkan bahwa algoritma *local search* seperti *steepest ascent hill climbing* serta variasinya berupa *sideways/random restart/stochastic*, lalu *simulated annealing* dan *genetic algorithm* dapat digunakan untuk menyelesaikan permasalahan *diagonal magic cube*.

Walaupun tidak ada algoritma yang dapat menemukan solusinya, terdapat beberapa algoritma seperti *simulated annealing* dan *random-restart hill climbing* yang dapat cukup mendekati global optima. Terdapat beberapa algoritma yang cukup lumayan dalam menyelesaikan permasalahan, seperti *steepest ascent* dan variasi *sideways*. Untuk algoritma seperti *stochastic hill climbing* serta *genetic algorithm* masih cukup buruk dalam permasalahan ini, karena sangat berbasis pada probabilitas dan tidak konsisten.

Saran dari penulis pada pembaca adalah agar dapat memanfaatkan algoritma-algoritma *local search* tersebut pada permasalahan-permasalahan lain yang dapat ditemukan sehari-hari. Perlu diperhatikan bahwa beragam algoritma yang digunakan memiliki kelebihannya masing-masing pada kasus-kasus tertentu. Untuk permasalahan *diagonal magic cube*, disarankan untuk menggunakan algoritma *simulated annealing* atau *random-restart hill climbing* yang cukup bagus dalam kinerjanya.

## IV. Pembagian Tugas

NIM	Nama	Tugas
13522122	Maulvi Ziadinda Maulana	Pembuatan <i>front-end</i> aplikasi, pembuatan infrastruktur <i>backend</i> aplikasi, struktur data pada <i>backend</i> aplikasi, integrasi <i>backend</i> dengan <i>frontend</i> , pembuatan algoritma <i>Simulated Annealing</i>
13522149	Muhammad Dzaki Arta	Pembuatan Algoritma Hill-Climbing pembuatan Steepest Ascent, pembuatan Hill-Climbing Sideways Move, pembuatan Hill-Climbing Stochastic. Integrasi algoritma-algoritma diatas dengan <i>frontend</i>
13522150	Albert Ghazaly	Pembuatan algoritma <i>Random Restart</i> , pembuatan fitur <i>video player</i> , pembuatan integrasi <i>video player</i> dengan <i>frontend</i>
13522158	Muhammad Rasheed Qais Tandjung	Pembuatan <i>Genetic Algorithm</i> . integrasi <i>frontend</i> dengan <i>backend</i>

## **V. Referensi**

- <https://www.magischvierkant.com/three-dimensional-eng/magic-features/>
- <https://www.trump.de/magic-squares/magic-cubes/cubes-1.html>
- [https://en.wikipedia.org/wiki/Magic\\_cube](https://en.wikipedia.org/wiki/Magic_cube)