LAPORAN TUGAS BESAR 1

Pencarian Solusi Diagonal Magic Cube dengan Local Search

Disusun Oleh:

Kelompok 52

Yasra Zhafirah	(18222002)
Vini Putiasa	(18222030)
Benedicta Eryka Santosa	(18222031)
Kerlyn Deslia Andeskar	(18222090)

IF3070 Dasar Inteligensi Artifisial

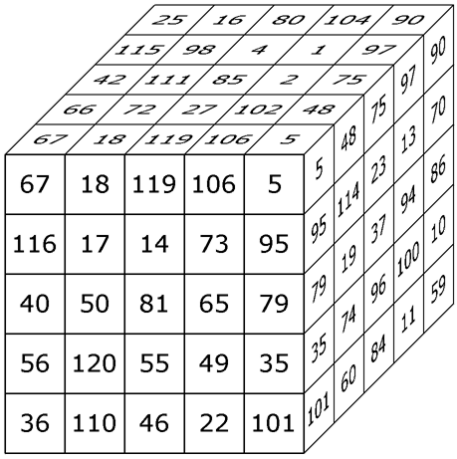
Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2024/2025

DAFTAR ISI

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. Angka-angka pada tersusun sedemikian rupa sehingga properti-properti berikut terpenuhi:

- 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).
- Jumlah angka-angka untuk setiap baris sama dengan *magic number*.
- Jumlah angka-angka untuk setiap kolom sama dengan *magic number*.
- Jumlah angka-angka untuk setiap tiang sama dengan *magic number*.
- Jumlah angka-angka untuk seluruh diagonal ruang pada kubus sama dengan *magic number*.
- Jumlah angka-angka untuk seluruh diagonal pada suatu potongan bidang dari kubus sama dengan *magic number*.

Persoalan ini mengenai permasalahan *Diagonal Magic Cube* berukuran 5×5×5. *Initial state* dari suatu kubus adalah susunan angka 1 hingga 5^3 secara acak. Kemudian, tiap iterasi pada algoritma *local search*, langkah yang boleh dilakukan adalah menukar posisi dari 2 angka pada kubus tersebut (2 angka yang ditukar tidak harus bersebelahan). Khusus untuk *genetic algorithm*,

boleh dilakukan penukaran posisi lebih dari 2 angka sekaligus dalam satu iterasi (tetap hanya menukar posisi 2 angka saja juga diperbolehkan).

Persoalan tersebut membutuhkan pembuatan metode *local search* untuk mencari solusi *Diagonal Magic Cube* yang mencakup *Steepest Ascent Hill-climbing*, *Hill-climbing with Sideways Move*, *Random Restart Hill-climbing*, *Stochastic Hill-climbing*, *Simulated Annealing,* dan *Genetic Algorithm*. Selanjutnya, melakukan eksperimen dengan skema yang telah ditentukan sesuai spesifikasi, lalu melakukan analisis terhadap hasil eksperimen tersebut.

PEMBAHASAN

A. Pemilihan *Objective Function*

Objective function merupakan fungsi yang dipakai untuk mengevaluasi atau mengukur seberapa baik kualitas solusi yang dihasilkan dalam memecahkan atau menyelesaikan sebuah persoalan atau permasalahan. *Objective function* pada persoalan *diagonal magic cube* dapat diartikan sebagai cara dalam mengukur seberapa dekat solusi yang dibuat dengan kondisi ideal, yakni jumlah baris, kolom, tiang, dan diagonal yang penjumlahan angkanya sama dengan *magic number*, yang mana semakin banyak yang sesuai dengan *magic number*, maka semakin baik solusi tersebut. Dalam konteks persoalan *magic cube* ini, *objective function* bertujuan untuk memaksimalkan jumlah baris, kolom, tiang, dan diagonal yang penjumlahan angkanya mendapatkan hasil angka yang sama dengan *magic number*. Atau, untuk menentukan seberapa optimal solusi-solusi *local search* yang dilakukan sehingga hasilnya dapat dibandingkan untuk menentukan jenis *local search* yang dapat menghasilkan solusi paling optimal.

Kami menentukan *objective function* dengan menghitung hasil deviasi atau selisih antara jumlah elemen/angka di setiap baris, kolom, tiang, diagonal ruang, dan diagonal bidang dengan *magic number*. *Magic number* yang dimaksud adalah angka hasil penjumlahan dari setiap baris, kolom, tiang, diagonal ruang, atau diagonal bidang. Semakin kecil deviasi atau selisihnya, maka semakin optimal solusi tersebut dan sebaliknya. Jika hasil deviasi dari *objective function*-nya adalah 0, maka solusi tersebut merupakan global optimum.

Di samping itu, cara menghitung *magic number* (M) pada suatu kubus yang memiliki rusuk n dapat dilakukan dengan rumus sebagai berikut.

$$M(n) = \frac{n\,(n^3 + 1)}{2}$$

Dengan keterangan bahwa M adalah *function magic number* dan n merupakan panjang sisi kubus. Cara tersebut diperoleh dari alasan bahwa karena semua bilangan berurutan dan tidak berulang, median dan rata-rata dari bilangan-bilangan ini akan sama. Kita bisa

menghitungnya dengan menggunakan rumus $\frac{(n^3 + 1)}{2}$. Sesuai dengan aturan *magic cube*, untuk menentukan *magic number* pada *magic cube* dengan ukuran n, kita perlu mengalikan rata-rata ini dengan n sehingga diperoleh rumus $M(n) = \frac{n\,(n^3 + 1)}{2}$.

Untuk kubus pada persoalan *magic cube*, yang memiliki panjang sisi 5 satuan, dapat diketahui besar nilai *magic number*-nya adalah sebagai berikut.

$$M(n) = \frac{n\,(n^3 + 1)}{2}$$

$$M(5) = \frac{5\,(5^3 + 1)}{2}$$

$$M(5) = \frac{5\,(126)}{2}$$

$$M(5) = \frac{630}{2}$$

$$M(5) = 315$$

Dari perhitungan yang sudah dilakukan di atas, kita dapat mengetahui bahwa besar nilai *magic number* untuk kubus yang memiliki panjang sisi 5 satuan adalah 315, yang artinya jumlah angka untuk setiap baris, kolom, tiang, diagonal bidang, atau diagonal ruang adalah 315.

Deviasi merupakan selisih suatu nilai terhadap nilai harapan atau nilai ideal. Pada masalah ini, deviasi yang dimaksud adalah besarnya selisih antara jumlah angka untuk setiap baris, kolom, tiang, diagonal bidang, ataupun diagonal ruang dengan *magic number*.

- Deviasi baris merupakan selisih antara jumlah angka pada baris kubus dengan *magic number*.

$$D_{baris} = |S_{baris} - 315|$$

- Deviasi kolom merupakan selisih antara jumlah angka pada kolom kubus dengan *magic number*.

$$D_{kolom} = |S_{kolom} - 315|$$

- Deviasi kolom merupakan selisih antara jumlah angka pada kolom kubus dengan *magic number*.

$$D_{tiang} = |S_{tiang} - 315|$$

- Deviasi kolom merupakan selisih antara jumlah angka pada kolom kubus dengan *magic number*.

$$D_{diagonalB} = |S_{diagonalB} - 315|$$

- Deviasi kolom merupakan selisih antara jumlah angka pada kolom kubus dengan *magic number*.

$$D_{diagonalR} = |S_{diagonalR} - 315|$$

Objective function dapat ditentukan dengan menjumlahkan hasil deviasi dari setiap baris, kolom, tiang, diagonal bidang, dan diagonal ruang untuk menghasilkan total deviasi secara keseluruhan. Pada sebuah kubus dengan ukuran 5x5x5, terdapat 25 baris, 25 kolom, 25 tiang, 20 diagonal bidang, dan 4 diagonal ruang, sehingga:

$$\textit{Objective Function} = \sum_{i=1}^{25} D_{baris} + \sum_{i=1}^{25} D_{kolom} + \sum_{i=1}^{25} D_{tiang} + \sum_{i=1}^{20} D_{diagonalB} + \sum_{i=1}^{4} D_{diagonalR}$$

Dari *objective* tersebut, solusi terbaik dapat dilihat dari hasil total deviasi yang paling kecil atau minimum.

Dengan keterangan:

D_{baris} = hasil deviasi atau selisih baris dan *magic number*

D_{kolom} = hasil deviasi atau selisih kolom dan *magic number*

D_{tiang} = hasil deviasi atau selisih tiang dan *magic number*

$D_{diagonalB}$ = hasil deviasi atau selisih diagonal bidang dan *magic number*

$D_{diagonalR}$ = hasil deviasi atau selisih diagonal ruang dan *magic number*

S_{baris} = jumlah angka pada suatu baris

S_{kolom} = jumlah angka pada suatu kolom

S_{tiang} = jumlah angka pada suatu tiang

$S_{diagonalB}$ = jumlah angka pada suatu diagonal bidang

$S_{diagonalR}$ = jumlah angka pada suatu diagonal ruang

315 = *magic number*

B. Penjelasan Implementasi Algoritma *Local Search*

- ***Steepest Ascent Hill-Climbing***

➔ **Definisi**

Steepest Ascent Hill-climbing merupakan salah satu teknik atau algoritma pencarian pada *local search* yang bermula dari dihasilkannya sebuah solusi acak dan melakukan *generate* serta menghitung nilai dari semua *neighbor state* yang mungkin dari suatu *state* awal. Jika nilai *neighbor state* > nilai *state* saat ini dan *neighbor state* lain, maka *neighbor state* tersebut akan dipilih menjadi *state* berikutnya. Dan, setiap kali *neighbor state* ada yang memiliki nilai lebih besar daripada nilai *state* saat ini, proses pencarian akan stop sehingga solusi yang optimal mungkin bisa saja tidak ditemukan dengan algoritma *Steepest Ascent Hill-Climbing*. *Steepest Ascent Hill-Climbing* digunakan untuk mencari solusi paling optimal pada suatu permasalahan tertentu.

➔ **Inisiasi Solusi Awal**

Pada *Steepest Ascent Hill-climbing*, inisiasi solusi awal akan dilakukan secara *random*. Hasil inisiasi tersebut kemudian disimpan sebagai *current*. Initial state atau solusi awal pada permasalahan diagonal *magic cube* kubus berukuran 5x5x5 adalah angka 1 sampai 5^3 yang posisi setiap angkanya masih acak.

◆ **Mencari *Successor***

Setelah dipilih solusi awal secara *random*, algoritma akan melakukan pencarian *successor* dari solusi yang dipilih sebelumnya tersebut. *Successor* merupakan hasil *current state* yang diberi perubahan berupa penukaran posisi pada 2 angka yang menyusun kubus. Tujuannya

adalah mencari nilai dengan *objective function* yang mendekati atau sama dengan 0 yang artinya deviasi dengan *magic number*-nya juga 0 atau merupakan solusi yang paling optimal.

➔ **Evaluasi *Successor***

Jika *successor* sudah dihasilkan, *successor* tersebut akan dievaluasi untuk membandingkan nilai *successor* dengan nilai *current*. Jika nilai *successor* lebih baik dibandingkan dengan nilai *current*, maka nilai current akan digantikan dengan nilai *successor*. Namun, jika nilai current lebih baik dibandingkan dengan nilai *successor*, nilai *current* akan tetap dan tidak diubah.

➔ **Iterasi dan Terminasi**

Proses atau langkah-langkah pencarian dari pemilihan *successor* hingga evaluasi *successor* akan dilakukan secara berulang hingga ditemukan solusi yang paling optimal. Solusi optimal yang dimaksud adalah solusi dimana tidak ada lagi *successor* yang lebih baik dari *current state*. Proses akan dihentikan ketika sudah ditemukan solusi terbaik tersebut sehingga current state dianggap sebagai solusi terbaik dan proses akan selesai.

➔ **Deskripsi Fungsi**

Fungsi utama dari *Steepest Ascent Hill-climbing* adalah **steepest_ascent_hill_climbing** dengan parameter berupa max_iterations dan num_neighbors. Parameter max_iterations merupakan jumlah maksimum iterasi yang akan dijalankan untuk membatasi berapa kali algoritma dapat mencoba menemukan solusi yang lebih baik, di mana dalam hal ini dibatasi sebanyak 200. Sementara itu, parameter num_neighbors merupakan jumlah tetangga yang akan diperoleh dan dievaluasi di setiap iterasi untuk menentukan seberapa banyak solusi alternatif yang akan dievaluasi di sekitar *current_state*, di mana dalam hal ini diisi dengan 100.

Fungsi steepest_ascent_hill_climbing dimulai dengan menginisialisasi keadaan awal di mana variabel initial_state dihasilkan

secara acak dengan memanfaatkan pemanggilan fungsi generate_random_state(). Variabel current_state untuk menyimpan solusi yang sedang dievaluasi oleh algoritma saat ini. Variabel current_value sebagai nilai dari *objective function* yang dievaluasi untuk current_state yang menyimpan hasil evaluasi atau nilai *objective function* dari solusi tersebut.

➔ ***Source Code***

Berikut ini merupakan *source code* dari fungsi steepest_ascent_hill_climbing.

```
def steepest_ascent_hill_climbing(max_iterations=200,
num_neighbors=100):
    # Store the initial state at the start of the algorithm
    initial_state = generate_random_state()
    current_state = initial_state
    current_value = objective_function(current_state)

    start_time = time.time()
    value_history = [current_value]  # Untuk melacak value di
setiap iterasi

    for iteration in range(max_iterations):
        best_neighbor = None
        best_neighbor_value = -np.inf

        # Search for the best neighbors
        for _ in range(num_neighbors):
            neighbor = generate_neighbor(current_state)
            neighbor_value = objective_function(neighbor)

            # Keep track of the best neighbor
            if neighbor_value > best_neighbor_value:
                best_neighbor, best_neighbor_value = neighbor,
neighbor_value

        # If the best neighbor is better, move to it
```

```
        if best_neighbor_value > current_value:
            current_state, current_value = best_neighbor,
best_neighbor_value

        # If no better neighbor, stop (local maximum reached)
        else:
            print("Local maximum reached.")
            break

        value_history.append(current_value)

    end_time = time.time()
    duration = end_time - start_time
    print(f"Steepest Ascent Hill Climbing finished after
{iteration + 1} iterations")
    return initial_state, current_state, current_value,
value_history, duration
```

- ***Hill-Climbing with Sideways Move***

  ➔ **Definisi**

  Hill-Climbing with Sideways Move merupakan salah satu teknik atau algoritma pencarian pada *local search* yang digunakan untuk mencari solusi paling optimal pada suatu permasalahan tertentu. *Hill-Climbing with Sideways Move* memungkinkan pergerakan tetap dilakukan pada fase *flat* karena dapat bergerak sideways ketika nilai *successor* sama dengan nilai *current*. Hal tersebut membantu pencarian solusi ketika terjebak pada bagian *flat*. Definisi dari *Hill-Climbing with Sideways Move* sebenarnya hampir serupa dengan definisi dari algoritma *Steepest Ascent Hill-Climbing*, yang membedakannya hanyalah algoritma tetap terus mencari jika nilai *neighbor* tertinggi sama dengan nilai terkini. Algoritma baru stop mencari jika nilai yang paling tinggi di antara *neighbor state* < nilai *state* terkini. Maka, solusi optimal mungkin tidak ditemukan dengan algoritma *Hill-Climbing with Sideways Move*.

➔ **Inisiasi solusi awal**

Pada *Hill-Climbing with Sideways Move*, inisiasi solusi awal akan dilakukan secara *random*. Hasil inisiasi tersebut kemudian disimpan sebagai *current state*. *Initial state* atau solusi awal pada permasalahan diagonal *magic cube* berukuran 5x5x5 adalah angka 1 sampai 5^3 yang posisi setiap angkanya masih acak.

➔ **Mencari *Successor***

Setelah solusi/*state* awal sudah ditentukan, algoritma akan melakukan pencarian *successor* dari solusi yang dipilih sebelumnya tersebut. *Successor* merupakan hasil *current state* yang diberi perubahan berupa penukaran posisi pada 2 angka yang menyusun kubus. Tujuannya adalah mencari nilai dengan *objective function* yang mendekati atau sama dengan 0 yang artinya deviasi dengan *magic number*-nya juga 0 atau merupakan solusi yang paling optimal.

➔ **Evaluasi *Successor***

Jika *successor* sudah dihasilkan, *successor* tersebut akan dievaluasi untuk mengetahui nilai *successor* tersebut. Kemudian, akan dipilih *successor* dengan nilai terbaik. Jika *successor* yang paling baik nilainya tersebut ternyata memiliki nilai yang sama dengan *current state*, maka algoritma akan tetap terus bergerak dan melakukan *sideways move*. Proses ini dilakukan dengan harapan bisa mendapatkan nilai *successor* yang lebih baik dan juga dapat keluar dari bagian *flat*.

➔ **Iterasi dan Terminasi**

Proses atau langkah-langkah pencarian *successor* dan evaluasi *successor* akan dilakukan terus secara berulang selama *successor* yang lebih baik ditemukan atau *sideways move* masih memungkinkan. Proses ini akan berhenti jika sudah mencapai solusi *local maximum* atau ketika *sideways move* telah mencapai batas langkahnya. Jika sudah selesai, maka *current state* dianggap sebagai solusi terbaik lalu proses akan selesai.

➔ **Deskripsi Fungsi**

Fungsi utama dari *Hill-Climbing with Sideways Move* adalah **hill_climbing_with_sideways_move()** yang menjabarkan cara kerja algoritma Hill Climbing with Sideways Move. Parameter yang digunakan pada fungsi ini adalah max_iterations = 10000 (algoritma dapat mengulang sampai maksimal 10000 kali) dan max_sideways =10 (algoritma mempunyai batas untuk melakukan gerakan sideways sebesar 10 kali). Pertama-tama, fungsi mulai dengan menghasilkan current_state dari *cube* secara acak menggunakan pemanggilan fungsi generate_random_state(). current_state berbentuk *cube* ini lalu disimpan sebagai initial_state. Lalu, nilai current_value juga ditentukan dengan menghitung *objective function* dari current_state saat ini. Sebelum memulai iterasi, variabel sideways_move (jumlah gerakan sideways yang sudah dilakukan pada setiap iterasi) dan iteration (jumlah iterasi yang telah dilewati) diinisialisasikan menjadi 0. Terdapat variabel start_time untuk mencatat waktu saat iterasi mulai dijalankan dan value_history untuk melacak value di setiap iterasi (saat ini diinisialisasi dengan current value).

Kemudian, fungsi masuk ke bagian *loop* yang akan berjalan hingga mencapai max_iterations, kecuali memenuhi salah satu kondisi dalam *loop* yang menyebabkan break (berhenti iterasi dan keluar dari *loop*). Di setiap iterasi, kondisi cube tetangga (neighbor) dihasilkan dengan fungsi generate_neighbor yang memakai parameter current_state saat iterasi sedang berlangsung. *Cube neighbor* juga dihitung value-nya dengan fungsi objective_function dan disimpan pada variabel neighbor_value. Setelah mengetahui neighbor state, fungsi akan membandingkan neighbor_value dengan current_value. Jika neighbor_value > current_value, maka neighbor state akan menjadi current_state yang baru dan current_value juga ikut diganti. *Counter* untuk sideways_move ditetapkan sebagai 0 karena tidak perlu dilakukan sideways_move pada iterasi kali ini. Jika neighbor_value = current_value, maka akan dilakukan sideways move (bergerak ke neighbor state dengan value yang sama) yang menyebabkan counter sideways_move bertambah 1, serta current_state

dan value_state yang juga ikut diubah. Sideways move hanya dapat dilakukan jika counter sideways_moves masih belum mencapai max_sideways. Jika sudah mencapai batas max_sideways, loop akan berhenti. Namun, jika neighbor value < current value, tidak dilakukan apa-apa dan lanjut ke langkah selanjutnya, yaitu menambahkan current_value yang didapat ke value_history lalu menambah 1 pada jumlah iterasi. Setelah keluar dari loop, waktu iterasi selesai dicatat pada variabel end_time. Durasi ditentukan dari selisih end_time dengan start_time. Terakhir, dicetak kalimat yang menunjukkan bahwa algoritma *Hill Climbing with Sideways Move* selesai setelah iterasi sebanyak berapa kali dari data *counter iterations* yang kita punya. Fungsi ini mengembalikan initial_state, current_state, current_value, value_history, dan duration.

➔ ***Source Code***

Berikut ini merupakan *source code* dari fungsi hill_climbing_with_sideways_move.

```
def hill_climbing_with_sideways_move(max_iterations=10000,
max_sideways=10):
    current_state = generate_random_state()
    initial_state = current_state
    current_value = objective_function(current_state)

    sideways_moves = 0  # Menghitung jumlah sideways move
    iteration = 0

    start_time = time.time()
    value_history = [current_value]  # Untuk melacak value di
setiap iterasi

    while iteration < max_iterations:
        neighbor = generate_neighbor(current_state)
        neighbor_value = objective_function(neighbor)
```

```
        # Jika neighbor lebih baik
        if neighbor_value > current_value:
            current_state = neighbor
            current_value = neighbor_value
            sideways_moves = 0  # Reset sideway move
        elif neighbor_value == current_value:
            # Jika value sama, terima sebagai sideways move
            if sideways_moves <= max_sideways:
                current_state = neighbor
                current_value = neighbor_value
                sideways_moves += 1  # Tambahkan sideways move
            else:
                # Jika mencapai batas max sideways move, berhenti
                break
        else:
            # Jika neighbor lebih buruk, abaikan
            pass

        value_history.append(current_value)  # Lacak value di
setiap iterasi
        iteration += 1

    end_time = time.time()
    duration = end_time - start_time

    print(f"Hill Climbing with Sideways Move finished after
{iteration} iterations")

    return initial_state, current_state, current_value,
value_history, duration
```

- ***Random Restart Hill-Climbing***

  ➔ **Definisi**

  Random Restart Hill-Climbing merupakan salah satu teknik atau algoritma *hill-climbing* yang melakukan pencarian solusi secara acak. Hal

tersebut membantu pencarian solusi ketika terjebak dalam *local maximum* atau solusi optimum lokal karena *Random Restart Hill-climbing* dapat melakukan *random restart* yang meningkatkan peluang menemukan solusi yang mendekati atau merupakan *global maximum* atau solusi terbaik suatu permasalahan. *Random Restart Hill-Climbing* adalah modifikasi dari algoritma *Steepest Ascent Hill-Climbing* yang digunakan beberapa kali sampai memperoleh solusi terbaik. *Random Restart Hill-Climbing* ini akan terus berlanjut apabila proses pencarian ada di plateau atau yang kita kenal dengan maksimum lokal. Sebaliknya, *state* yang ada di plateau atau yang kita tahu sebagai maksimum lokal ini akan tergantikan dengan *state* baru secara *random* untuk melanjutkan pencarian dan akan stop sampai fungsi yang kita inginkan berhasil tercapai. Tetapi, *Random Restart Hill-Climbing* membutuhkan waktu yang lebih lama walau mampu menemukan cara terbaik untuk menyelesaikan persoalan.

➔ **Inisiasi Solusi Awal**

Pada *Random Restart Hill-climbing*, inisiasi solusi awal akan dilakukan secara *random*. Hasil inisiasi tersebut kemudian disimpan sebagai *current state*. *Initial state* atau solusi awal pada permasalahan diagonal magic cube kubus berukuran 5x5x5 adalah angka 1 sampai 5^3 yang posisi setiap angkanya masih acak.

➔ **Mencari *Successor***

Setelah solusi/*state* awal sudah ditentukan, algoritma akan melakukan pencarian *successor* dari solusi yang dipilih sebelumnya tersebut. Pemilihan *successor* pada *Random Restart Hill-Climbing* dilakukan secara *random*. *Successor* merupakan hasil *current state* yang diberi perubahan berupa penukaran posisi pada 2 angka yang menyusun kubus. Tujuannya adalah mencari nilai dengan *objective function* yang mendekati atau sama dengan 0 yang artinya deviasi dengan *magic number*-nya juga 0 atau merupakan solusi yang paling optimal.

➔ **Evaluasi *Successor***

Jika *successor* sudah dihasilkan, *successor* tersebut akan dievaluasi untuk membandingkan nilai *successor* dengan nilai *current*. Jika nilai *successor* lebih baik dibandingkan dengan nilai *current*, maka nilai *current* akan digantikan dengan nilai *successor*. Namun, jika nilai *current* lebih baik dibandingkan dengan nilai *successor*, nilai *current* akan tetap dan tidak diubah.

➔ **Iterasi dan *Restart***

Proses atau langkah-langkah *restart* atau pemilihan *successor* hingga evaluasi *successor* akan dilakukan berulang secara random selama masih ada *successor* yang memiliki nilai yang lebih baik daripada *current*. *Random restart* memungkinkan kita dapat mendapatkan solusi paling optimal secara global dan tidak hanya lokal.

➔ **Deskripsi Fungsi**

Fungsi utama dari *Random Restart Hill-Climbing* adalah **random_restart_hill climbing** menggunakan parameter max_restart = 7 (algoritma mempunyai batas untuk melakukan restart sebesar 7 kali), max_iterations = 200 (algoritma dapat mengulang sampai maksimal 200 kali), dan num_neighbors = 100 (jumlah neighbor yang akan dihasilkan dan dibandingkan dalam setiap iterasi berjumlah 100). Pertama-tama, dilakukan proses inisialisasi pada variabel-variabel yang akan dipakai. Variabel best_state (menyimpan state terbaik) diinisialisasikan menjadi None, variabel best_value (menyimpan value dari best_state) diinisialisasikan menjadi nilai negatif tak terhingga, variabel overall_value_history (menyimpan value_history dari setiap iterasi pada semua restart) diinisialisasikan menjadi list kosong, dan total_iterations (menyimpan jumlah iterasi yang telah dilakukan algoritma) diinisialisasikan menjadi 0. Terdapat variabel start_time untuk mencatat waktu saat iterasi mulai dijalankan.

Kemudian, fungsi masuk ke bagian loop yang akan melakukan restart hingga mencapai max_restarts. Di setiap restart, akan dilakukan algoritma Steepest Ascent Hill-Climbing dengan memanggil fungsi

steepest_ascent_hill_climbing yang menggunakan parameter max_iterations dan num_neighbors. Hasil dari fungsi steepest_ascent_hill_climbing akan disimpan dalam variabel initial_state, current_state, current_value, value_history, dan restart_duration. Setelah itu, overall_value_history diperbaharui dengan menambahkan semua nilai value_history dari setiap iterasi. Variabel total_iterations juga diperbaharui dengan menambahkan jumlah iterasi dari value_history pada restart saat ini. Kemudian current_value dibandingkan dengan best_value. Awalnya current_value ditetapkan sebagai nilai terakhir pada value_history. Jika current_value > best_value, maka best_value dan best_state akan diubah dengan nilai dan state dari iterasi saat ini.

Setelah loop selesai dan seluruh restart telah dilakukan, waktu selesai dicatat pada variabel end_time. Durasi ditentukan dari selisih end_time dengan start_time. Terakhir, dicetak kalimat yang menunjukkan jumlah restart dan total iterasi yang dilakukan selama semua restart. Fungsi ini mengembalikan initial_state, current_state, best_value, overall_value_history, dan duration.

➔ ***Source Code***

Berikut ini merupakan *source code* dari fungsi random_restart_hill_climbing.

```
def random_restart_hill_climbing(max_restarts=7,
max_iterations=200, num_neighbors=100):
    best_state = None
    best_value = -np.inf
    overall_value_history = []
    total_iterations = 0

    start_time = time.time()

    for restart in range(max_restarts):
        print(f"\nRestart {restart + 1}/{max_restarts}")
        initial_state, current_state, current_value,
value_history, restart_duration =
```

```
steepest_ascent_hill_climbing(max_iterations, num_neighbors)

        # Track value history and total iterations
        overall_value_history.extend(value_history)
        total_iterations += len(value_history)

        # Check if the result from this run is the best
        current_value = value_history[-1]
        if current_value > best_value:
            best_value = current_value
            best_state = current_state

        print(f"Iterations in this restart:
{len(value_history)}")

    end_time = time.time()
    duration = end_time - start_time

    print(f"\nTotal restarts: {restart + 1}")
    print(f"Total iterations across all restarts:
{total_iterations}")

    return initial_state, current_state, best_value,
overall_value_history, duration
```

- ***Stochastic Hill-Climbing***

  ➔ **Definisi**

  Stochastic Hill-climbing merupakan salah satu teknik atau algoritma pencarian pada *local search* yang merupakan variasi dari algoritma *hill-climbing* dan digunakan untuk mencari solusi paling optimal pada suatu permasalahan tertentu. *Stochastic Hill-Climbing* akan memilih *successor* secara *random* dan tidak selalu memilih *successor* terbaik. Hal tersebut membuat algoritma *Stochastic Hill-climbing* ini memungkinkan untuk melakukan penjelajahan yang lebih luas walaupun akan berdampak

pada lambatnya proses karena langkah-langkah yang dilakukan akan lebih banyak pula.

➔ **Inisiasi Solusi Awal**

Pada *Stochastic Hill-Climbing,* inisiasi solusi awal akan dilakukan secara *random*. Hasil inisiasi tersebut kemudian disimpan sebagai *current state*. *Initial state* atau solusi awal pada permasalahan diagonal *magic cube* kubus berukuran 5x5x5 adalah angka 1 sampai 5^3 yang posisi setiap angkanya masih acak.

➔ **Mencari *Successor***

Setelah dipilih solusi awal secara *random*, algoritma akan melakukan pencarian *successor* dari solusi yang dipilih sebelumnya tersebut. Pemilihan *successor* akan dilakukan dengan memilih satu *successor* secara *random* dan tidak memeriksa atau mencari dari semua *successor* yang ada. *Successor* merupakan hasil *current state* yang diberi perubahan berupa penukaran posisi pada 2 angka yang menyusun kubus. Tujuannya adalah mencari nilai dengan *objective function* yang mendekati atau sama dengan 0 yang artinya deviasi dengan *magic number*-nya juga 0 atau merupakan solusi yang paling optimal.

➔ **Evaluasi *Successor***

Jika *successor* sudah dihasilkan, nilai *successor* tersebut akan dievaluasi untuk membandingkan nilai *successor* dengan nilai current. Jika nilai *successor* lebih baik dibandingkan dengan nilai *current*, maka nilai *current* akan digantikan dengan nilai *successor*. Namun, jika nilai *current* lebih baik dibandingkan dengan nilai *successor*, nilai *current* akan tetap dan tidak diubah.

➔ **Iterasi dan Terminasi**

Proses atau langkah-langkah pencarian dan evaluasi *successor* akan dilakukan secara berulang hingga tercapai iterasi maksimal. Hal tersebut memungkinkan proses berhenti sebelum ditemukan solusi yang paling optimal karena algoritma akan berhenti setelah iterasi maksimal tercapai walaupun nilai *current* belum mencapai yang terbaik.

Berdasarkan hal tersebut, terminasi akan dilakukan jika salah satu dari dua kondisi tercapai. Kondisi pertama adalah jika iterasi maksimum sudah tercapai. Kondisi kedua adalah tidak ada lagi *successor* dengan nilai lebih baik dari nilai *current*.

➔ **Deskripsi Fungsi**

Fungsi utama dari *Stochastic Hill Climbing* adalah **stochastic_hill_climbing** yang menggunakan parameter max_iterations = 100000 (algoritma dapat mengulang sampai maksimal 100000 kali). Pertama-tama, fungsi dimulai dengan menghasilkan initial_state secara acak menggunakan fungsi generate_random_state(). initial_state berbentuk cube ini lalu disimpan sebagai current_state. Lalu, nilai current_value juga ditentukan dengan menghitung objective function dari current_state saat ini. Terdapat variabel start_time untuk mencatat waktu saat iterasi mulai dijalankan dan value_history untuk melacak value di setiap iterasi (saat ini diinisialisasi dengan current value).

Kemudian, fungsi masuk ke bagian loop yang akan melakukan iteration hingga mencapai max_iterations. Di setiap iterasi, kondisi cube tetangga (neighbor) dihasilkan dengan fungsi generate_neighbor yang memakai parameter current_state saat iterasi sedang berlangsung. Cube neighbor juga dihitung value-nya dengan fungsi objective_function dan disimpan pada variabel neighbor_value. Setelah mengetahui neighbor state, fungsi akan membandingkan neighbor_value dengan current_value. Jika neighbor_value > current_value, maka neighbor state akan menjadi current_state yang baru dan current_value juga ikut diganti dengan neighbor_value. Lalu, current_value yang didapat ditambahkan ke value_history. Setelah keluar dari loop, waktu iterasi selesai dicatat pada variabel end_time. Durasi ditentukan dari selisih end_time dengan start_time. Terakhir, dicetak kalimat yang menunjukkan bahwa algoritma Stochastic Hill Climbing selesai setelah iterasi sebanyak berapa kali. Fungsi ini mengembalikan initial_state, current_state, current_value, value_history, dan duration.

➔ ***Source Code***

Berikut ini merupakan *source code* dari fungsi stochastic_hill_climbing.

```
def stochastic_hill_climbing(max_iterations=100000):

    initial_state = generate_random_state()
    current_state = initial_state
    current_value = objective_function(current_state)

    start_time = time.time()
    value_history = [current_value]

    for iteration in range(max_iterations):
        neighbor = generate_neighbor(current_state)  # Choose a
random neighbor
        neighbor_value = objective_function(neighbor)

        # If neighbor is better, move to the neighbor
        if neighbor_value > current_value:
            current_state, current_value = neighbor,
neighbor_value

        value_history.append(current_value)

    end_time = time.time()
    duration = end_time - start_time

    print(f"Stochastic Hill Climbing finished after {iteration +
1} iterations")
    return initial_state, current_state, current_value,
value_history, duration
```

- ***Simulated Annealing***

➔ **Definisi**

Simulated Annealing merupakan salah satu teknik atau algoritma pada *local search* yang meniru konsep pelunakkan atau pengerasan logam dan kaca pada metalurgi yang menggunakan suhu tinggi dan mendinginkannya secara bertahap. *Simulated Annealing* mengombinasikan algoritma *Hill-Climbing* dengan *Random-Walk* yang mana merupakan versi dari *Stochastic Hill-Climbing* di mana diizinkan pemindahan beberapa *downhill*. Bertujuan untuk mencari solusi yang lebih baik tetapi masih mungkin bagi kita untuk berpindah ke solusi yang lebih buruk yang mana solusi tersebut (utamanya di tahap awal) mempunyai probabilitas tertentu.

➔ **Inisiasi Solusi Awal**

Kubus diinisialisasi dengan solusi awal yang terdiri atas angka random. Pada tugas ini, kami menginisialisasi temperatur dengan nilai 1000 dan *cooling rate* sebesar 0,003.

➔ **Evaluasi *Successor***

Evaluasi dihitung menggunakan *objective function* yang memberikan total deviasi yang merupakan selisih sum tiap baris, kolom, tiang, dan diagonal dari *magic number*. Semakin kecil total deviasi, maka semakin tinggi nilai dari *successor*.

➔ **Mencari *Successor***

Dua angka dalam kubus ditukar secara acak, dan evaluasi dihitung ulang. Jika deviasi lebih rendah dibanding *current evaluation* atau solusi lebih baik, maka solusi diterima. Jika deviasi lebih tinggi dibanding *current evaluation* atau solusi lebih buruk, maka solusi tetap bisa diterima dengan probabilitas $e^{-\Delta E/T}$.

➔ **Iterasi dan Terminasi**

Temperatur diturunkan perlahan pada setiap iterasi, sehingga semakin lama semakin sulit menerima solusi yang lebih buruk. Rumus penurunan temperatur adalah $T = T\times(1 - cooling\,rate)$. Pencarian akan berhenti saat temperatur kurang dari 1.

➔ **Deskripsi Fungsi**

Fungsi utama dari *Simulated Annealing* adalah **simulated_annealing** yang menggunakan parameter max_iterations = 100000 (algoritma dapat mengulang sampai maksimal 100000 kali), initial_temperature = 1000 (temperatur awal), dan cooling_rate = 0,999 (tingkat penurunan temperatur di setiap iterasi). Nilai cooling_rate yang mendekati 1 berarti temperatur akan menurun secara perlahan dan durasi algoritmanya tidak cepat berhenti. Pertama-tama, fungsi dimulai dengan menghasilkan initial_state secara acak menggunakan fungsi generate_random_state(). initial_state berbentuk cube ini lalu disimpan sebagai current_state. Lalu, nilai current_value juga ditentukan dengan menghitung objective function dari current_state saat ini. Variabel temperature diinisialisasikan dengan nilai initial_temperature, yang seiring berjalannya algoritma akan terus menurun sesuai cooling_rate. Lalu, variabel probability_history (untuk menyimpan riwayat probabilitas penerimaan state yang lebih buruk) diinisialisasikan sebagai list kosong dan variabel stuck_count (menyimpan berapa kali algoritma gagal menemukan kemajuan atau dalam kata lain berapa kali algoritma terjebak di local optima) diatur ke angka 0. Terdapat variabel start_time untuk mencatat waktu saat iterasi mulai dijalankan dan value_history untuk melacak value di setiap iterasi (saat ini diinisialisasi dengan current value). Kemudian, fungsi masuk ke bagian loop yang akan melakukan iteration hingga mencapai max_iterations. Di setiap iterasi, kondisi cube tetangga (neighbor) dihasilkan dengan fungsi generate_neighbor yang memakai parameter current_state saat iterasi sedang berlangsung. Cube neighbor juga dihitung value-nya dengan fungsi objective_function dan disimpan pada variabel neighbor_value. Kemudian selisih antara neighbor_value dan current_value disimpan dalam variabel delta_e.

Setelah itu, kita akan menelaah nilai delta_e. Jika nilai delta_e > 0 (positif), artinya neighbor state lebih baik daripada state saat ini. Maka neighbor state akan menjadi current_state yang baru dan current_value juga ikut diganti dengan neighbor_value. Sementara itu, jika nilai delta_e

<= 0 (negatif atau sama dengan 0), maka kita fungsi akan menghitung probabilitas penerimaan neighbor state dengan rumus e^(delta_e / temperature). Hasil probabilitas ini ditambahkan ke list probability_history. Untuk menentukan apakah neighbor diterima atau tidak berdasarkan probabilitas, fungsi akan membandingkan nilai acak yang dihasilkan np.random.random() dengan probabilitas yang sudah dihitung tadi. Jika nilai acak < probabilitas, neighbor tetap diterima meskipun lebih buruk. Masing-masing dari neighbor_state dan neighbor_value disimpan sebagai current_state dan current_value. Jika nilai acak >= probabilitas, neighbor tidak akan diterima dan stuck_count bertambah 1 sebagai tanda tidak adanya kemajuan. Selesai menentukan penerimaan neighbor, current_value yang didapat ditambahkan ke value_history dan nilai temperature diperbarui dengan hasil perkaliannya dengan cooling_rate. Jika temperature mencapai 0, loop dihentikan lebih awal, dan pesan penyelesaian algoritma akan dicetak. Setelah keluar dari loop, waktu iterasi selesai dicatat pada variabel end_time. Durasi ditentukan dari selisih end_time dengan start_time. Kemudian dicetak kalimat yang menunjukkan total iterasi yang dijalankan algoritma ini dan jumlah stuck_count yang didapat. Terakhir, dibuat plotting penerimaan state yang lebih buruk (e^(ΔE / T)) terhadap iterations dalam bentuk grafik. Fungsi ini mengembalikan initial_state, current_state, current_value, value_history, dan duration.

```
# ================================ Simulated Annealing ================================
def simulated_annealing(max_iterations=100000, initial_temperature=1000, cooling_rate=0.999):
    initial_state = generate_random_state()
    current_state = initial_state
    current_value = objective_function(current_state)
    temperature = initial_temperature
    value_history = [current_value]
    probability_history = []
    stuck_count = 0

    start_time = time.time()

    for iteration in range(max_iterations):
        neighbor = generate_neighbor(current_state)
        neighbor_value = objective_function(neighbor)
        delta_e = neighbor_value - current_value

        if delta_e > 0:
            # Accept better neighbor
            current_state, current_value = neighbor, neighbor_value
        else:
            # Calculate acceptance probability
            probability = np.exp(delta_e / temperature)
            probability_history.append(probability)

            if np.random.random() < probability:
                # Accept worse neighbor based on probability
                current_state, current_value = neighbor, neighbor_value
            else:
                # Increment stuck count if no improvement
                stuck_count += 1

        value_history.append(current_value)
        temperature *= cooling_rate

        if temperature == 0:  # Menghentikan jika suhu terlalu rendah
            print(f"Simulated Annealing Finished early due to low temperature at iteration {iteration}")
            break

    end_time = time.time()
    duration = end_time - start_time

    print(f"Total Iterations: {iteration + 1}")
    print(f"Stuck Count (Local Optima): {stuck_count}")

    # Plotting e^(Delta E / T) over Iterations
    plt.plot(probability_history, label="Acceptance Probability (e^(ΔE / T))")
    plt.xlabel("Iterations")
    plt.ylabel("Acceptance Probability")
    plt.title("Simulated Annealing: e^(ΔE / T) over Iterations")
    plt.legend()
    plt.show()

    return initial_state, current_state, current_value, value_history, duration
```

➔ ***Source Code***

Berikut ini merupakan *source code* dari fungsi simulated_annealing.

```
def simulated_annealing(max_iterations=100000,
initial_temperature=1000, cooling_rate=0.999):
    initial_state = generate_random_state()
    current_state = initial_state
    current_value = objective_function(current_state)
    temperature = initial_temperature
    value_history = [current_value]
    probability_history = []
    stuck_count = 0

    start_time = time.time()

    for iteration in range(max_iterations):
        neighbor = generate_neighbor(current_state)
        neighbor_value = objective_function(neighbor)
        delta_e = neighbor_value - current_value

        if delta_e > 0:
            # Accept better neighbor
```

```
            current_state, current_value = neighbor,
neighbor_value
        else:
            # Calculate acceptance probability
            probability = np.exp(delta_e / temperature)
            probability_history.append(probability)

            if np.random.random() < probability:
                # Accept worse neighbor based on probability
                current_state, current_value = neighbor,
neighbor_value
            else:
                # Increment stuck count if no improvement
                stuck_count += 1

        value_history.append(current_value)
        temperature *= cooling_rate

        if temperature == 0:  # Stop if temp gets too low
            print(f"Simulated Annealing Finished early due to low
temperature at iteration {iteration}")
            break

    end_time = time.time()
    duration = end_time - start_time

    print(f"Total Iterations: {iteration + 1}")
    print(f"Stuck Count (Local Optima): {stuck_count}")

    # Plotting e^(Delta E / T) over Iterations
    plt.plot(probability_history, label="Acceptance Probability
(e^(ΔE / T))")
    plt.xlabel("Iterations")
    plt.ylabel("Acceptance Probability")
    plt.title("Simulated Annealing: e^(ΔE / T) over
Iterations")
    plt.legend()
```

```
    plt.show()

    return initial_state, current_state, current_value,
value_history, duration
```

- ***Genetic Algorithm***

  ➔ **Definisi**

  Genetic Algorithm merupakan salah satu teknik atau algoritma pencarian pada *local search* yang meniru konsep evolusi. Definisi dari *Genetic Algorithm* sebenarnya dimulai dari menginisialisasi populasi awal yang terdiri atas berbagai kemungkinan solusi yang ditandai sebagai string. Setiap anggota populasi memiliki nilai *fitness*-nya masing-masing, yang menunjukkan takaran seefektif apa solusi yang diberikan. Selama proses seleksi, individu yang mempunyai nilai *fitness* lebih tinggi kemungkinan lebih besar terpilih sebagai *parents*. Para *parents* ini menghasilkan keturunan baru yang menggabungkan gen mereka melalui proses *crossover*. Di sisi lain, populasi memperoleh variasi baru melalui mutasi. Sampai solusi terbaik ditemukan, proses ini diulang sehingga kualitas solusi biasanya akan meningkat seiring dengan jumlah generasi yang dibuat.

  ➔ **Inisialisasi Solusi Awal**

  Algoritma dimulai dengan sejumlah populasi awal yang kami tetapkan sebanyak 10 *state*, di mana masing-masing individu adalah kubus yang sudah diacak.

  ➔ **Evaluasi**

  Dilakukan evaluasi pada tiap individu populasi dengan menghitung total deviasi dari *magic number*.

  ➔ **Seleksi**

  Individu-individu terbaik dipilih berdasarkan fungsi *fitness*. Proses seleksi ini menggunakan probabilitas yang berbanding terbalik dengan

total deviasi, sehingga individu dengan total deviasi yang lebih kecil memiliki peluang yang lebih besar untuk dipilih.

➔ ***Crossover***

Dua individu yang terseleksi dijadikan *parent* untuk melakukan *crossover*, yaitu pertukaran sebagian konfigurasi antara dua *parent* tersebut yakni berupa segmen kubus. Sehingga menghasilkan individu baru.

➔ **Mutasi**

Individu-individu baru yang dihasilkan dari *crossover* dimutasi dengan cara menukar sejumlah posisi angka yang terdapat dalam kubus. Kami memutuskan untuk melakukan mutasi 3 posisi sekaligus untuk meminimalisir kemungkinan terjebak di *local optimum,* namun tidak lebih dari itu karena mutasi yang terlalu besar akan mempersulit penyesuaian solusi di saat sudah mendekati *global optimum*.

➔ **Iterasi dan Terminasi**

Proses tersebut diiterasi sebanyak beberapa generasi hingga terdapat beberapa individu yang masuk kriteria *fitness,* di mana di antara individu *fit* tersebut akan dipilih individu terbaik untuk dikembalikan.

➔ **Deskripsi Fungsi**

Fungsi utama dari *Genetic Algorithm* adalah **genetic_algorithm** yang menggunakan parameter pop_size, max_generations, dan mutation_rate. Pertama-tama, populasi awal dihasilkan dengan memanggil fungsi create_initial_population. Untuk setiap generasi, dihitung nilai fitness terbaik dan fitness rata-rata lalu disimpan ke variabel best_fitness_history dan avg_fitness_history. Masuk ke dalam loop, setiap pasangan parent dipilih dengan memanggil fungsi select_parents. Dari pasangan ini, dua keturunan (offspring1 dan offspring2) dihasilkan melalui fungsi crossover dan mutate. Populasi diperbaharui dengan kedua keturunan yang baru dihasilkan. Setelah keluar dari loop, fungsi mencetak kalimat yang menampilkan ukuran populasi, generasi, dan durasi dari

eksekusi algoritma ini. Fungsi ini mengembalikan best_individual, best_fitness_history, avg_fitness_history.

Selain itu, pada fungsi *Genetic Algorithm* juga terdapat fungsi lain seperti:

- **create_initial_population(pop_size)**: Fungsi ini menggunakan parameter pop_size dan berguna untuk menghasilkan populasi awal dalam Genetic Algorithm dalam bentuk list. Setiap elemen dalam populasi merupakan state cube acak yang dihasilkan oleh fungsi generate_random_state(). Fungsi ini mengembalikan population.
- **select_parents dengan parameter berupa population dan fitnesses** untuk menyeleksi atau memilih dua individu dari populasi berdasarkan nilai *fitness* yang dimiliki oleh masing-masing individu tersebut yang mana nilai *fitness* ini akan dinormalisasikan supaya selalu bernilai positif. Akan dipilih *random* 2 individu ketika seluruh nilai *fitness* sama atau = 0. Apabila nilai *fitness* tidak semuanya sama atau ≠0, individu akan dipilih berdasarkan seberapa besar nilai *fitness* dibandingkan individu lain terhadap nilai *fitness* yang dimiliki masing-masing
- **crossover dengan parameter parent1, parent2:** Fungsi ini menggunakan parameter parent1 dan parent2 dan berguna untuk menghasilkan keturunan dari dua *parents* yang disimpan dalam variabel *offspring*. Setiap elemen dalam *offspring* disalin dari parent1 untuk paruh pertamanya dan dari parent2 untuk paruh keduanya. Fungsi ini mengembalikan *offspring*.
- **mutate(cube, mutation_rate=0.1):** Fungsi ini menggunakan parameter cube dan mutation_rate = 0,1 dan berguna untuk memutasi individu (cube) atau membuat

variasi pada individu. Jika nilai acak yang dihasilkan fungsi random.random() < mutation_rate (0,1), maka individu ini akan mengalami perubahan menjadi *neighbor*. Jika sebaliknya, individu tidak akan berubah. Fungsi ini mengembalikan individu dalam bentuk cube.

- **print_cube_state(cube, title="Cube State")**: Fungsi ini menggunakan parameter cube dan title = "Cube State" yang berguna untuk menampilkan visualisasi dari *state*. Visualisasi ditampilkan dengan plot dan array 2 dimensi.
- **test_generic_algorithm()**: Fungsi ini berguna untuk menjalankan pengujian Genetic Algorithm dengan parameter yang bervariasi. Pengguna diminta untuk memasukkan tiga variasi ukuran populasi (pop_sizes) dan jumlah generasi (max_generations). Lalu, dua plotting dibuat, satu untuk menampilkan riwayat fitness, dan satu lagi untuk visualisasi cube terbaik dari setiap kombinasi parameter. Untuk setiap variasi populasi dan jumlah generasi, dijalankan fungsi genetic_algorithm. Kemudian hasilnya divisualisasikan menggunakan fungsi visualize_cube_GA. Grafik untuk setiap uji coba menampilkan perkembangan fitness terbaik dan rata-rata untuk setiap variasi parameter. Terakhir, seluruh plot dirapikan dan ditampilkan ke layar.

➔ ***Source Code***

Berikut ini merupakan *source code* dari fungsi *Genetic Algorithm*.

```
# Function to create initial population
def create_initial_population(pop_size):
    population = []
    for _ in range(pop_size):
        cube = generate_random_state()
        population.append(cube)
```

```
    return population

# Function to select parents with normalization
def select_parents(population, fitnesses):
    min_fitness = min(fitnesses)
    if min_fitness < 0:
        normalized_fitnesses = [f - min_fitness + 1 for f in
fitnesses]
    else:
        normalized_fitnesses = fitnesses

    if sum(normalized_fitnesses) == 0:
        return random.sample(population, 2)

    parents = random.choices(population,
weights=normalized_fitnesses, k=2)
    return parents

# Crossover function
def crossover(parent1, parent2):
    n = parent1.shape[0]
    offspring = np.empty_like(parent1)
    for i in range(n):
        offspring[i] = parent1[i] if i < n // 2 else
parent2[i]
    return offspring

# Mutation function
def mutate(cube, mutation_rate=0.1):
    if random.random() < mutation_rate:
        return generate_neighbor(cube)
    return cube

# Function to print cube visualisation
def print_cube_state(cube, title="Cube State"):
    print(f"\n{title}")
    num_slices = cube.shape[0]
    slice_height, slice_width = cube.shape[1], cube.shape[2]
```

```
    slice_titles = [f"Slice {i+1}" for i in
range(num_slices)]
    print("                    ".join(slice_titles))

    for row in range(slice_height):
        row_data = []
        for slice_idx in range(num_slices):
            row_data.append(" ".join(f"{cube[slice_idx, row,
col]:3}" for col in range(slice_width)))
        print("     ".join(row_data))


def genetic_algorithm(pop_size, max_generations,
mutation_rate):
    population = create_initial_population(pop_size)
    start_time = time.time()

    initial_fitnesses = [objective_function(individual) for
individual in population]
    initial_best_index = np.argmax(initial_fitnesses)
    initial_best_individual = population[initial_best_index]

    best_fitness_history = []
    avg_fitness_history = []

    for generation in range(max_generations):
        fitnesses = [objective_function(individual) for
individual in population]
        best_fitness = max(fitnesses)
        avg_fitness = np.mean(fitnesses)

        best_fitness_history.append(best_fitness)
        avg_fitness_history.append(avg_fitness)

        new_population = []
        for _ in range(pop_size // 2):
            parent1, parent2 = select_parents(population,
fitnesses)
            offspring1 = crossover(parent1, parent2)
```

```
            offspring2 = crossover(parent2, parent1)

            offspring1 = mutate(offspring1, mutation_rate)
            offspring2 = mutate(offspring2, mutation_rate)

            new_population.extend([offspring1, offspring2])

        population = new_population

    duration = time.time() - start_time
    print(f"\nPop {pop_size}, {max_generations} Iterations")
    print(f"  Duration: {duration:.2f} seconds")
    print(f"  Final Objective Function Value:
{best_fitness}")

    # Final state
    best_index = np.argmax(fitnesses)
    best_individual = population[best_index]
    print_cube_state(initial_best_individual, title="Initial
State - Best Individual")
    print_cube_state(best_individual, title="Final State -
Best Individual")

    return best_individual, best_fitness_history,
avg_fitness_history

# Test function without cube visualizations, only graphs
def test_genetic_algorithm():
    mutation_rate = 0.1
    pop_sizes = []
    max_generations = []

    print("Control Parameters")

    for i in range(3):
        size = int(input(f"Input population size variation
#{i+1}: "))
        pop_sizes.append(size)
```

```
    for i in range(3):
        generation = int(input(f"Input iteration count
variation #{i+1}: "))
        max_generations.append(generation)

    fig, axes = plt.subplots(len(pop_sizes),
len(max_generations), figsize=(24, 16))

    for trial in range(3):
        print(f"\n\n+====================== GENETIC ALGORITHM
TRIAL {trial + 1} ======================+")
        for row, pop_size in enumerate(pop_sizes):
            for col, gen_count in
enumerate(max_generations):
                ax = axes[row, col]
                best_cube, best_fitness_history,
avg_fitness_history = genetic_algorithm(pop_size, gen_count,
mutation_rate)

                ax.plot(best_fitness_history, label=f"Best
Fitness T{trial + 1}")
                ax.plot(avg_fitness_history, label=f"Average
Fitness T{trial + 1}", linestyle='--')
                ax.set_title(f"Pop Size {pop_size}, Gen
{gen_count}")
                ax.set_xlabel("Iteration")
                ax.set_ylabel("Fitness")
                ax.legend()

    plt.tight_layout()
    plt.show()
```

- **Tambahan**

  Terdapat pula deskripsi dan *source code* dari Objective Function, Helper Functions, dan Run Function yakni sebagai berikut.

a. Objective Function

- objective_function(cube)

→ Deskripsi fungsi: Fungsi ini untuk mengevaluasi seberapa baik konfigurasi *Diagonal Magic Cube* memenuhi nilai *magic number* yaitu 315. Fungsi tersebut menerima input kubus 3 dimensi dengan ukuran 5×5×5 kemudian jumlah elemen pada baris, kolom, dan beberapa diagonal kubus diselisihkan terhadap nilai MAGIC_NUMBER (yang sudah didefinisikan sebesar 315), yang mana kita akan semakin dekat dengan konfigurasi yang diinginkan apabila selisih yang diperoleh semakin kecil. Setelah perbedaan total dari masing-masing jumlah dihitung dan dikumpulkan, fungsi akan menghasilkan output berupa nilai negatif dari total perbedaan tersebut untuk mengurangi perbedaan untuk mencapai nilai optimal.

→ *Source Code*:

```
MAGIC_NUMBER = 315

def objective_function(cube):
    n = cube.shape[0]
    total_difference = 0

    # Sum for each row in every 5x5 slice (plane)
    for plane in range(n):
        for row in range(n):
            row_sum = np.sum(cube[plane, row, :])
            total_difference += abs(row_sum - MAGIC_NUMBER)

    # Sum for each column in every 5x5 slice (plane)
    for plane in range(n):
        for col in range(n):
            col_sum = np.sum(cube[plane, :, col])
            total_difference += abs(col_sum - MAGIC_NUMBER)

    # Sum for each "pillar" (column through planes)
    for row in range(n):
```

```
        for col in range(n):
            pillar_sum = np.sum(cube[:, row, col])
            total_difference += abs(pillar_sum -
MAGIC_NUMBER)

    # Sum for each main diagonal in every plane
    for plane in range(n):
        diag1_sum = np.sum([cube[plane, i, i] for i in
range(n)])
        diag2_sum = np.sum([cube[plane, i, n - 1 - i] for i
in range(n)])
        total_difference += abs(diag1_sum - MAGIC_NUMBER)
        total_difference += abs(diag2_sum - MAGIC_NUMBER)

    # Sum for space diagonals
    space_diag1 = np.sum([cube[i, i, i] for i in range(n)])
    space_diag2 = np.sum([cube[i, i, n - 1 - i] for i in
range(n)])
    space_diag3 = np.sum([cube[i, n - 1 - i, i] for i in
range(n)])
    space_diag4 = np.sum([cube[n - 1 - i, i, i] for i in
range(n)])

    total_difference += abs(space_diag1 - MAGIC_NUMBER)
    total_difference += abs(space_diag2 - MAGIC_NUMBER)
    total_difference += abs(space_diag3 - MAGIC_NUMBER)
    total_difference += abs(space_diag4 - MAGIC_NUMBER)

    return -total_difference
```

b. Helper Functions

- generate_random_state()

→ Deskripsi fungsi: Fungsi ini untuk membuat konfigurasi *random* dari kubus 5×5×5 dengan angka dari 1 sampai 5^3 (1 sampai 125) yang di-*shuffle* atau diacak, lalu angka-angka tersebut nantinya diatur ke

dalam kubus 3 dimensi yang memiliki 5 lapisan, masing-masing berukuran 5×5.

→ *Source Code*:

```
def generate_random_state():
    numbers = np.arange(1, 126)
    np.random.shuffle(numbers)
    cube = numbers.reshape((5, 5, 5))
    return cube
```

- generate_neighbor(cube)

→ Deskripsi fungsi: Fungsi ini untuk menghasilkan *neighbor* atau versi baru dari susunan elemen *cube* 3 dimensi berukuran 5×5×5 melalui proses *copy* susunan asli dan pertukaran dua elemen secara *random* supaya terdapat *neighbor* yang sedikit berbeda dari susunan awal sehingga dapat digunakan dalam proses pencarian solusi yang lebih baik.

→ *Source Code*:

```
def generate_neighbor(cube):
    neighbor = cube.copy()
    flattened_cube = neighbor.flatten()
    idx1, idx2 = np.random.choice(len(flattened_cube),
size=2, replace=False)
    flattened_cube[idx1], flattened_cube[idx2] =
flattened_cube[idx2], flattened_cube[idx1]
    neighbor = flattened_cube.reshape(cube.shape)
    return neighbor
```

- visualize_cube(cube, title=”Cube State”)

→ Deskripsi fungsi: Fungsi ini untuk memvisualisasikan kubus 3 dimensi di mana setiap lapisan kubus 5×5 (5 baris dan 5 kolom) digambarkan dengan warna tertentu. Nilai elemen untuk setiap posisi di dalam lapisan ditampilkan sebagai teks di tengah kotak yang di mana maksud kotak di sini adalah representasi visual dari satu elemen kubus

saja (bukan keseluruhan kubus). Fungsi ini berguna dalam membantu memvisualisasikan kondisi kubus ketika titik waktu tertentu.

→ *Source Code*:

```
def visualize_cube(cube, title="Cube State"):
    fig, axes = plt.subplots(1, 5, figsize=(15, 3))
    fig.suptitle(title, fontsize=16)

    for i in range(5):
        axes[i].imshow(cube[i], cmap="viridis",
aspect="equal")
        for x in range(5):
            for y in range(5):
                axes[i].text(y, x, str(cube[i, x, y]),
va='center', ha='center', color="white")
        axes[i].set_xticks([])
        axes[i].set_yticks([])
        axes[i].set_title(f"Slice {i+1}")

    plt.show()
```

c. Run Function

- run_experiment()

→ Deskripsi fungsi: Fungsi ini untuk menjalankan algoritma-algoritma pencarian yang tersedia untuk persoalan *Magic Cube*, yaitu berbagai variasi *Hill Climbing* (*Steepest-Ascent Hill Climbing, Hill Climbing with Sideways Move, Stochastic Hill Climbing, Random Restart Hill Climbing*), *Simulated Annealing*, dan *Genetic Algorithm*. Pertama-tama, program akan menampilkan daftar keenam algoritma tersebut dalam urutan nomor 1-6. Lalu, program meminta *user* untuk memasukkan nomor dalam rentang 1 sampai 6 yang sesuai dengan algoritma yang ingin dijalankan. Jika nomor yang dimasukkan di luar rentang tersebut, program akan mengeluarkan pesan error dan meminta user untuk memasukkan nomor yang valid. Setelah *user* menginput nomor dari salah satu algoritma, program akan menjalankan algoritma tersebut

dengan memanggil fungsi algoritma bersangkutan yang telah dibuat sebelumnya. Algoritma yang dipilih akan dicoba sebanyak 3 kali *trial*. Khusus *Genetic Algorithm*, *user* akan diminta untuk memasukkan informasi tambahan yaitu 3 variasi ukuran populasi dan 3 variasi jumlah iterasi. Pada setiap *trial*, akan ditampilkan informasi unik dari masing-masing fungsi algoritma dan informasi yang berlaku untuk semua algoritma yaitu *Final Objective Function Value* (nilai objective function akhir yang dicapai), *Duration* (durasi proses pencarian dalam detik), grafik yang menunjukkan plot nilai *objective function* terhadap banyak iterasi yang telah dilewati, serta visualisasi state awal dan akhir kubus setelah pencarian.

→ *Source Code*:

```
def run_experiment():
    # Dictionary to store algorithm names and their
functions
    algorithms = {
        'Steepest Ascent Hill Climbing':
steepest_ascent_hill_climbing,
        'Hill Climbing with Sideways Move':
hill_climbing_with_sideways_move,
        'Stochastic Hill Climbing':
stochastic_hill_climbing,
        'Random Restart Hill Climbing':
random_restart_hill_climbing,
        'Simulated Annealing': simulated_annealing,
        'Genetic Algorithm': test_genetic_algorithm
    }

    print("Select the algorithm you want to run:")
    for idx, algo_name in enumerate(algorithms.keys(), 1):
        print(f"{idx}. {algo_name}")

    # User input
    choice = int(input("Choose the number of an algorithm:
```

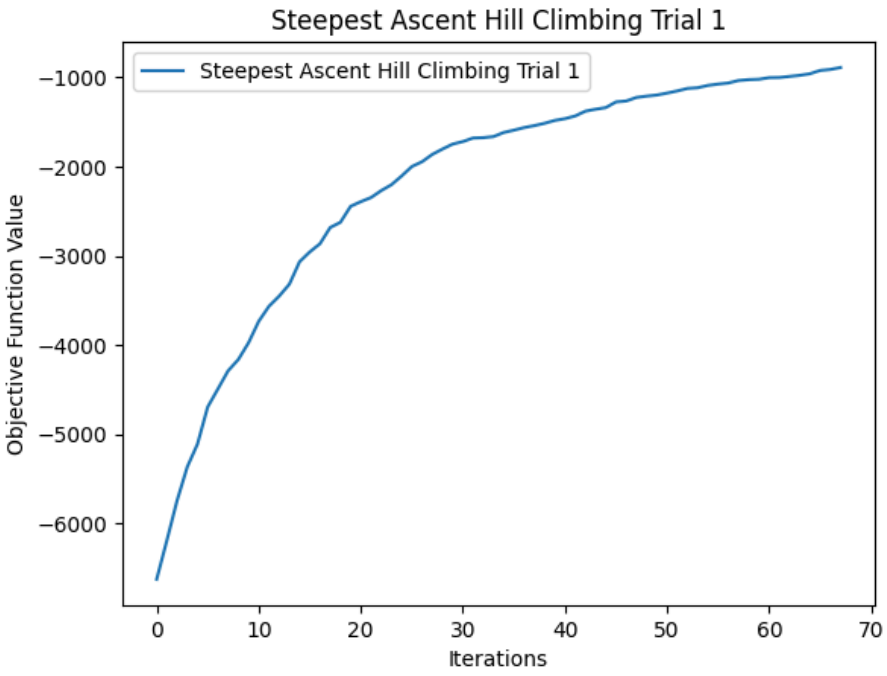
```
"))

    if choice < 1 or choice > len(algorithms):
        print("Invalid choice! Please select a valid
option.")
        return

    selected_algo_name = list(algorithms.keys())[choice -
1]
    selected_algo_func = algorithms[selected_algo_name]

    print(f"\nRunning {selected_algo_name}...")

    if selected_algo_name == 'Genetic Algorithm':
        selected_algo_func()
    else:
        for trial in range(3):
            print(f"\n\n+====================== TRIAL {trial
+ 1}: {selected_algo_name} ======================+")
            initial_state, final_state, final_value,
history, duration = selected_algo_func()
            print(f"Final Objective Function Value:
{final_value}")
            print(f"Duration: {duration:.2f} seconds")
            plt.plot(history, label=f'{selected_algo_name}
Trial {trial + 1}')
            plt.title(f'{selected_algo_name} Trial {trial +
1}')
            plt.xlabel("Iterations")
            plt.ylabel("Objective Function Value")
            plt.legend()
            visualize_cube(initial_state, title=f"Trial
{trial + 1} Initial State")
            visualize_cube(final_state, title=f"Trial
{trial + 1} Final State")
            plt.show()
```

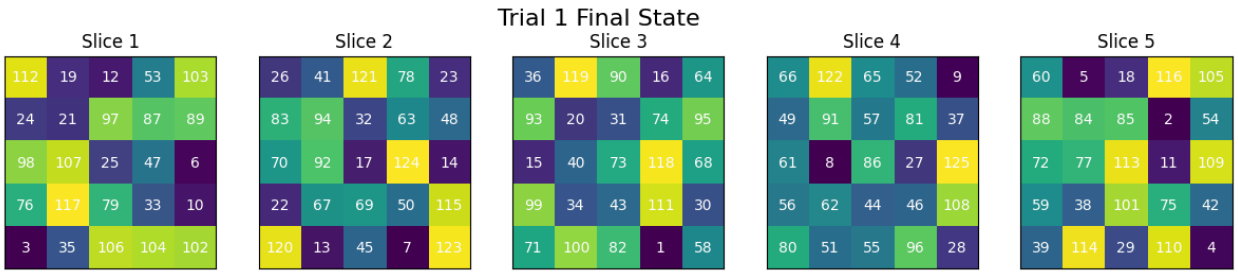
C. Hasil Eksperimen dan Analisis

- **Eksperimen**

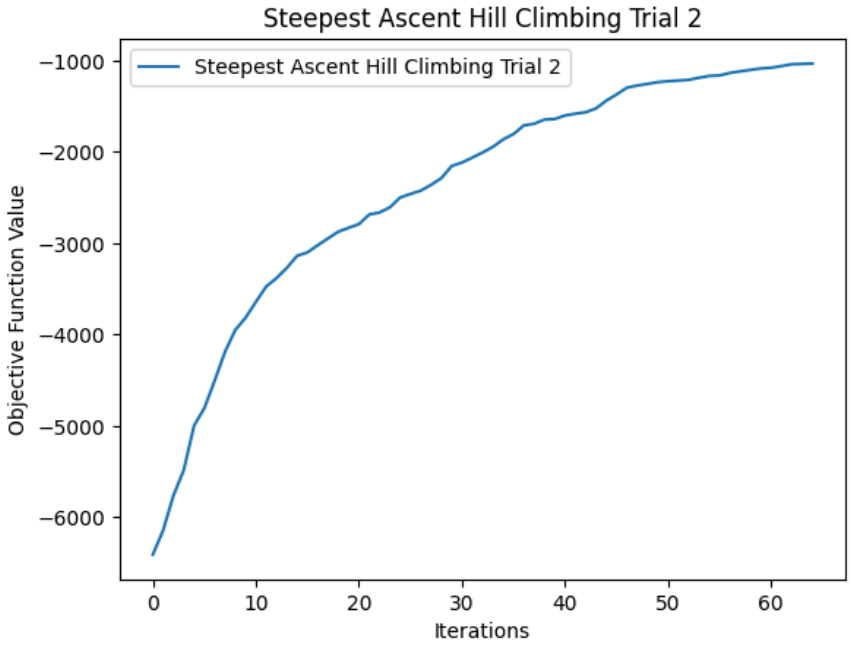
Berikut merupakan hasil analisis terhadap hasil eksperimen yang telah dilakukan.

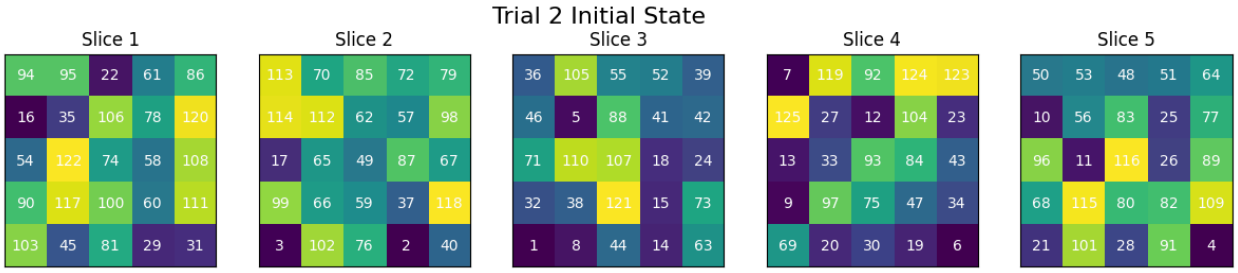
- ***Steepest Ascent Hill Climbing***

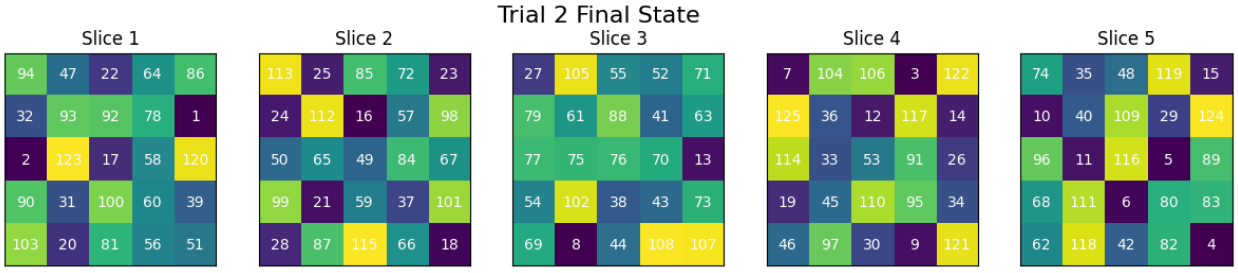




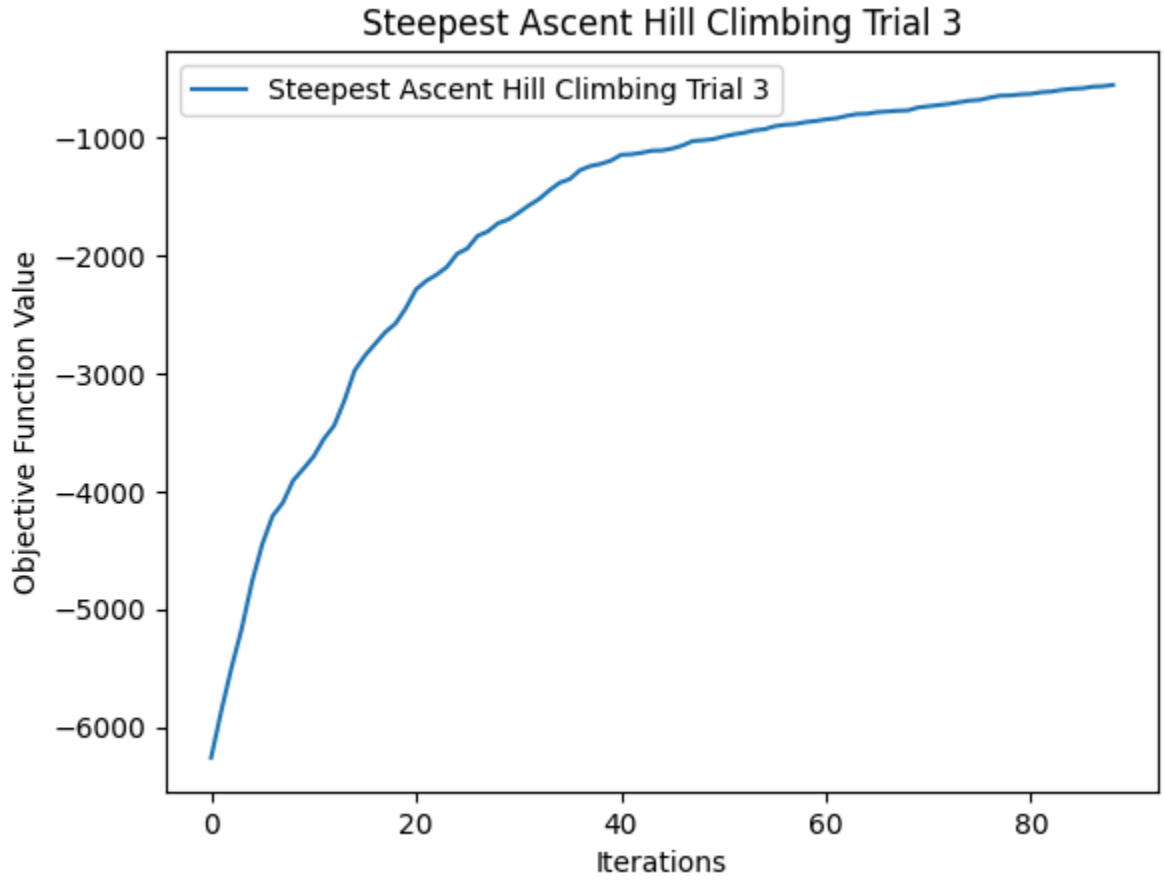
Pada percobaan pertama, algoritma *Steepest Ascent Hill Climbing* titik lokal maksimum dapat dicapai setelah dilakukan 68 iterasi dengan nilai akhir fungsi objektifnya adalah -888. Untuk mencapai hasil tersebut, dibutuhkan waktu selama 5,85 detik. Dari hasil tersebut, terlihat bahwa nilai akhir masih bernilai negatif sehingga masih ada solusi lebih baik yang lain, termasuk titik maksimum global.

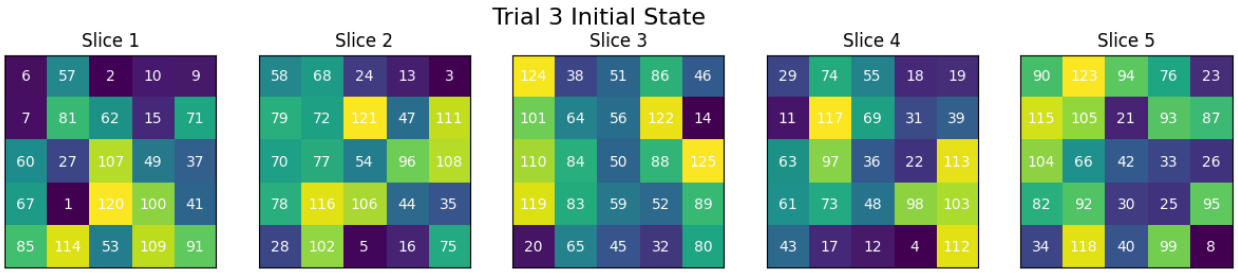


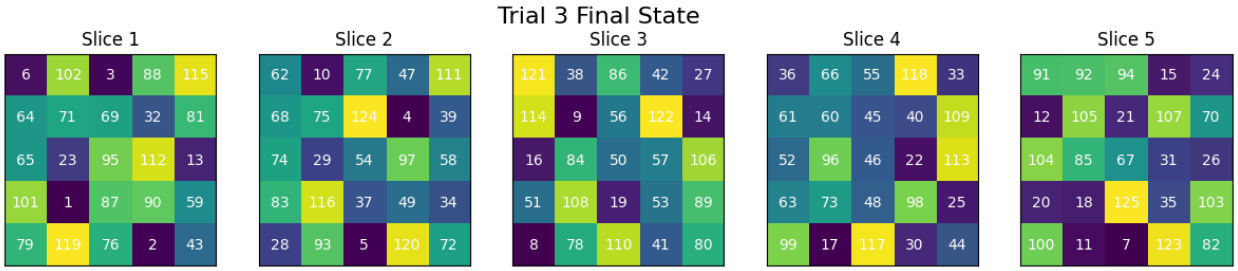




Pada percobaan kedua, algoritma *Steepest Ascent Hill Climbing* titik lokal maksimum dapat dicapai setelah dilakukan 65 iterasi dengan nilai akhir fungsi objektifnya adalah -1031. Untuk mencapai hasil tersebut, dibutuhkan waktu selama 3,67 detik. Dari hasil tersebut, terlihat bahwa walaupun jumlah iterasi dan waktu yang dibutuhkan lebih sedikit daripada percobaan pertama, nilai akhir masih bernilai negatif sehingga masih mencapai lokal maksimum dan ada solusi lebih baik yang lain, termasuk titik maksimum global.

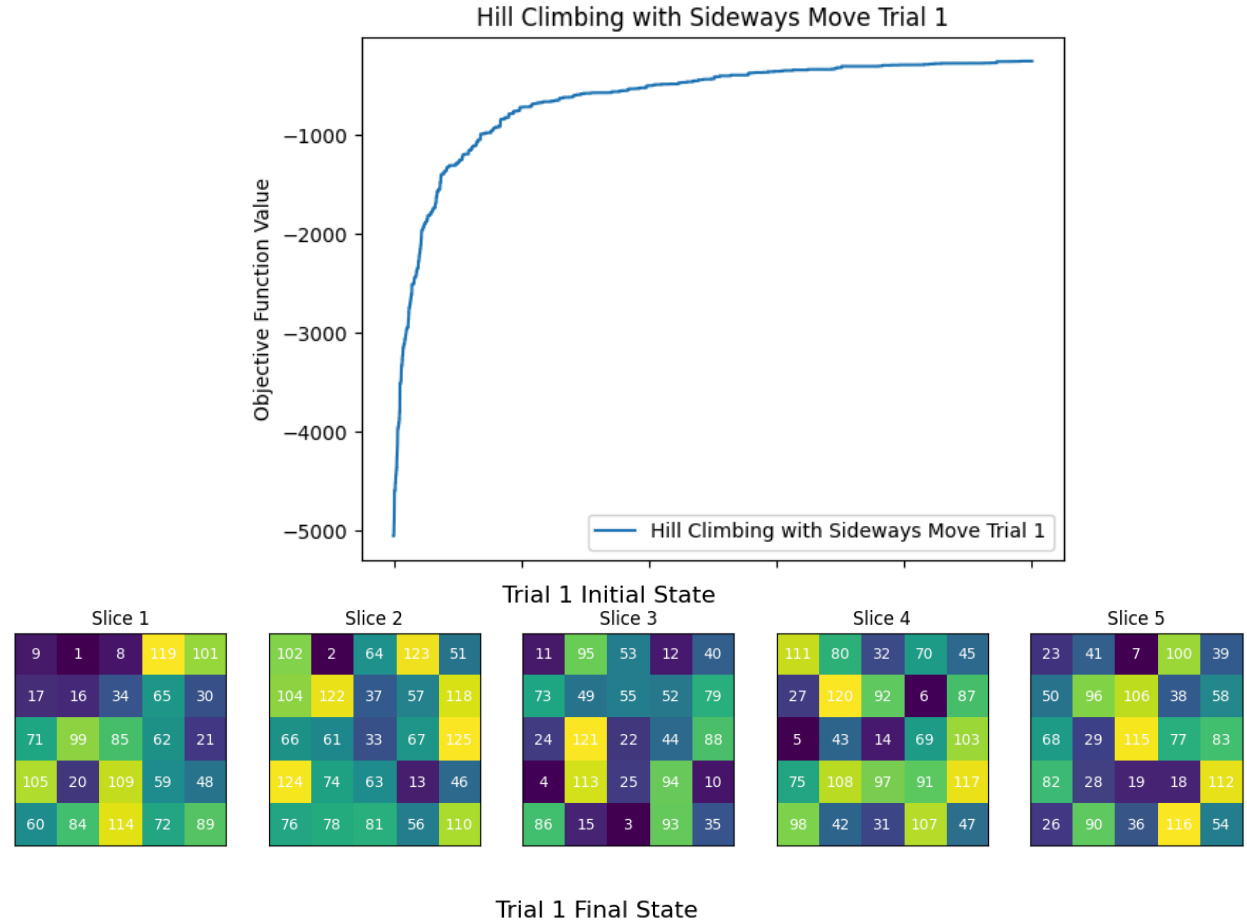




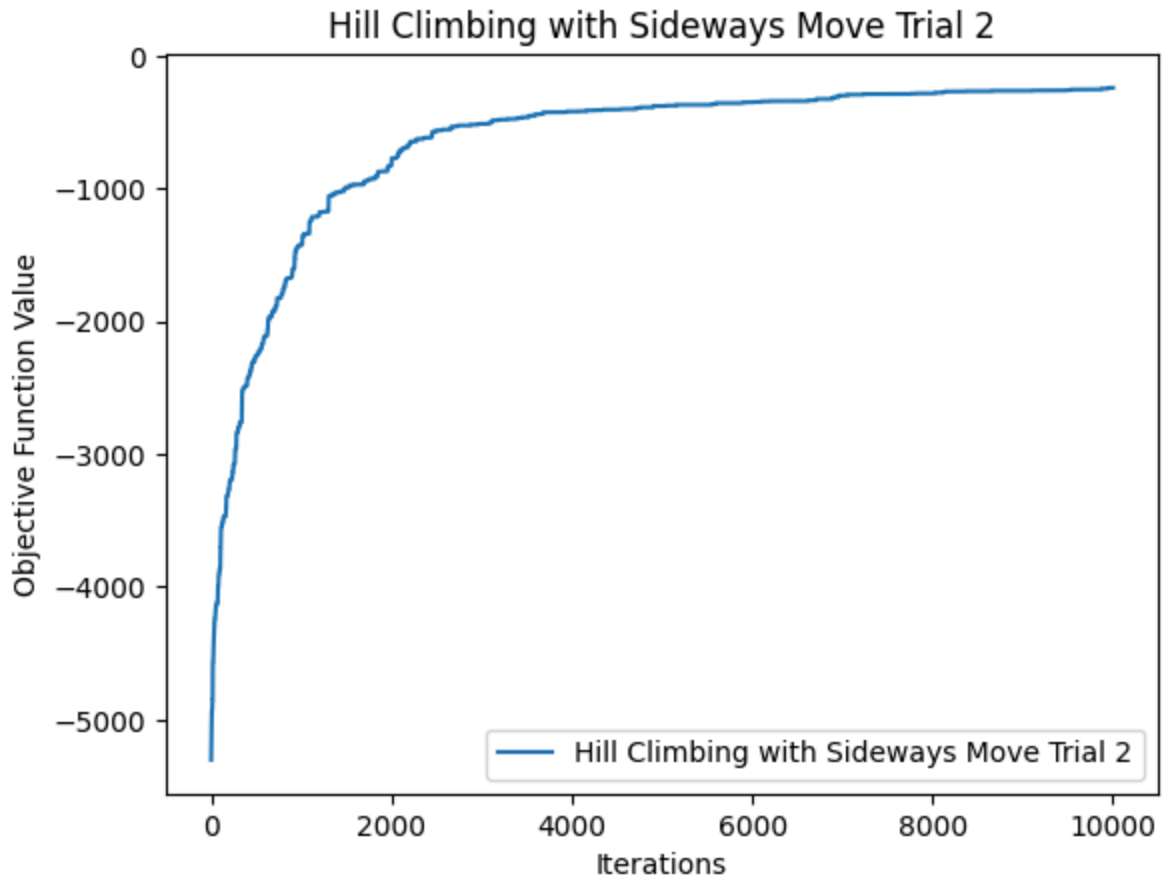


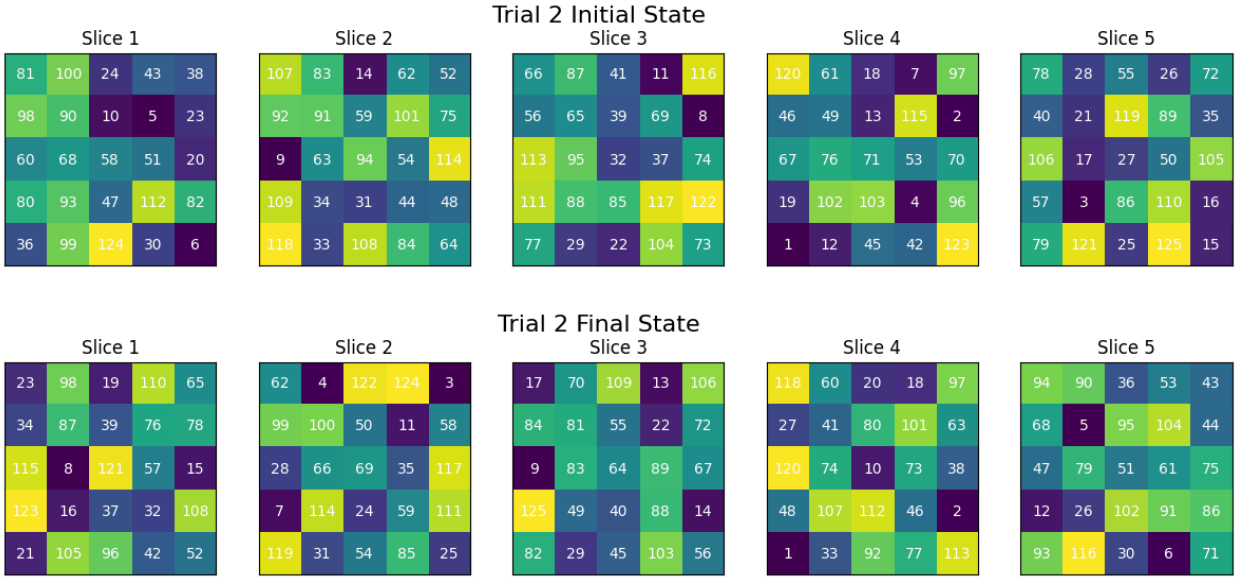
Pada percobaan ketiga, algoritma *Steepest Ascent Hill Climbing* titik lokal maksimum dapat dicapai setelah dilakukan 89 iterasi dengan nilai akhir fungsi objektifnya adalah -550. Untuk mencapai hasil tersebut, dibutuhkan waktu selama 5,9 detik. Dari hasil tersebut, terlihat bahwa nilai akhirnya merupakan yang paling mendekati 0 dibanding percobaan-percobaan sebelumnya sehingga merupakan solusi terbaik dibanding percobaan-percobaan sebelumnya. Walaupun begitu, percobaan ketiga ini juga hanya mencapai lokal maksimum dan belum mencapai titik maksimum global.

- ***Hill Climbing with Sideways Move***



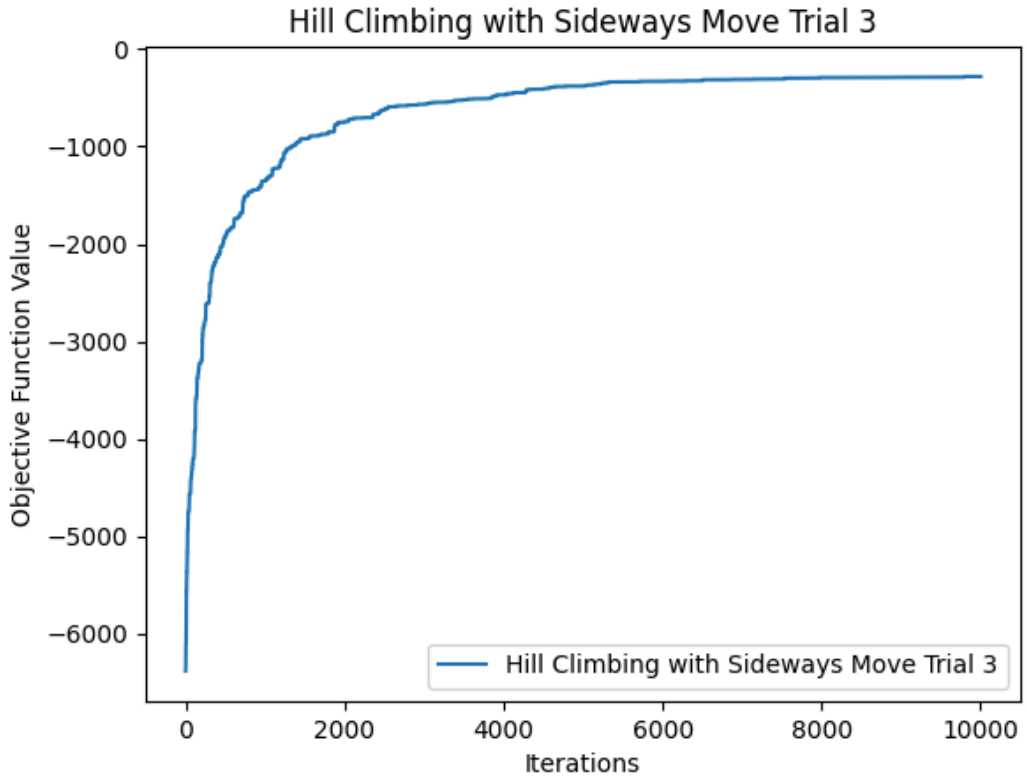
Pada percobaan pertama, titik optimum lokal pada algoritma *Hill Climbing with Sideways Mov*e dapat dicapai setelah tercapai batas maksimal atau dilakukan 10.000 iterasi dengan nilai akhir fungsi objektifnya adalah -251. Untuk mencapai hasil tersebut, dibutuhkan waktu selama 5,34 detik. Dari hasil tersebut, terlihat bahwa nilai akhir masih bernilai negatif sehingga masih ada solusi lebih baik yang lain atau belum mencapai titik global optimum.

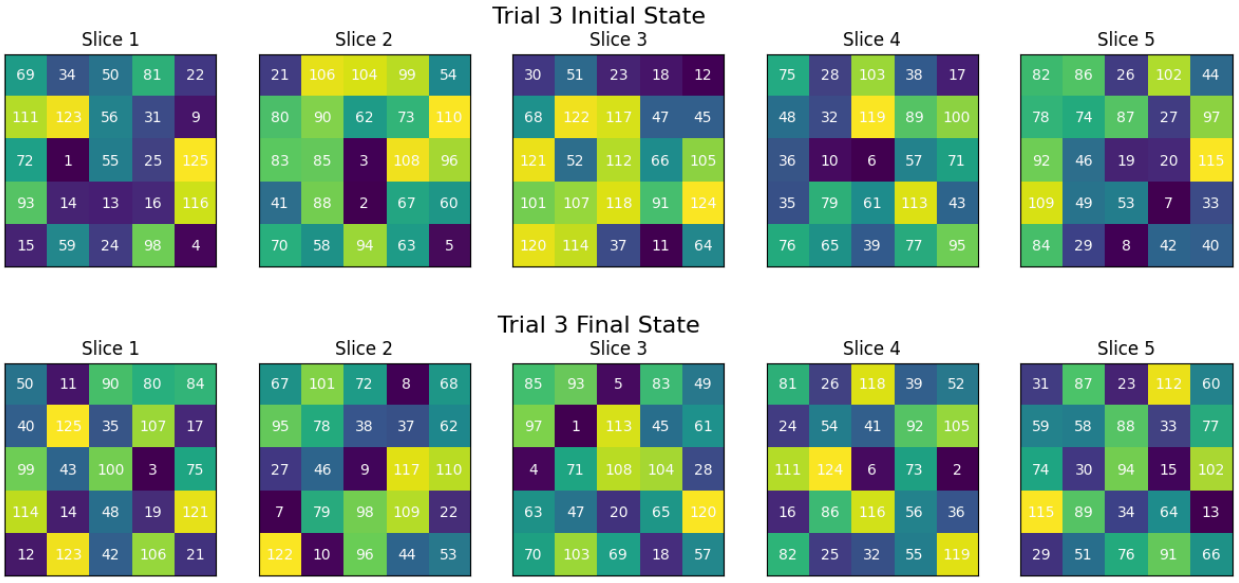




Pada percobaan kedua, titik optimum lokal pada algoritma *Hill Climbing with Sideways Mov*e dapat dicapai setelah tercapai batas maksimal atau dilakukan 10.000

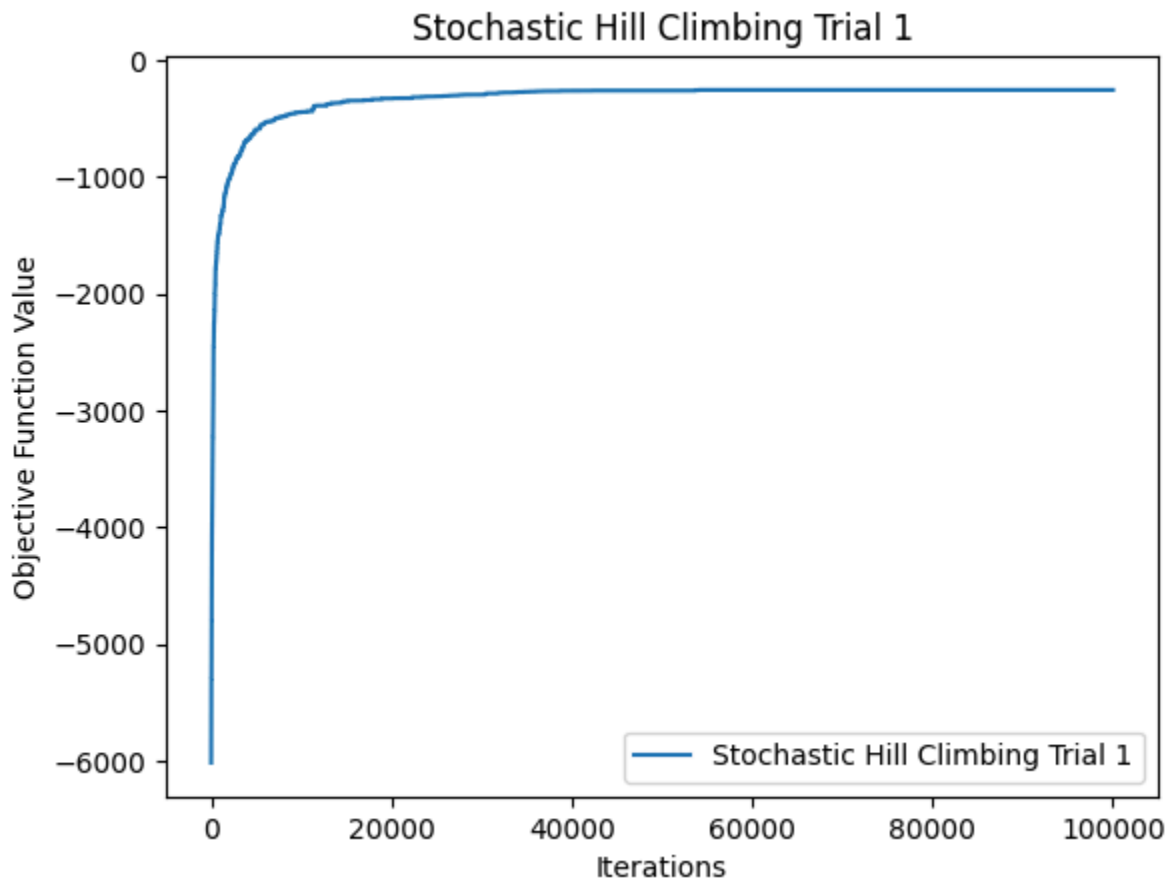
iterasi dengan nilai akhir fungsi objektifnya adalah -242. Untuk mencapai hasil tersebut, dibutuhkan waktu selama 7,44 detik. Dari hasil tersebut, terlihat bahwa nilai akhir masih bernilai negatif sehingga masih ada solusi lebih baik yang lain atau belum mencapai titik global optimum.

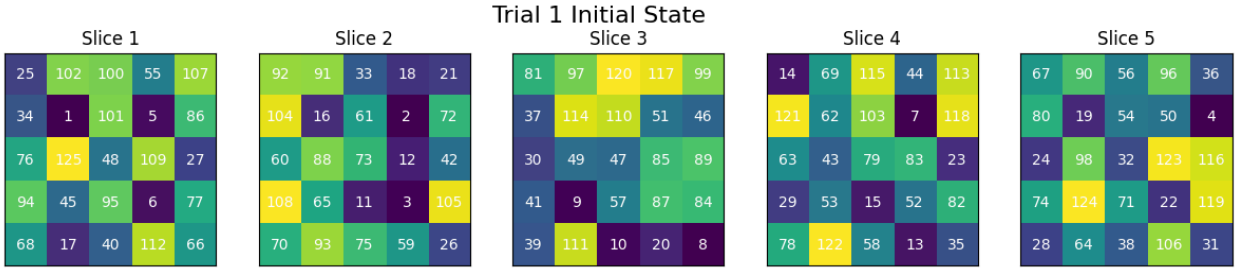


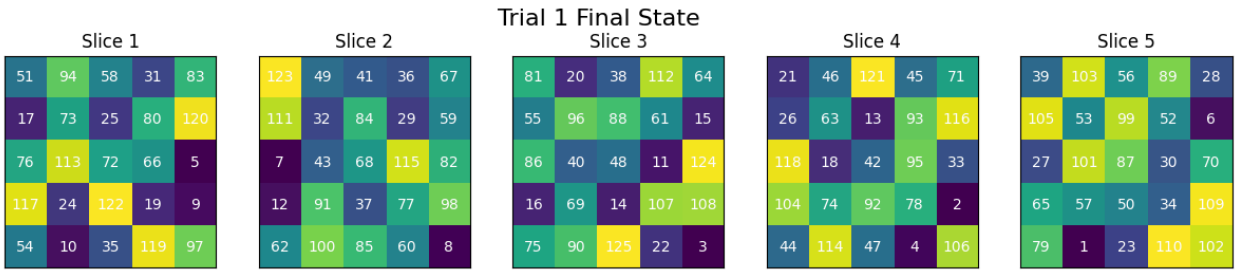


Pada percobaan ketiga, titik optimum lokal pada algoritma *Hill Climbing with Sideways Move* dapat dicapai setelah tercapai batas maksimal atau dilakukan 10.000 iterasi dengan nilai akhir fungsi objektifnya adalah -282. Untuk mencapai hasil tersebut, dibutuhkan waktu selama 6,1 detik. Dari hasil tersebut, pada percobaan ketiga ini terlihat bahwa nilai akhir masih bernilai negatif sehingga masih ada solusi lebih baik yang lain atau belum mencapai titik global optimum.

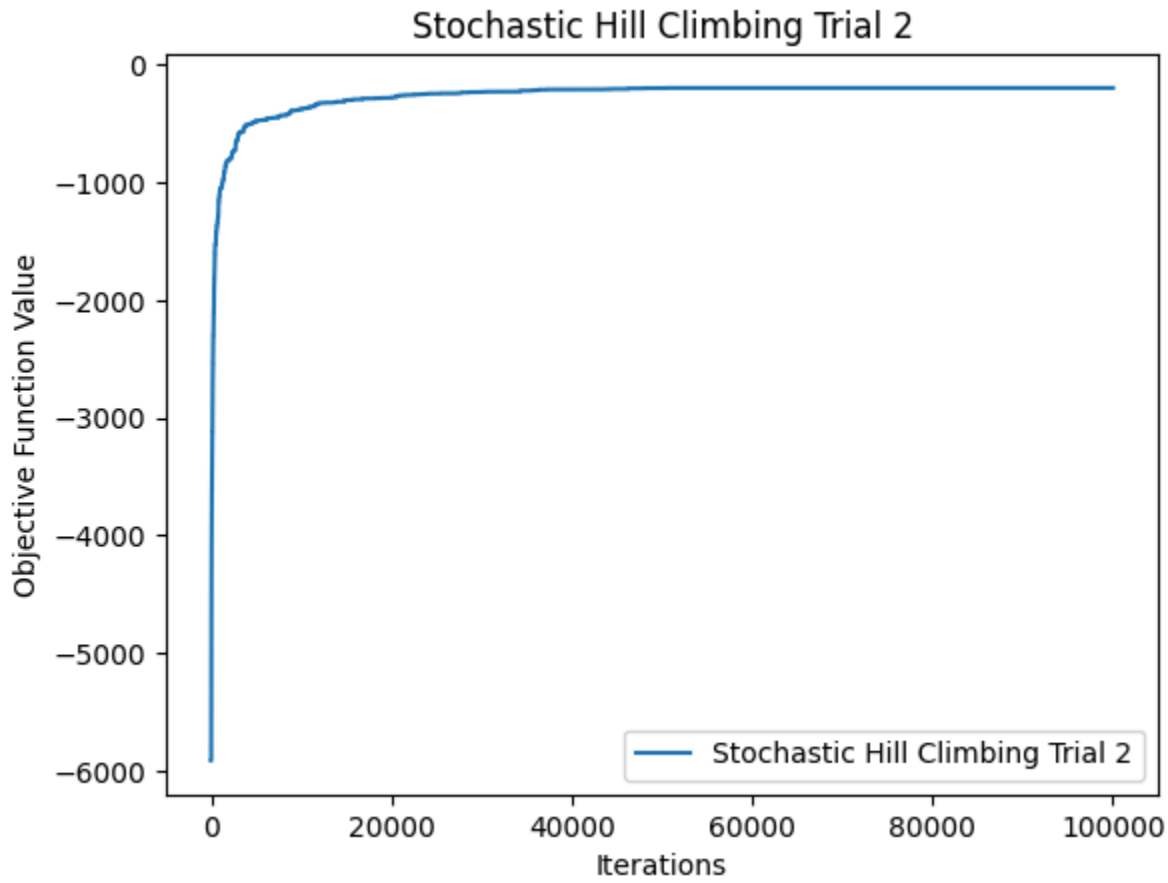
- ***Stochastic Hill Climbing***

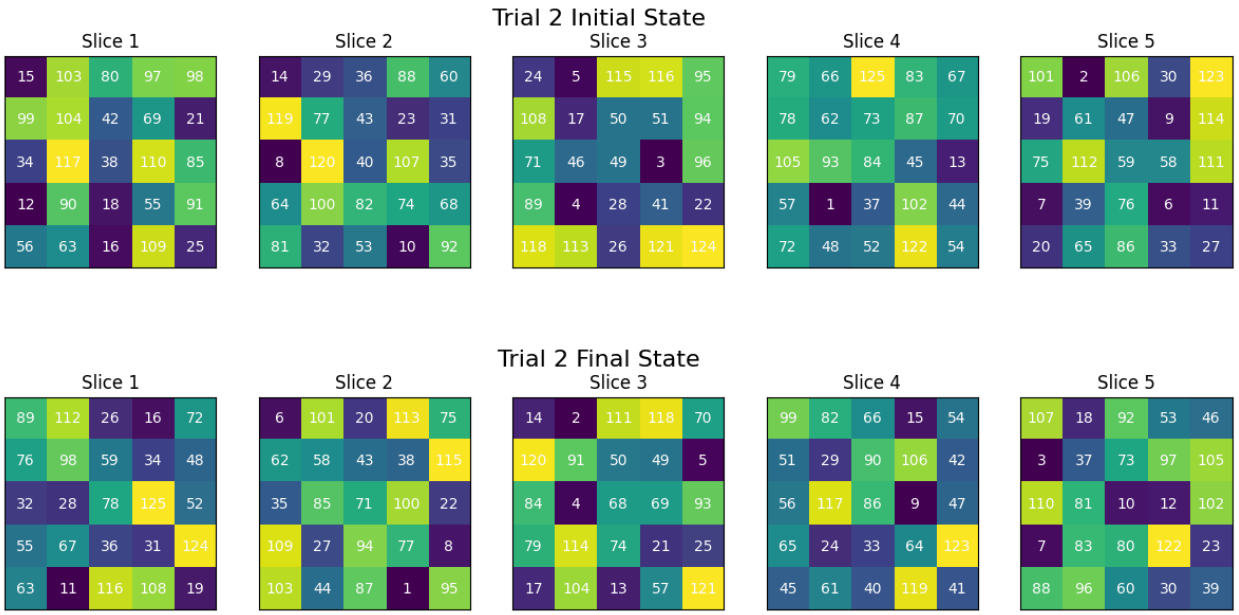




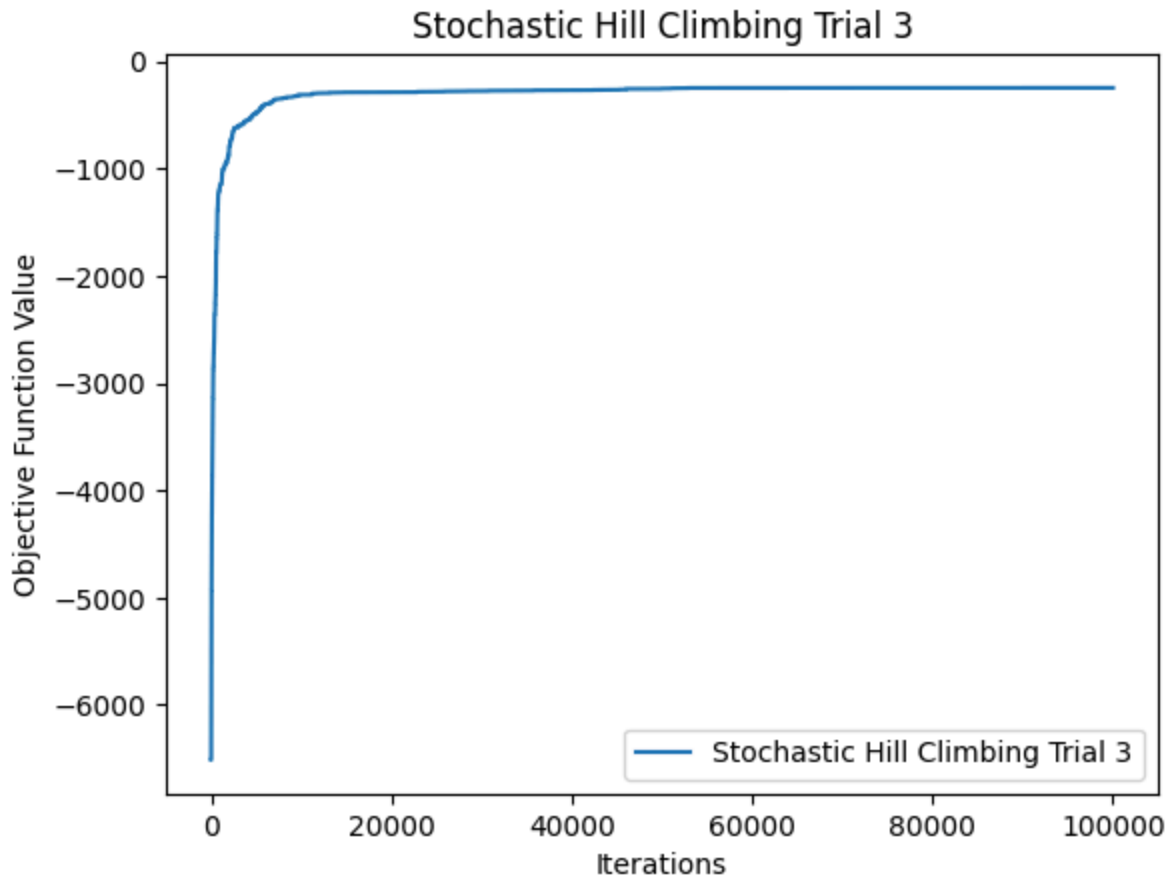


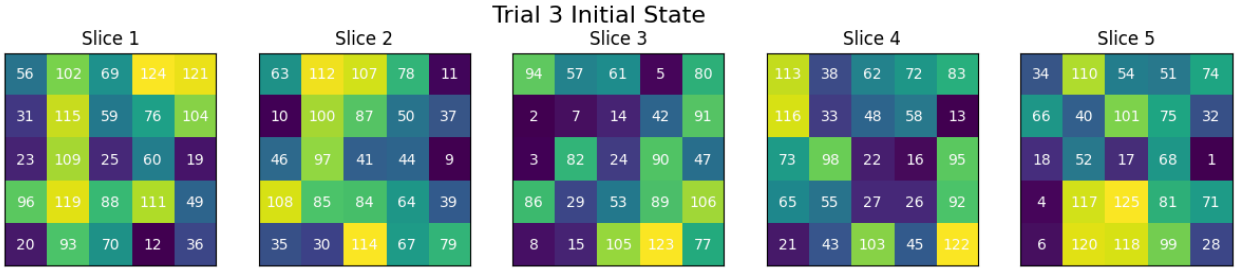
Pada percobaan pertama, titik optimum lokal pada algoritma *Stochastic Hill Climbing* dapat dicapai setelah tercapai batas maksimal atau dilakukan 100.000 iterasi dengan nilai akhir fungsi objektifnya adalah -256. Untuk mencapai hasil tersebut, dibutuhkan waktu selama 54,02 detik. Dari hasil tersebut, terlihat bahwa nilai akhir masih bernilai negatif yang artinya algoritma masih terjebak pada solusi optimum lokal sehingga masih ada solusi lebih baik yang lain atau belum mencapai titik global optimum.

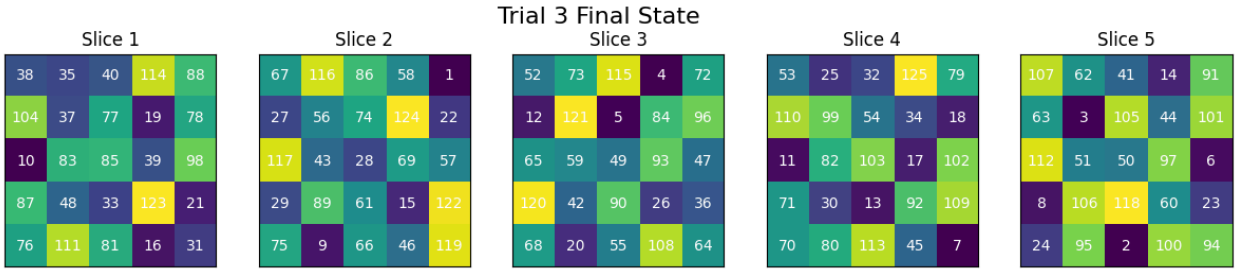




Pada percobaan kedua, titik optimum lokal pada dapat dicapai setelah tercapai batas maksimal atau dilakukan 100.000 iterasi dengan nilai akhir fungsi objektifnya adalah -198. Untuk mencapai hasil tersebut, dibutuhkan waktu selama 53, 88 detik. Walaupun nilai akhir fungsi objektifnya lebih baik dibandingkan percobaan pertama, nilai akhir tersebut masih bernilai negatif yang artinya algoritma masih terjebak pada solusi optimum lokal sehingga atau belum mencapai titik global optimum.


Stochastic Hill Climbing Trial 3
Objective Function Value
0
−1000
−2000
−3000
−4000
−5000
−6000
0
20000
40000
60000
80000
100000
Iterations
Stochastic Hill Climbing Trial 3


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

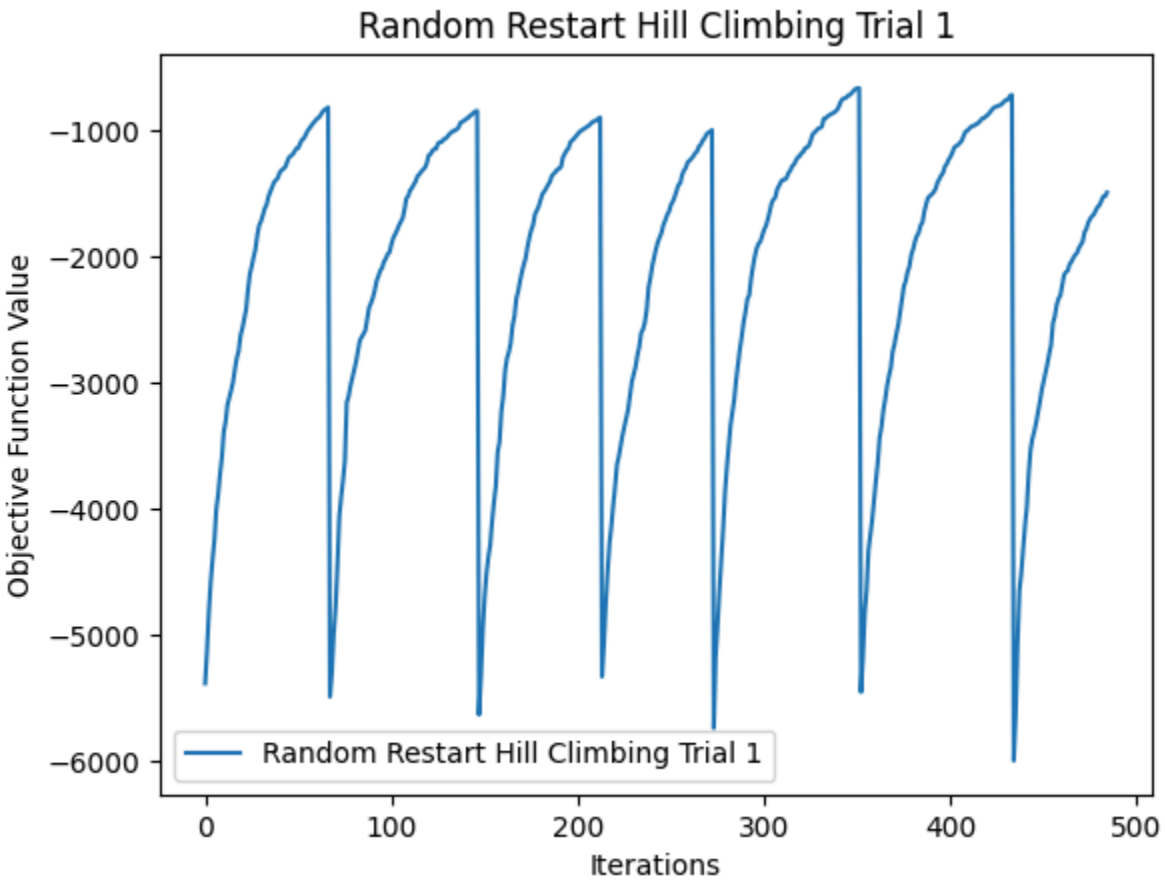

Trial 3 Final State
Slice 1
38 35 40 114 88
104 37 77 19 78
10 83 85 39 98
87 48 33 123 21
76 111 81 16 31
Slice 2
67 116 86 58 1
27 56 74 124 22
117 43 28 69 57
29 89 61 15 122
75 9 66 46 119
Slice 3
52 73 115 4 72
12 121 5 84 96
65 59 49 93 47
120 42 90 26 36
68 20 55 108 64
Slice 4
53 25 32 125 79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

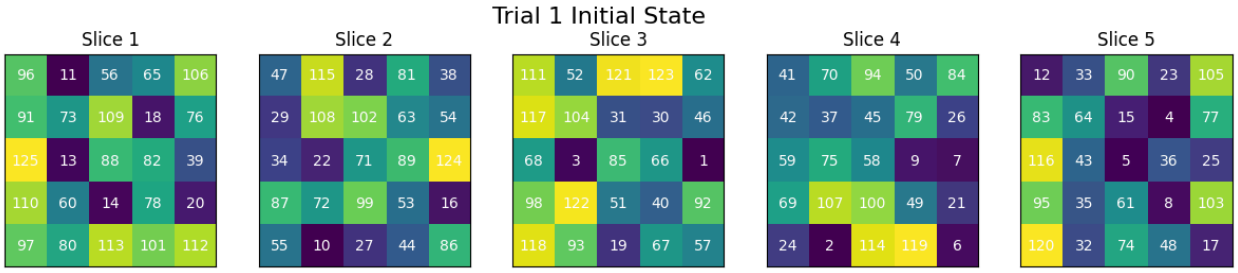
Pada percobaan ketiga, titik optimum lokal pada dapat dicapai setelah tercapai batas maksimal atau dilakukan 100.000 iterasi dengan nilai akhir fungsi objektifnya adalah -250. Untuk mencapai hasil tersebut, dibutuhkan waktu selama 55,18 detik. Dari hasil tersebut, terlihat bahwa nilai akhir fungsi objektifnya masih bernilai negatif. Artinya, algoritma masih terjebak pada solusi optimum lokal atau belum mencapai titik global optimum.

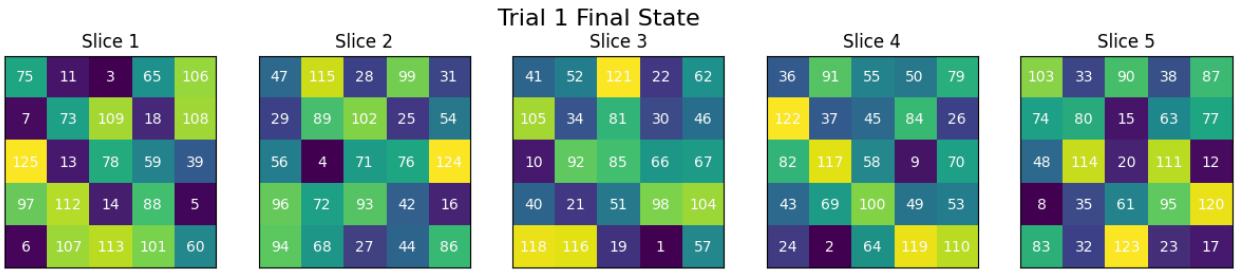
- ***Random Restart Hill Climbing***

```
Restart 1/7
Local maximum reached.
Steepest Ascent Hill Climbing finished after 67 iterations
Iterations in this restart: 67

Restart 2/7
Local maximum reached.
Steepest Ascent Hill Climbing finished after 80 iterations
Iterations in this restart: 80

Restart 3/7
Local maximum reached.
Steepest Ascent Hill Climbing finished after 66 iterations
Iterations in this restart: 66

Restart 4/7
Local maximum reached.
Steepest Ascent Hill Climbing finished after 60 iterations
Iterations in this restart: 60

Restart 5/7
Local maximum reached.
Steepest Ascent Hill Climbing finished after 79 iterations
Iterations in this restart: 79

Restart 6/7
Local maximum reached.
Steepest Ascent Hill Climbing finished after 82 iterations
Iterations in this restart: 82

Restart 7/7
Local maximum reached.
Steepest Ascent Hill Climbing finished after 51 iterations
```

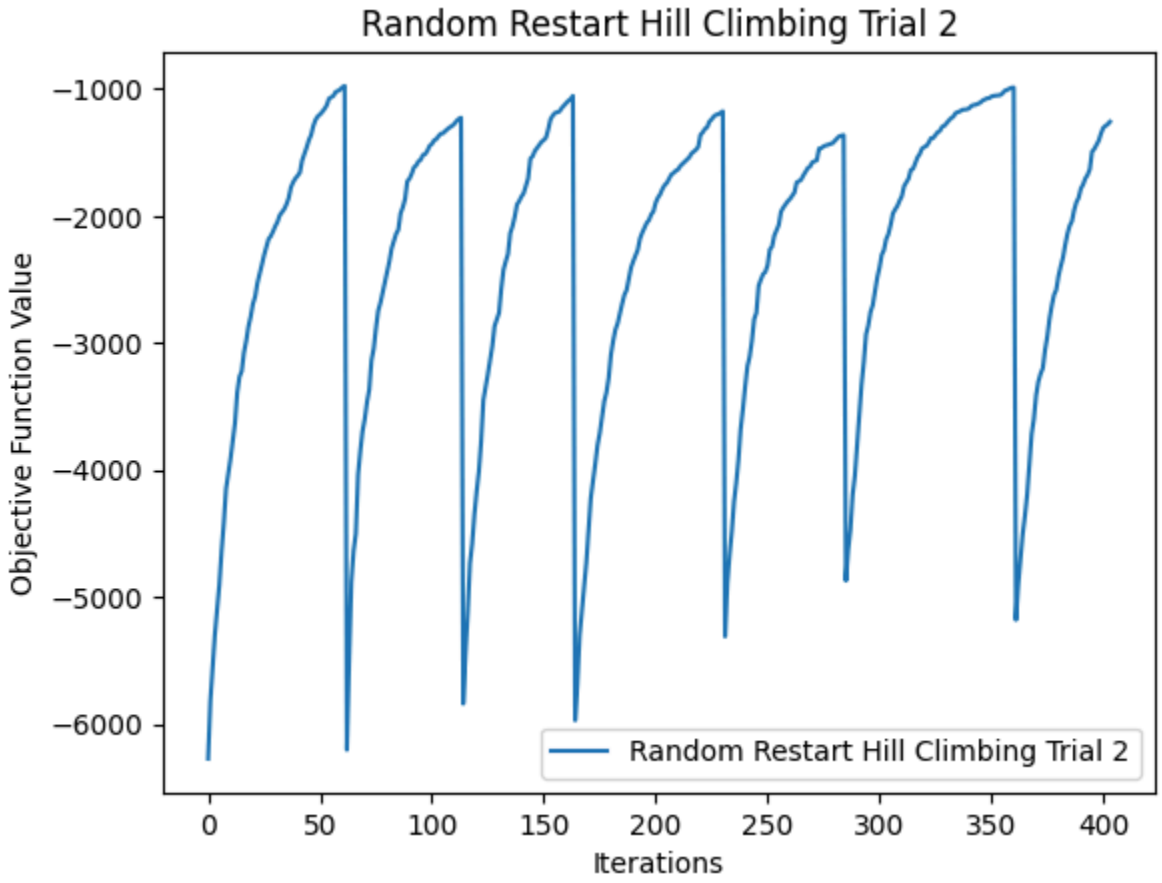
```
Iterations in this restart: 51
```

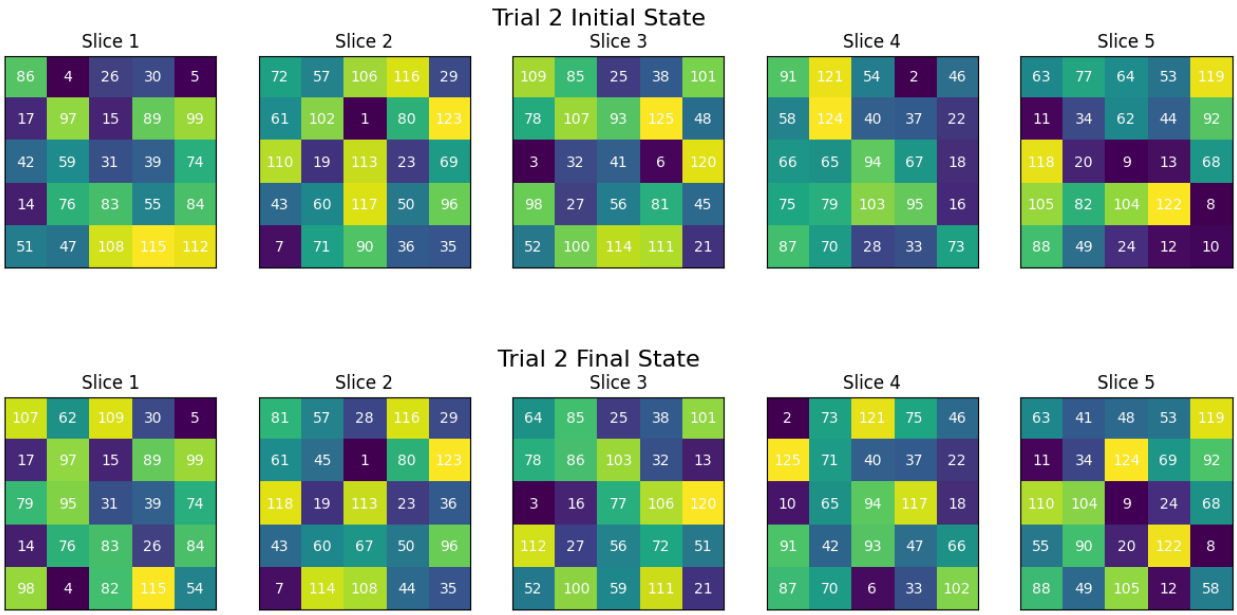




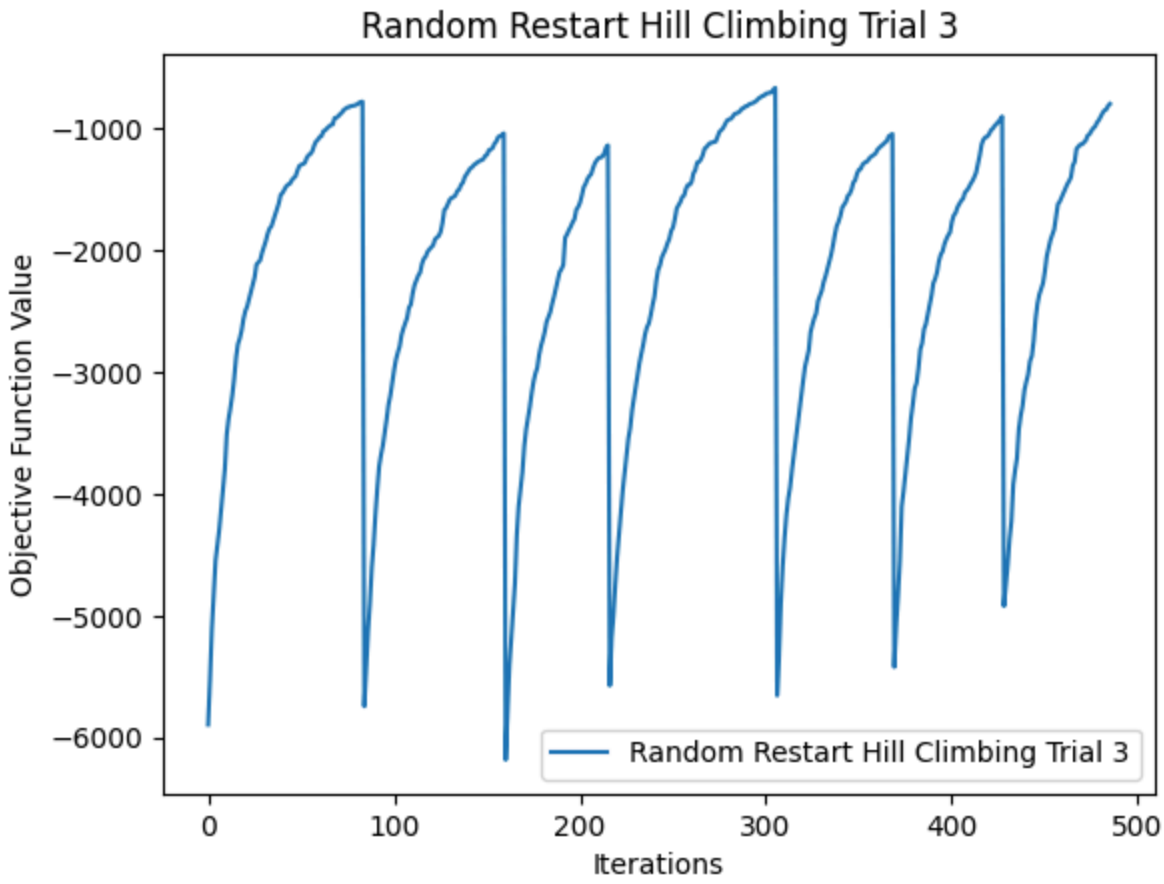


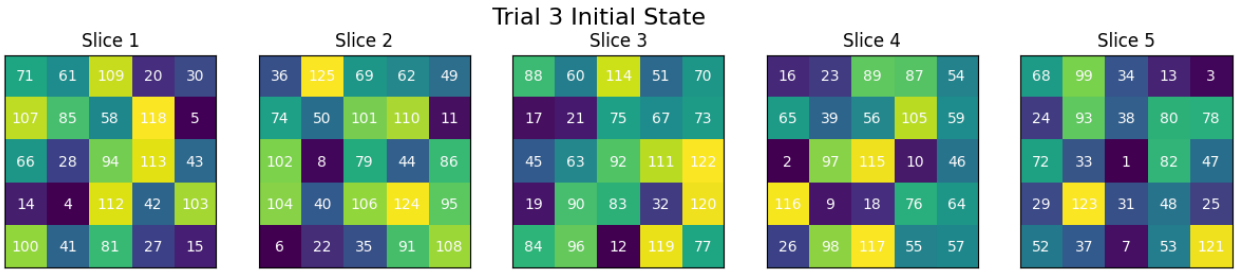
Pada percobaan pertama algoritma Random Restart Hill Climbing dilakukan restart sebanyak tujuh kali dengan setiap restart adalah secara acak atau random. Pada percobaan pertama ini, setiap restart hanya dapat mencapai titik optimum lokal dengan total iterasi yang dilakukan adalah sebanyak 485 kali. Waktu yang dibutuhkan untuk melakukan semua iterasi tersebut adalah 26,1 detik dengan hasil nilai akhir fungsi objektif yang didapatkan adalah -661. Nilai tersebut belum mencapai titik optimum global dan hanya mencapai titik optimum lokal.

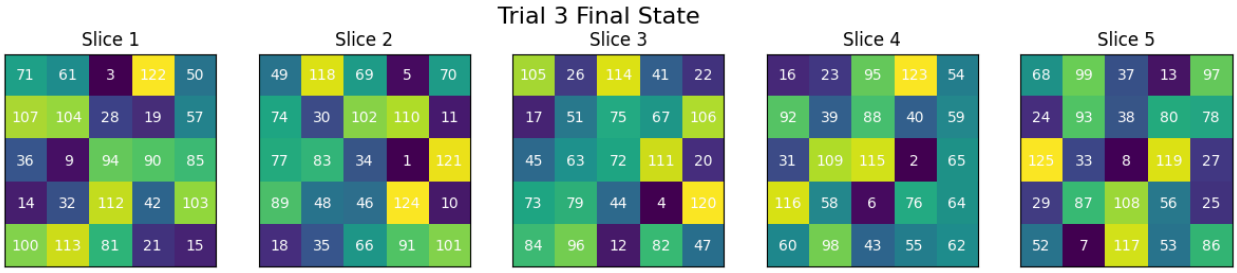




Pada percobaan kedua algoritma Random Restart Hill Climbing dilakukan restart sebanyak tujuh kali dengan setiap restart adalah secara acak atau random. Pada percobaan kedua ini, setiap restart hanya dapat mencapai titik optimum lokal dengan total iterasi yang dilakukan adalah sebanyak 404 kali. Waktu yang dibutuhkan untuk melakukan semua iterasi tersebut adalah 22,04 detik dengan hasil nilai akhir fungsi objektif yang didapatkan adalah –977. Nilai tersebut belum mencapai titik optimum global dan hanya mencapai titik optimum lokal.

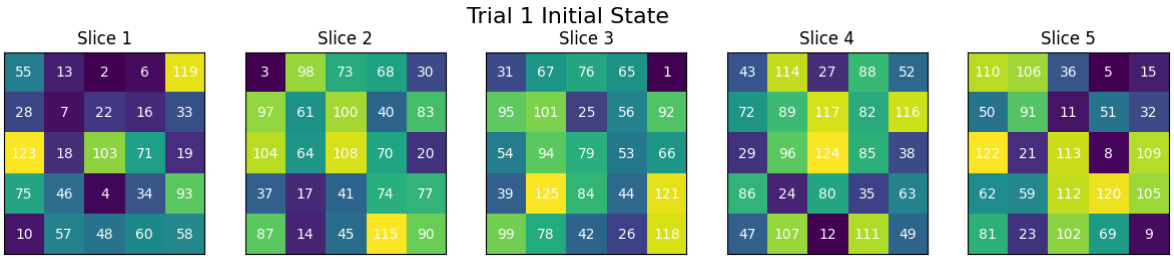

Random Restart Hill Climbing Trial 3
Objective Function Value
−1000
−2000
−3000
−4000
−5000
−6000
0
100
200
300
400
500
Iterations
Random Restart Hill Climbing Trial 3

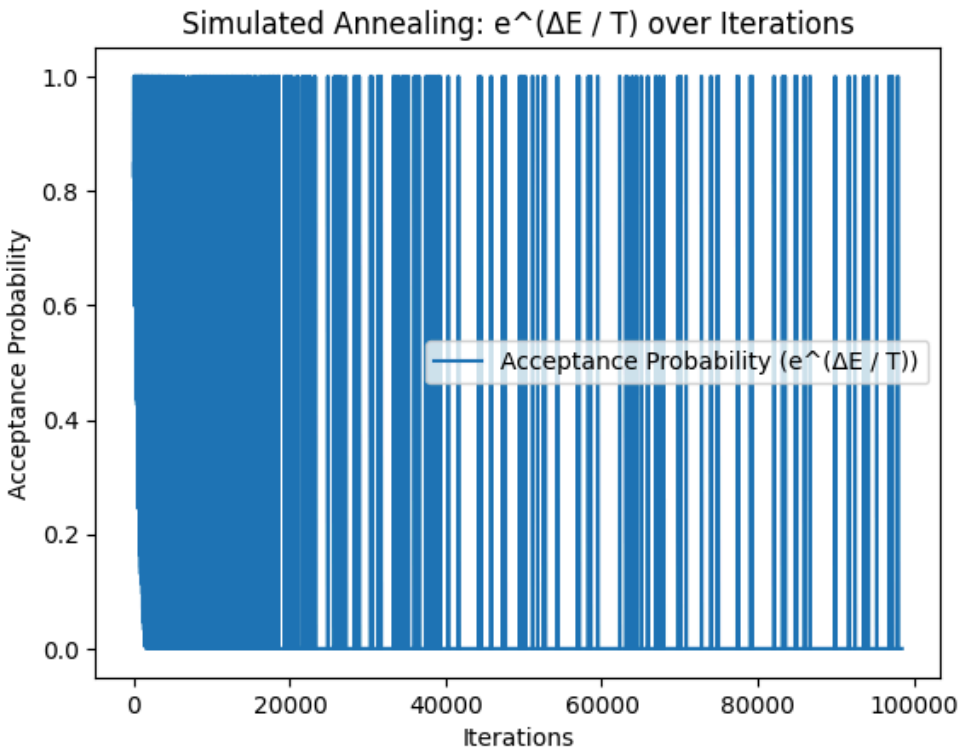

Trial 3 Initial State
Slice 1
71 61 109 20 30
107 85 58 118 5
66 28 94 113 43
14 4 112 42 103
100 41 81 27 15
Slice 2
36 125 69 62 49
74 50 101 110 11
102 8 79 44 86
104 40 106 124 95
6 22 35 91 108
Slice 3
88 60 114 51 70
17 21 75 67 73
45 63 92 111 122
19 90 83 32 120
84 96 12 119 77
Slice 4
16 23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

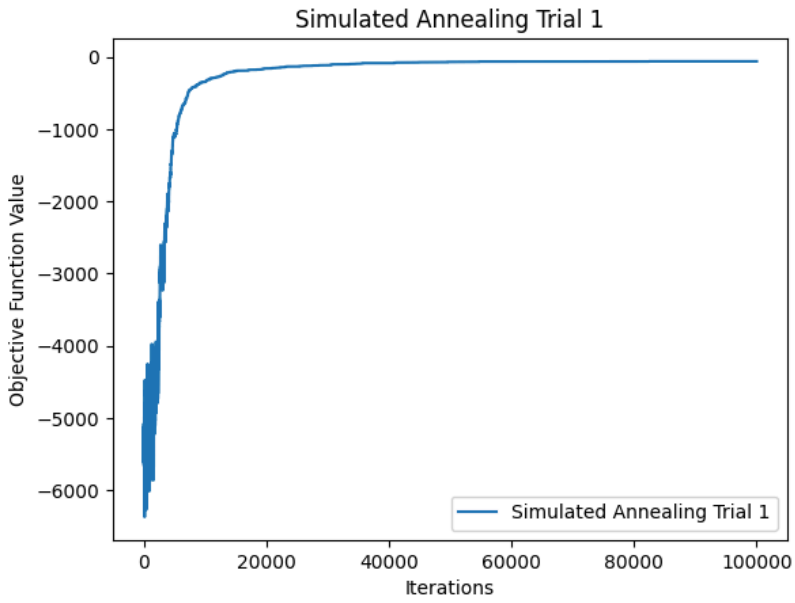

Trial 3 Final State
Slice 1
71 61 3 122 50
107 104 28 19 57
36 9 94 90 85
14 32 112 42 103
100 113 81 21 15
Slice 2
49 118 69 5 70
74 30 102 110 11
77 83 34 1 121
89 48 46 124 10
18 35 66 91 101
Slice 3
105 26 114 41 22
17 51 75 67 106
45 63 72 111 20
73 79 44 4 120
84 96 12 82 47
Slice 4
16 23 95 123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

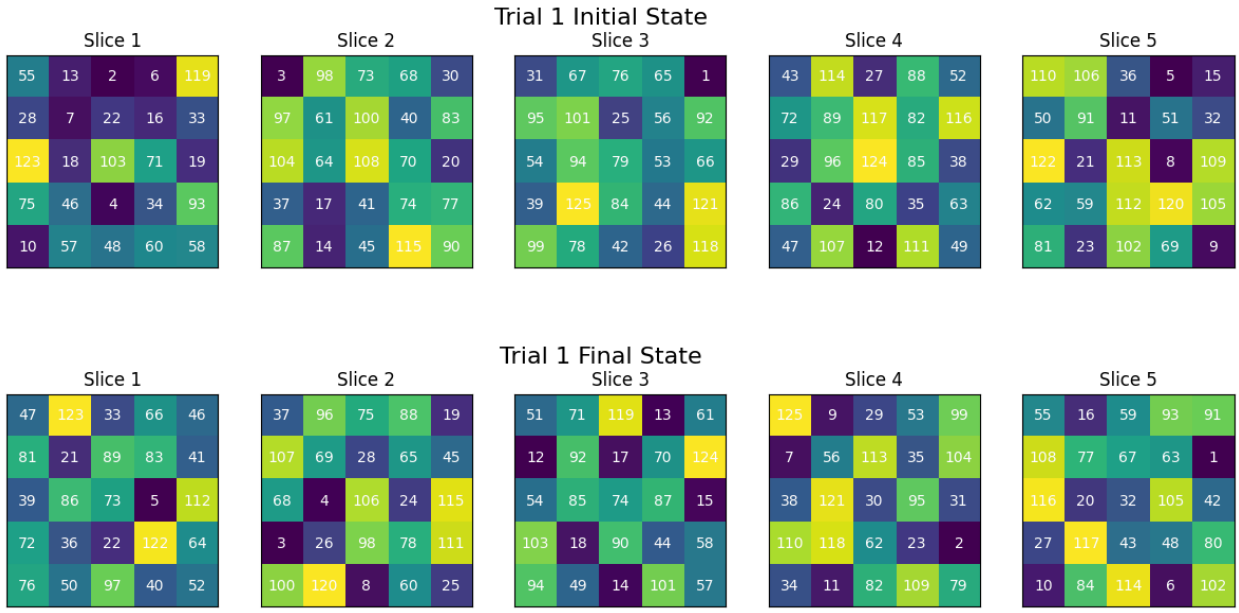
Pada percobaan ketiga, dilakukan restart sebanyak tujuh kali dengan setiap restart adalah secara acak atau random. Pada percobaan ketiga ini, setiap restart hanya dapat mencapai titik optimum lokal dengan total iterasi yang dilakukan adalah sebanyak 486 kali. Waktu yang dibutuhkan untuk melakukan semua iterasi tersebut adalah 26,1 detik dengan hasil nilai akhir fungsi objektif yang didapatkan adalah -662. Nilai tersebut lebih baik dibanding percobaan kedua namun sedikit lebih buruk dibanding percobaan pertama sehingga nilai akhir fungsi objektif tersebut belum mencapai titik optimum global.

- ***Simulated Annealing***



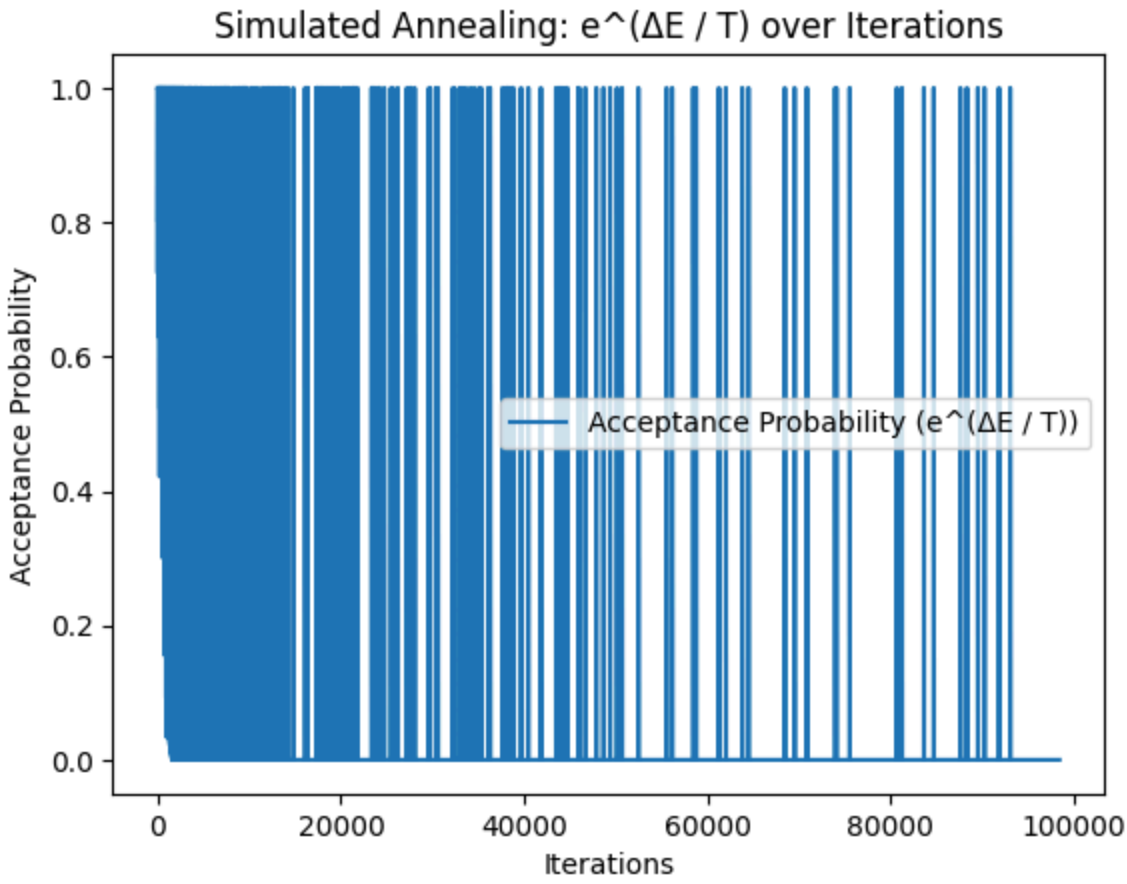


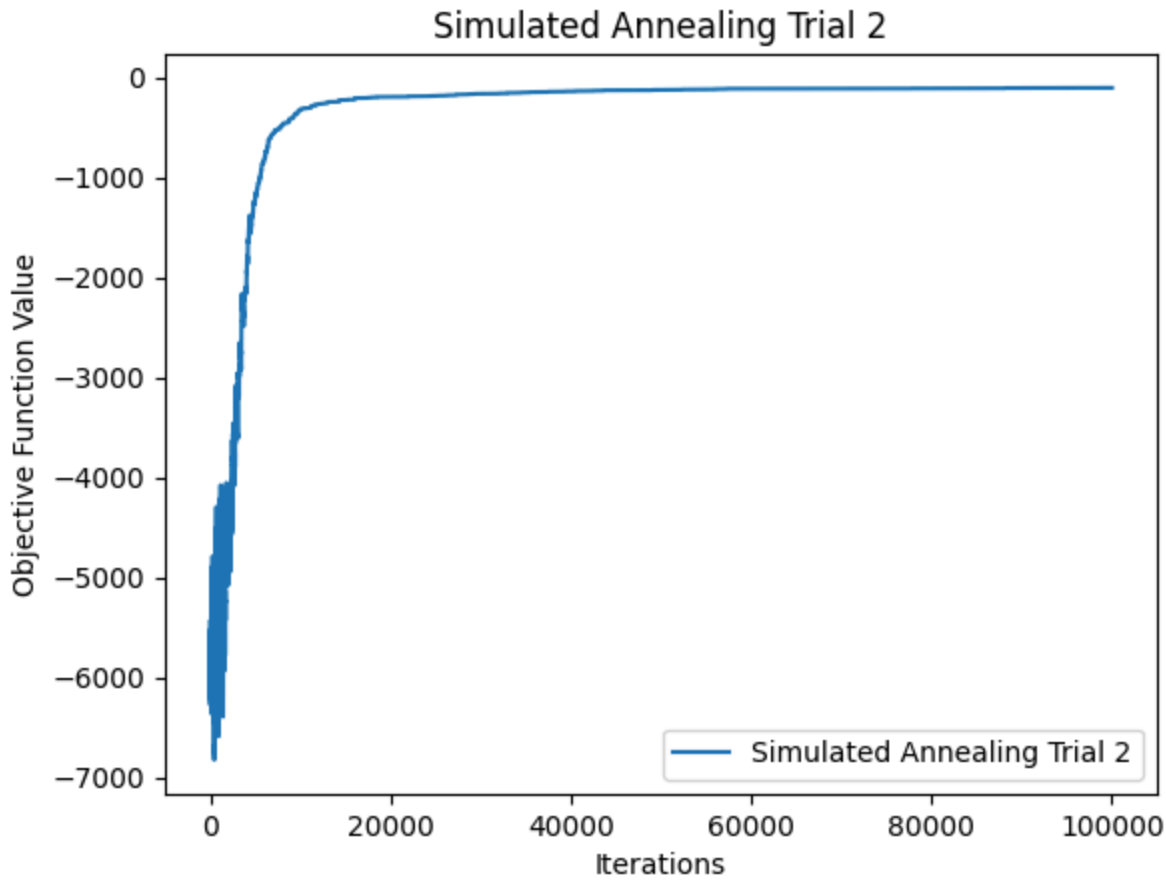


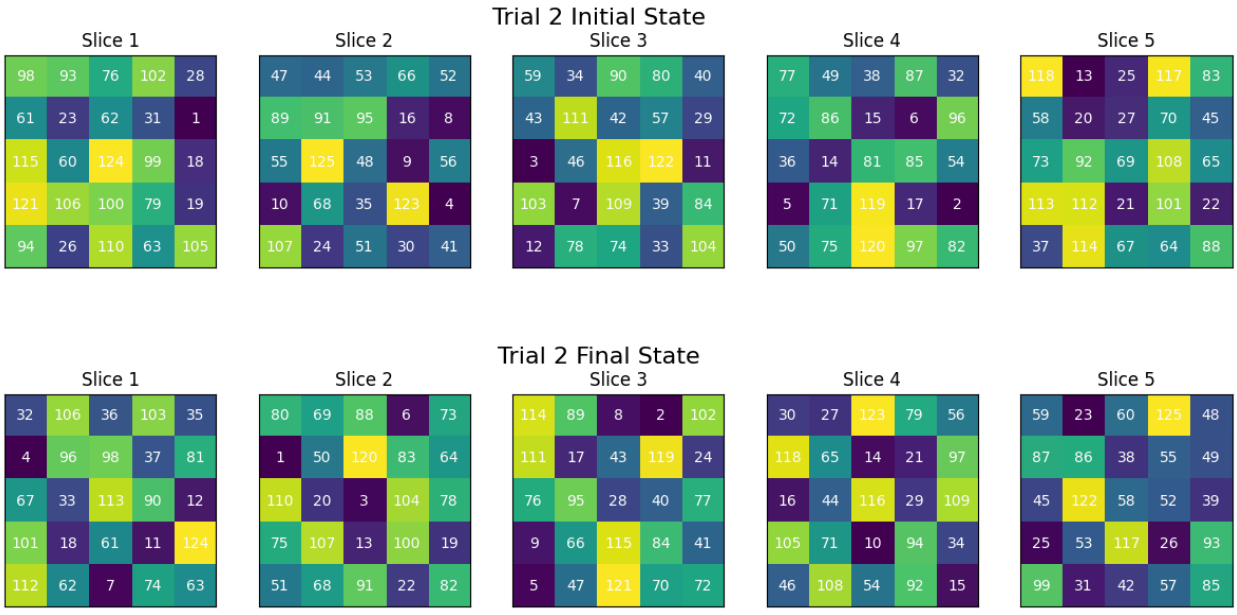


Pada percobaan pertama menggunakan algoritma Simulated Annealing, setelah tercapai batas maksimal atau dilakukan 100.000 iterasi didapatkan -60 sebagai nilai akhir fungsi objektif. Selama proses iterasi ini, algoritma ini terjebak pada optimum lokal sebanyak 96.706 kali. Untuk mencapai hasil tersebut, dibutuhkan waktu selama 54,24 detik. Dari hasil tersebut, walaupun nilai akhir fungsi objektif merupakan yang terbaik dibanding hasil percobaan lainnya, nilai akhir tersebut masih bernilai negatif yang artinya

algoritma masih terjebak pada solusi optimum lokal atau belum mencapai titik global optimum.

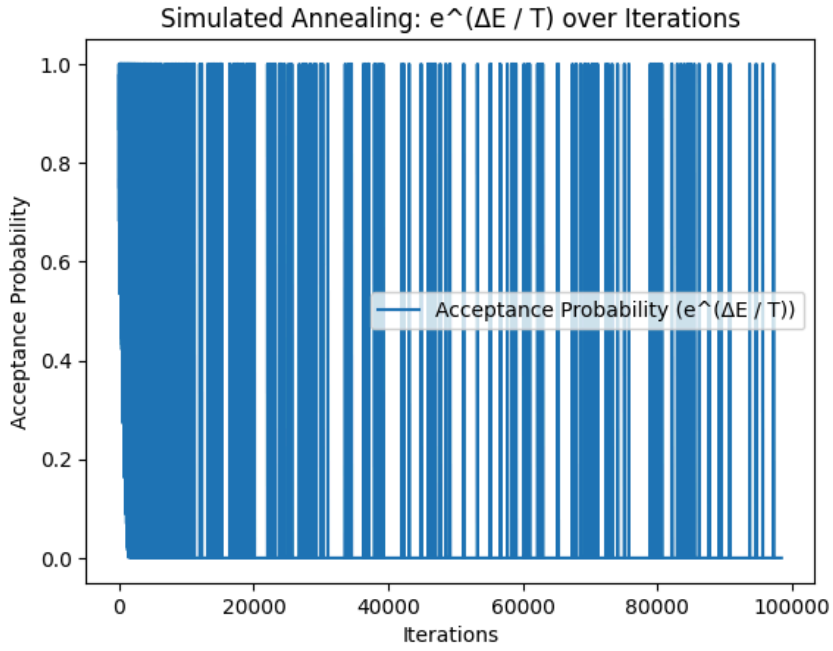

Simulated Annealing: e^(ΔE / T) over Iterations
Acceptance Probability
1.0
0.8
0.6
0.4
0.2
0.0
0
20000
40000
60000
80000
100000
Iterations
Acceptance Probability (e^(ΔE / T))

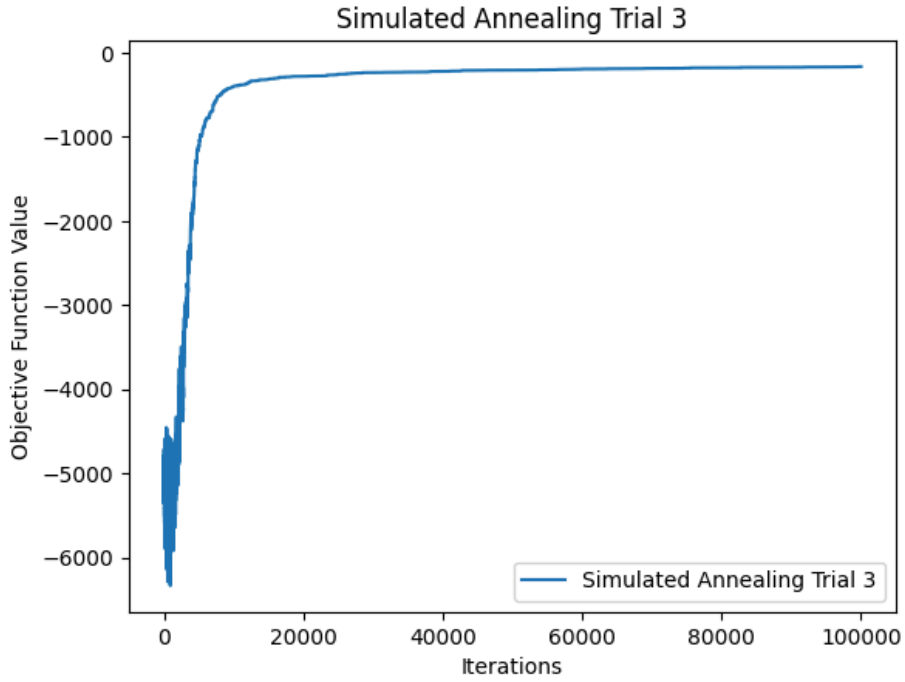


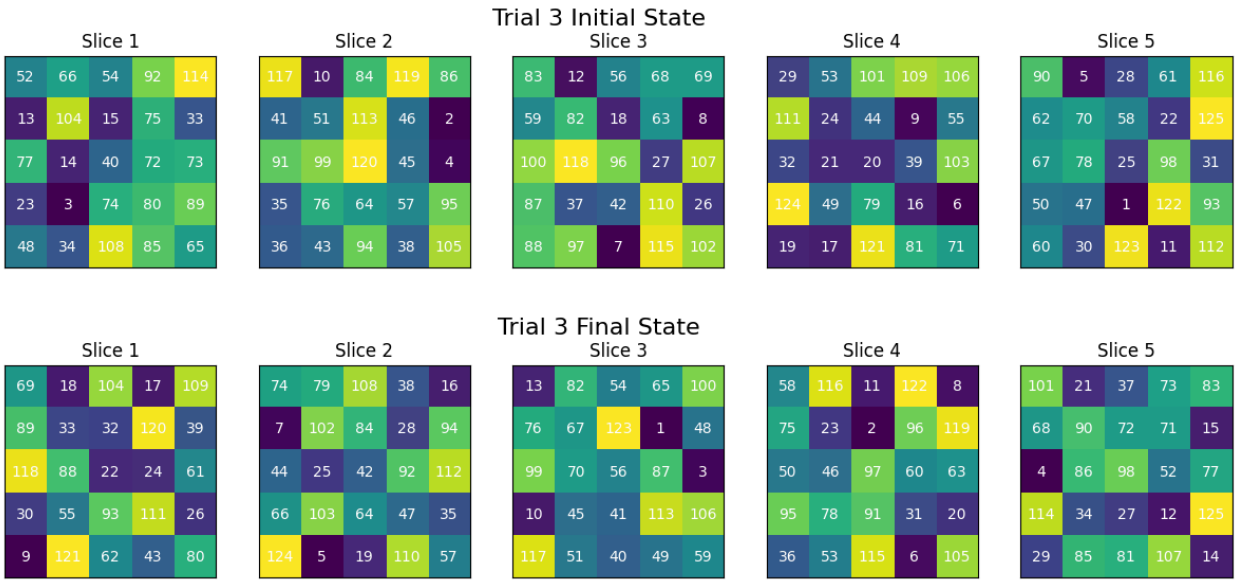


Pada percobaan kedua, setelah tercapai batas maksimal atau dilakukan 100.000 iterasi didapatkan -108 sebagai nilai akhir fungsi objektif. Selama proses iterasi ini,

algoritma ini terjebak pada optimum lokal sebanyak 96.693 kali. Untuk mencapai hasil tersebut, dibutuhkan waktu selama 54,12 detik. Dari hasil tersebut, terlihat bahwa nilai akhir fungsi objektif percobaan kedua ini lebih buruk dibanding percobaan pertama dan nilai akhir masih bernilai negatif. Artinya, algoritma masih terjebak pada solusi optimum lokal atau belum mencapai titik global optimum.







Pada percobaan ketiga, setelah tercapai batas maksimal atau dilakukan 100.000 iterasi didapatkan -163 sebagai nilai akhir fungsi objektif. Selama proses iterasi ini, algoritma ini terjebak pada optimum lokal sebanyak 96.572 kali. Untuk mencapai hasil tersebut, dibutuhkan waktu selama 54,33 detik. Dari hasil tersebut, terlihat bahwa nilai akhir fungsi objektif percobaan ketiga ini merupakan yang paling buruk jika dibandingkan dengan dua percobaan menggunakan algoritma simulated annealing lainnya.Nilai tersebut juga menunjukkan bahwa percobaan ketiga ini hanya dapat mencapai titik optimum lokal dan belum mencapai titik optimum global.

- ***Genetic Algorithm***

```
+==================== GENETIC ALGORITHM TRIAL 1 ====================+

Pop 100, 150 Iterations
  Duration: 8.26 seconds
  Final Objective Function Value: -2364

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 48 114  24 103  64       83   3  36  66  87       81  64  75  50  20      54  33  53  51 116      72 124 106  36  28
 50  86  93  15  47      101  74  92  42  49       79 104  34  94   4      85  42  93  74  36       1  31  10  96 103
 59  12 113  70  63       18  24  90  92  99       16 115  45   7  89      95  78  30  92  22      50  92  38  68  62
 46  28   6 118  54       79  58  94  49  24       56 101  72   9  93      70  58  49 108  26      66  50  88  58  91
103  12  65  58  80       25 120  13  53  45       77   2  86  12  87      39 119  71  63  69      52  61  80 105  55
```

```
Pop 100, 200 Iterations
  Duration: 10.51 seconds
  Final Objective Function Value: -2468

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 46  94  92  11  66      69  58  36 113  44       59  65 122  12  33      11   6 117  38  92     111  69   2  81  68
 72  69  19 111  81      94  16  64  57 118      106  93  14  77  51       8  86 125  32  62      46  17   7  82  78
 67  27  54 123  36      38 102 121  32  31       83  22 104  63  84      56 117  25 108   5      43  50 119   3 123
 68  55  86  62  35       5  76  51 106  82       47  72  58  53 114     103  72  21  57  70     109  48 101  18  36
 51  61  65  23 109     116  63  11  39  43       16  69 105  94  15      64  49  11  98 124      67  71  90  41  34

Pop 100, 250 Iterations
  Duration: 12.99 seconds
  Final Objective Function Value: -2295

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 11 115  50 105  38      93  53   6  51 106     119 111  12  59  44      77  41  49  13  26      55  27 104 103  35
 26  91  23  87  99      94  42  84  34 101      19  58  92  11  75      22  88 107 121  28     105  21   6  64  98
100  20  57  55  48       3  69  53  56  83      76  84  51 118  37     108  69  67  62  46      18 116 101  14 100
 82  52 109  60  19      60  23 104 108  28      38  93  85  17  97      73 102   1  10 113      69  53  51  99  50
 98  18  88  28 101      69 125  67  62  13      89   3  23 124  68      16   4 111 109  88      72  51 114  29  42
```

```
Pop 200, 150 Iterations
  Duration: 15.98 seconds
  Final Objective Function Value: -2314

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 86  16  62  81  88      41  87  44  56 107     45 101  38   5 114      26  78  89 109   5      66  31 112  85  23
 95  10  80  43  83      30  84  37  49  62     11 122 119  49  21      65  47  94  86 124     108  52  24 100  83
 35  74  32  73  67      72  57 120  93  12     97  44  27  75  64      35  54  73  71  57      46  58  79  84  55
 27 114   4 110  21     106  18  45  55 124     84  29 104  93  32      67 116  16  48  48      56   4  88  10 102
 84  31 109  11  92      60 100  69  47  14    121  62  25  60  41      49  37  33 106  91      43  72  90  68  47

Pop 200, 200 Iterations
  Duration: 21.87 seconds
  Final Objective Function Value: -2056

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 24   5  96 118  73      94 120  54  40   3     29 110  39  96  50     118  38  20   4 114      51  30  87  46 100
 74  62  97   7  79      25  78  71  75 114     59   8  15 115 117       5 125  94  67  10      71  88  26  28   3
 94  48  36  51  22      55  10 103 124  24     90  98  66  33   9      22  83  13  84 120      47  71  60  19 117
 20 117  57  39  56      88  72  26  20 108     94  65  74  69  24      56  14 116  73  40      63  17  62 108  68
 83  53  46  52  80     102   8  70  63  32     44  18 122  16 104      58 102  74  57  31      43 121  23  92  26
```

```
Pop 200, 250 Iterations
  Duration: 26.88 seconds
  Final Objective Function Value: -1820

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

Final State - Best Individual
Slice 1                 Slice 2                 Slice 3                 Slice 4                 Slice 5
 36  76  33 117  65      39  66 107   9  35     117  33  75  13  61      29  14 119  24  82      92 111  49  63  14
 13 109  54  27 122      51  53  96  80  14      57  15  95  96  34     104  33  26  86  73      88  97  51  47  56
 87  22 102  10 113      75  49  14  74  73     125 114  51  90  36       8 102 107  68  10       9  27  60 100 107
 92  63  82  41  29      60  72  42 118  91       9  54  46  71 113      68  32  45  77  74      87  87 108   1  25
 71  58  57  64  46      99  38  45  23  94      42 113  82  38  59      54 105  31  72  76      45  11  51 122  91

Pop 300, 150 Iterations
  Duration: 24.15 seconds
  Final Objective Function Value: -1961

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

Final State - Best Individual
Slice 1                 Slice 2                 Slice 3                 Slice 4                 Slice 5
 14 113  84  12  51      95   6  71 119  91      94  61  25  68  73      26  11 104  96  48      59  93   9  41   7
 23  81  25  98  83      87 107  26  35 117      79  15 101  55  31     115 118  83  40  10      58  24 113  67  77
123  72  13  61  57      40  41 107  97  11       5 107  63  69  65      99  49  42  74  71      46   6  66  80  87
 44  62  95  94  39      91  82  38  20  78      16  35  37 109 116      19 118  88  33 120      86  39 114  72  38
118   4  77  23  93      16  96  99 106   8     102  91  95  16  30      50  31   4 118 106      47 108  23  64  81
```

```
Pop 300, 200 Iterations
  Duration: 33.28 seconds
  Final Objective Function Value: -2193

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 20 101  28  98 114      53  66  75  88  33     113  77  32  54  42      82  48 118  22  72      81  36  75  79  69
 19  96   3  72  16      43  13 117  36 118      98  66  61  35  91     103  52   7 107  58      74  19 106 116  23
 86 115  98  52  99     110  56  74  49  52      10  45  55  86  66      44  95  40 117  48      72 121  76   6  53
122   9  94  25  12      23  93   6 107  85      23  51  99 124  85      92 114  17  71  16      67  33  68  30 122
 68   6  97  55  76      92 100   3  47  63     122  93  65   8  18      12   1 125 100 119       9 115  26 110  57

Pop 300, 250 Iterations
  Duration: 40.84 seconds
  Final Objective Function Value: -1690

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 50  42  65 122  23      63 107   1 100  61      44  83   3  73 112      85  81 121  31   1     102  28  88  26  72
 36  56  96  56  80      52  84  20  83  76      96 115 124   5   8     116  42  43  90  40       9  42  47  68  84
 82  62  22  11  98      66  95  68  41  32      18  23  60  89  49      48  50 108  53  51     109  15  57  95  82
 72  69 108  77  25      51  14 110  45  66      62   7  11  75 111      49 122  30  63 107      56  87  70  71  16
 74  68  35  86  71     103  13 112  43  48     113  80  99  55  19      11  36   4 104 103      14 117  65  37  58
```

Pada percobaan pertama, dipilih ukuran populasi sebanyak 100, 200, dan 300 individu dengan jumlah iterasi sebanyak 150, 200, dan 250 kali. Pada populasi 100, divariasikan iterasi sebanyak 150, 200, dan 250 kali, begitu pula dengan individu 200 dan 300. Waktu atau durasi proses untuk jumlah individu yang sama akan semakin besar semakin banyak jumlah iterasinya. Untuk nilai akhirnya cenderung lebih baik semakin banyak individu atau iterasi yang dilakukan. Misal pada populasi 100 individu, iterasi 150 kali memerlukan waktu 8,26 detik dengan nilai akhir -2364 sedangkan pada 250 iterasi memerlukan 12,99 detik dengan nilai akhir -2295. Begitu pula untuk jumlah iterasi yang sama, semakin banyak individu semakin besar juga waktu yang diperlukan dan semakin mendekati lebih baik pula nilai akhirnya.

```
+==================== GENETIC ALGORITHM TRIAL 2 ====================+

Pop 100, 150 Iterations
  Duration: 8.30 seconds
  Final Objective Function Value: -2265

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 82 111 108  38  44      80  27  65  49  81      36  80  31 119  20      49  21  55  85  35      75  64  66   2 115
 11  84   8 106 102     114   2  68  80  26      32  52  61  49  96      67  69  95   6  68      63 100  60 123  17
110  53  86  60  13      14 104  96  52  16      41   6  50  76 103      27  23  65  89 111     101  92  24  44  72
 98  58   6  41  66      50  85  92  21  97      88  73 112  80  25      29  90  82 116  33      38  77  43  54  78
 25  23  69  75  83      29  63  20  95  78      84  93  58  15  53     109 120  45   9  83      42   6  69  70  57
```

```
Pop 100, 200 Iterations
  Duration: 9.93 seconds
  Final Objective Function Value: -2165

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 27  65 100  21 103     108  26  48  89  75      11  16 116  81  77      88  90  37  15  74       8 111   2 107  56
108  73  81  33  15     104  63  61  19  64      29  57  23  92 108      43  69  53  98  59      62  80 101  75   9
120  80  97 121  93       2  94  13  16 101     109  26  61 118  19      64  61  31  26  84      36  55 121  55  50
 49  50  41  59  56      32  80 122  86  26     110 103  40  21  20      14  18  48 100 125     104   5  72  12  76
  6  43  52  81  62      66  53  69  87  51      82  93  16  32  91      95  80 123   3  17      73  45  35  63 100

Pop 100, 250 Iterations
  Duration: 12.71 seconds
  Final Objective Function Value: -2119

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
117  28  35  98  51      16 100  74  45  89      69  91  74  19  51      32  45 115  23  87      73  60  78  94  27
 97  52  13 109  72      91  24 119  82  11      70  82  33 119  35      28  71 123  40  61      26  88  52 106  99
  2 125  40  62  92     108  56 106  60  21       3  76  80 120  36      82  29  18  20 109     108  48  34  56  53
 66  81  94  17  29      50  36   7  95 125     112  34 117   2  92      58  95  59 122  13      17 109  25  46 117
 34  31 119  70  77      49 118   9  18 102      56  63  24  72 101     107  10   7 114  82      70  31 111  64  51
```

```
Pop 200, 150 Iterations
  Duration: 16.51 seconds
  Final Objective Function Value: -2204

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 50  40  92  84  59      53  32  44 117  96     81  84   9  48  85     42 119  93  13  79    102 106  18  80   2
 45  87  93  55 113     109  73  72  59   8     58  49  34  44 121     83  74   3  99  44     11 105 115  58  40
 57 119  21 107  35     121  34  89  17  15     65  89  66  57  47     23   6 101  62 104     59  64  33  68 120
  5  30  48  98  99      12  38  68  66  63    117  17  96  69   8    118  56  39  53  59     63 112  76  14  61
123  42  71  19  10      34 112  58  21 103     89  75 124  21  10     67  50  51 107  73     98  16  12 113  72

Pop 200, 200 Iterations
  Duration: 20.39 seconds
  Final Objective Function Value: -1862

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 86  16  77  83  22      48  89  70  63  95     58  11 110  30 114     67 109  37  29  84     52 105  21 108   3
 42  94  48  66  87     116  79  12  84  49    118  69  10  61  76     47  17  97 109  28      1  86 119  49  89
 71  22  43  96  91      77 102  44  40  51     16 125  75  68  44     53  18  91  65  66     88   8  63  54 116
 47 105   6  66  53      51  21 103  75 125     35  67  85  79  72    104  48 102  27  38     71  92  50  71  34
 66  67 111  34  68      37  91  85  56  38     51  20  48  87  23     64 121  19  13 101    103   9  77 107  22
```

```
Pop 200, 250 Iterations
  Duration: 26.61 seconds
  Final Objective Function Value: -1947

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 63  42  63  67  71      54  70 107  58  24       6  97  23  58 115      93   8 100  33  86     120  95  17  63 101
110  76   3  30  77      62  40  38  53  43      87  78 103  22  40      62  25  98  57 123      28  81  66  91  24
104  64  73  26  46      74  23  56  84 115      33  99  60  29  51       9  75  45  89  82      77  84  63 103   4
 18 103  82  99  94      35  90 114  60  28      79  12  63 113  27      81  74  18 110  39      98   1  44  37 125
  4  81  79  96  29      81 111  13  84  98     112  30  70  79  24      57 122  53  16  63      68  13 107  35  46

Pop 300, 150 Iterations
  Duration: 24.75 seconds
  Final Objective Function Value: -2012

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
  5 108 111  17  53     102  11  34 107  46     107  48  45  52  50      48  70  40 123  34      85  68 102   7  88
 57  90  26  14  93     103  76   1  90  66      10  62 105  75  72     104  37 110   1  57      33  14 117 114  35
 75  21  50  87  78      31 100  62  65  34      51  27  60  21 108     118   6 109  69 100      32 108   3  84  88
 84  27  92 110  51      14  45 109  48  88      95 101  74  43  71      31 107  23  39  53      99  30  15  78 107
 90  66  68  33  63      69  70  91  38  82      24  90  73 119  16      46  65   5 120  82      79  55  89  13  92
```

```
Pop 300, 200 Iterations
  Duration: 33.18 seconds
  Final Objective Function Value: -1904

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 67  52  34  59  66      14  64  99  20  26      99  87  85  21  27      66  39  62 125  74      90  53  32  39 118
 31  85  47  37 112     123  79  78  55  30      53  29  80 106  81      52  92  35  32  98     113  37 110  55  11
109  88  76 110  39      16  97  80 119  12      83  30  57  26 100      96  65  48  59  49      64  23  68   6  79
  9  42  46  90  89      51  37  50  45 110      79  42 112  78   3      51 110 107  16  12      28  91   4 114  97
 91  53 113  11  10     125  29  12  84 102      19 123  50  61 113      64  41  63  82  67      22  70  80 116  14

Pop 300, 250 Iterations
  Duration: 40.59 seconds
  Final Objective Function Value: -1678

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
102  48  56 102   8      35  79  39  57  93      90  78 111  36  85      34  95  16 124  43      82  17  76  22  97
 90  51  78   1  73      72  60  73  98  19      45  24  53 123  57      91 121  61  43  73      27  87  54  30 112
 20  32  85 121 119      67  81  47   2 116      35  93  18  62  28     122  77  81   3  43      67  65  77 109  15
 18  90  70  50  76      87  38  89 117   4      80  11  33 117  47      29 101  37  50 125     116  73  96  20  21
117  73  36  59  26      58  53  69  61  43      66 102 100   1  38       5   2 110 102 119      53  56  12  83  93
```

Pada percobaan kedua, dipilih ukuran populasi sebanyak 100, 200, dan 300 individu dengan jumlah iterasi sebanyak 150, 200, dan 250 kali. Pada populasi 100, divariasikan iterasi sebanyak 150, 200, dan 250 kali, begitu pula dengan individu 200 dan 300. Waktu atau durasi proses untuk jumlah individu yang sama akan semakin besar semakin banyak jumlah iterasinya. Untuk nilai akhirnya cenderung lebih baik semakin banyak individu atau iterasi yang dilakukan. Misal pada populasi 200 individu, iterasi 150 kali memerlukan waktu 16,51 detik dengan nilai akhir -2661 sedangkan pada 250 iterasi memerlukan 22,04 detik dengan nilai akhir -1947. Begitu pula untuk jumlah iterasi yang sama, semakin banyak individu semakin besar juga waktu yang diperlukan dan semakin mendekati lebih baik pula nilai akhirnya.

```
+==================== GENETIC ALGORITHM TRIAL 3 ====================+

Pop 100, 150 Iterations
  Duration: 6.99 seconds
  Final Objective Function Value: -2208

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 35  60 125  48  39     109  51  73  80  89      89  35  36  48  50      64  62  14 109  47      91 100  51  35  74
114  64  12  11 111      52  17  69 107  83      45 117 103  52   5       9  71  86  57 104      97  37  51 122   3
104  52  96  19  85      14 121  29  65  25      53  56  58  81 101      78  52  59 107  22      35 115  76  46  64
  4  62  45 106  36      94  22 101  67  56     106  72  39  29  65      48  99  61  49 102      13  39  59  24  67
 97  86  44  87  90      59 117  47  35  76      96  69  28  95  33      70  43  94  30  75       7  17  85 120  68
```

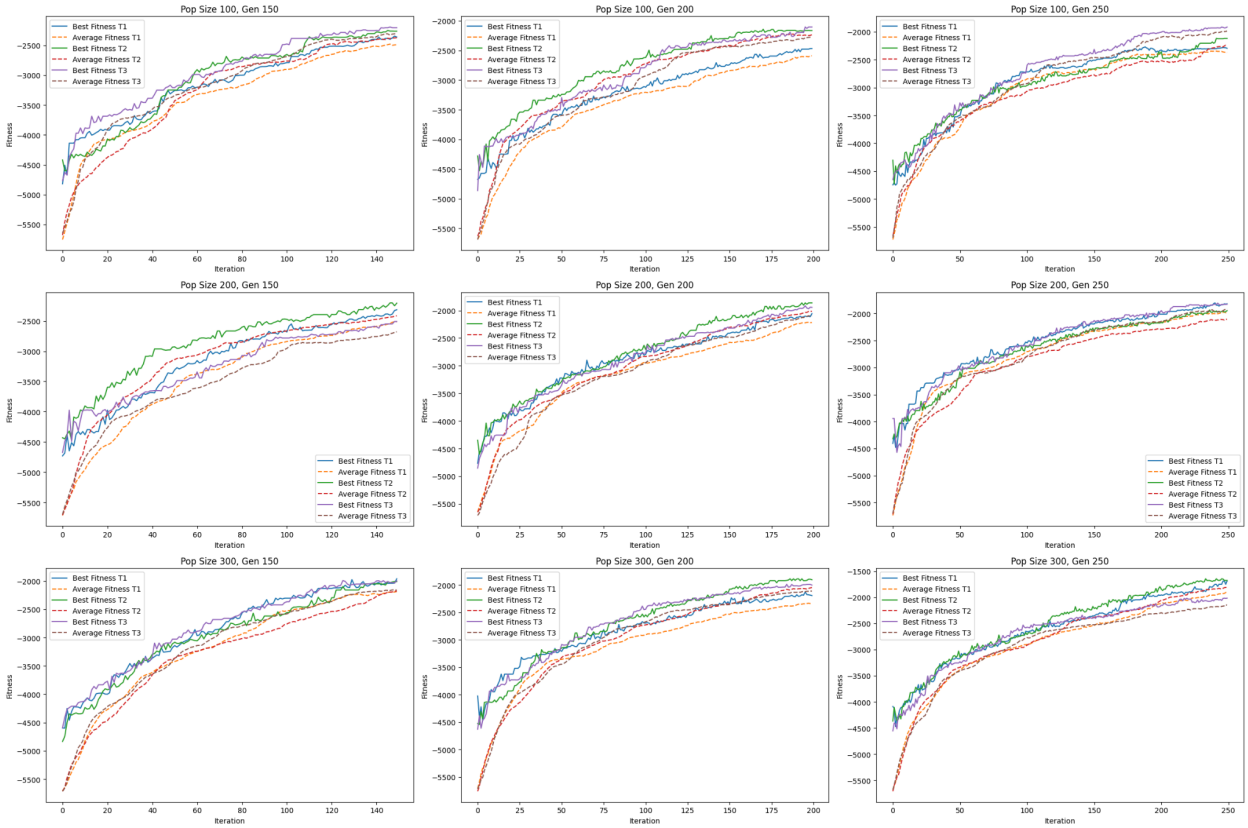
```
Pop 100, 200 Iterations
  Duration: 10.53 seconds
  Final Objective Function Value: -2105

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 97  86  37  11  54     107   3  61 116  87      17 116  31  21  99      25  35  87  84  92      57  69 100  83  27
122  83  50  92   7      48  33  14  70 111      61  74  80  54  38      44  86 113   1  96       7  54  81  90  87
125  88  26  56  31      39  72  96  34 113      43  73  72  46  63      22  64  54  90  70     105   4  97  40  87
 15  38 101  62 116      99  46  76  40  28     122  33  81 117  32      59  95  59  78   5      24  97  11   8 114
103  16  55  59  59      27 111  70  44  30      74  15  58  77  92     121  27  37  45  71      47 109  76 102  14

Pop 100, 250 Iterations
  Duration: 14.22 seconds
  Final Objective Function Value: -1914

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 98 125  68   3  40       4  79 106  64  68      57  12  37  96  87      94  56  47  28  91      75  83  19 105  36
 79  70 113  57   2      83  37  38  72 112     114 111   1   6  43      36  53  62  79  90      29 100  73  82  60
106  23   9 124  52     111  36 105  51  70       3 103  62  84  61      39  99 109  45  11      70   8  50  49 101
 43  87  18  67 103      26 118  28  89   3      24  65 109  79  25     121  27  23  38 106      26  41 113  28  85
 21  25  94  57  82      85  62   3  28  56     111   2  87  30  95      83  72  68 119  23     108  80  48  76  57
```

```
Pop 200, 150 Iterations
  Duration: 15.59 seconds
  Final Objective Function Value: -2508

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
112  42  50   5  95       3  71 121 103  89      51  77  41 110  34     112   3  93   7 116      45  79  43  82  22
 73   4  14  80 105      90  65  93  37  84      46  87  18  56  21      23  84  61  93  53      80  74 125  39  33
109  94  21  57  80     101  19 101  15  39      64  81  58 106  56       4  90  78  96  32      11  95  17  97 103
  3  90  86  81  34      69  56  52 102  28      24  57  86  68  99     123 102  12  15  49     104  14  83  71  94
 20  25 122  78  96      63 115  18  46  30     124  26 108  20  55      50  60  18 114  31      30  69  35  70 109

Pop 200, 200 Iterations
  Duration: 21.74 seconds
  Final Objective Function Value: -1943

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 26 109  29 121 104      50  90  88  27  69      16  32  42  51  67     109  64 116  30  26      71  46   2 125  67
 48 107 120  25  15     111  49   7  89  65     110  73  90  68  34      41  18  15 119 105      10  76  81   9  97
 97   9  68  38  85      12  20  75  77  69      72 100  65  44 123      98 117  38  65  20      47  68  85  88  11
 65   3  86  77  67      93  78  32  95  46      40  35  25 106  54      28  58 120  37  65      87 104  25   7  76
 82  62  47  53  65      56  68  83  60  64      48  77  93  69  51      77  50  29  72  82      49  21  86 107  45
```

```
Pop 200, 250 Iterations
  Duration: 26.21 seconds
  Final Objective Function Value: -1828

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

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

Pop 300, 150 Iterations
  Duration: 24.55 seconds
  Final Objective Function Value: -1997

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

Final State - Best Individual
Slice 1                 Slice 2                 Slice 3                 Slice 4                 Slice 5
 50  82  96  23 104     68 103   4  43  72     81  49 115  86  13     27  75  92  80  23    107   9  39  34  89
 80  98  52  59  16    121  14  77 114  32      2  91  97  63  73     90  66  57  38   4     42  60  76  48  96
 63   6  57 116  74     29 111  67  49   9    122  37  40  23 112     32  53 113 109  87     65 115  49  95  28
 44  33  39  55 101     59  17 115  62  65     10 125   7 120  44     85  55  36  67 103     99  88  58  33   1
123 100  52  28  40     34  92  41  49  95     84  25  72 108  26     70  59  68  16  83      8  22  98 116 102
```

```
Pop 300, 200 Iterations
  Duration: 31.61 seconds
  Final Objective Function Value: -2001

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
123  46  72  98  74      67  60 115  70   2      46   2  64  60  60      45 106  54  18 103     39  88  13  84  98
 66  18  83  31 105      27  53  37  90  80      55  78  70  40  53      91  85  63  60  26     86  77  55  41  74
 48  79  95  87  49     100  65  39  21 113      29 115  38  33  93      47  28  92 113  32     56  52  51  52  62
 15  55  26  30  16       4  77  67 103 101     104 107 120  41  56     105   5  58  11 121     67  87  46 118  25
 62 121  17  52  58      84  77  56  43  32      82  12  37 116  68      34 101 108  30  59     76  14  79  97  62

Pop 300, 250 Iterations
  Duration: 40.60 seconds
  Final Objective Function Value: -2017

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

Final State - Best Individual
Slice 1                  Slice 2                  Slice 3                  Slice 4                  Slice 5
 12 109  49  27  63      70  37  45 114  22     109  25 105  72  23      55  82  98  22  92      60  70  31  83  64
105  88  36  31  90      32  81  80  40  82       3  36 103  22  61     107   9  13 125  35      43  96  32  73  88
  6  67 111  91  28      99  48   7  25 124      91  45  99  41  37     111 117  63  58  33       9  38  46 116 100
 97  14  67 100  32       2 120  83  88   5      48  93  28  12 121      47  31  71  83 113     108  76 106   8  42
 93  34  63  66  80      61  30 103  92  76      80  88  30  63  53      17 119  56  30 102      66  40  79  90  19
```

Pada percobaan ketiga, dipilih ukuran populasi sebanyak 100, 200, dan 300 individu dengan jumlah iterasi sebanyak 150, 200, dan 250 kali. Pada populasi 100, divariasikan iterasi sebanyak 150, 200, dan 250 kali, begitu pula dengan individu 200 dan 300. Waktu atau durasi proses untuk jumlah individu yang sama akan semakin besar semakin banyak jumlah iterasinya. Untuk nilai akhirnya cenderung lebih baik semakin banyak individu atau iterasi yang dilakukan. Misal pada populasi 100 individu, iterasi 150 kali memerlukan waktu 6,99 detik dengan nilai akhir -2208 sedangkan pada 250 iterasi memerlukan 14,22 detik dengan nilai akhir -1914. Begitu pula untuk jumlah iterasi yang sama, semakin banyak individu semakin besar juga waktu yang diperlukan dan semakin mendekati lebih baik pula nilai akhirnya.



- **Analisis**

Berikut merupakan hasil analisis terhadap hasil eksperimen yang telah dilakukan.

- **Seberapa dekat tiap-tiap algoritma bisa mendekati global optima**

Pada *Steepest Ascent Hill Climbing*, algoritma masih terjebak di *local optimum* dengan nilai akhir yang negatif. Algoritma ini sulit untuk mencapai global optimum karena berdasarkan cara kerjanya tidak memiliki mekanisme yang memungkinkan untuk keluar dari lokal optimum. Jika sudah terjebak di *local optimum*, algoritma akan sulit mencapai lokal optimum yang lebih baik ataupun global optimum.

Pada *Hill Climbing with Sideways Move*, algoritma masih terjebak di *local optimum* dengan nilai akhir yang negatif. Algoritma ini sulit untuk mencapai global optimum karena berdasarkan cara kerjanya tidak memiliki mekanisme

yang memungkinkan untuk keluar dari lokal optimum. Walaupun algoritma memungkinkan untuk tetap bergerak saat nilai sama (tidak lebih buruk atau tidak lebih baik), jika sudah terjebak di *local optimum*, algoritma tetap akan sulit mencapai lokal optimum yang lebih baik ataupun global optimum.

Pada *Random Restart Hill Climbing*, algoritma tersebut memiliki mekanisme yang membuat algoritma dapat keluar dari jebakan lokal optimum dengan melakukan restart dan mendekati nilai global optimum. Namun, pada percobaan ini, nilai akhir algoritma *Random Restart Hill Climbing* masih bernilai negatif dan cukup jauh dari solusi global optimum.

Pada *Simulated Annealing*, algoritma berhasil mendekati solusi *global optimum* (nilai akhir fungsi objektifnya 0) dengan mencapai nilai akhir -60 dibanding algoritma *local search* yang lain. Hal tersebut terjadi karena cara kerja *Simulated Annealing* memungkinkan algoritma untuk keluar dari jebakan lokal optimum untuk mencari solusi yang lebih baik lagi dengan menggunakan temperatur. Walaupun begitu, semakin banyak iterasi, temperatur akan semakin turun hingga iterasi dihentikan.

Pada *Genetic Algorithm*, jumlah variasi populasi dan jumlah iterasi mempengaruhi hasil nilai yang dapat dicapai algoritma. Jumlah populasi atau iterasi yang semakin besar memungkinkan algoritma untuk mencapai nilai akhir fungsi objektif yang lebih baik lagi dan mendekati solusi *global optimum*.

- **Perbandingan hasil pencarian tiap-tiap algoritma dengan algoritma local search yang lain**

Pada Steepest Ascent Hill Climbing, hasil pencariannya masih kurang optimal jika dibandingkan dengan algoritma search yang lainnya, terutama algoritma yang cara kerjanya memungkinkan untuk keluar dari jebakan lokal optimum.

Pada Hill Climbing with Sideways Move, tidak jauh berbeda dengan Steepest Ascent Hill Climbing. Namun, pada algoritma ini, pencarian dapat terus

dilakukan bahkan jika nilai neighbor sama sehingga memiliki peluang ynag lebih baik untuk mendekati global optimum jika dibandingkan dengan Steepest Ascent Hill Climbing. Walaupun begitu, jika dibandingkan dengan algoritma lain, Steepest Ascent Hill Climbing masih menghasilkan solusi yang kurang optimum.

Pada Stochastic Hill Climbing, hasil akhir yang didapatkan lebih baik jika dibandingkan dengan Hill Climbing with Sideways Move ataupun Hill Climbing with Sideways Move namun lebih buruk jika dibandingkan dengan Simulated Annealing.

Pada Random Restart Hill Climbing, hasil akhirnya juga lebih baik jika dibandingkan dengan Hill Climbing with Sideways Move ataupun Hill Climbing with Sideways Move. Walaupun begitu, hasil Random Restart Hill Climbing masih belum bisa lebih baik dibandingkan jika menggunakan algoritma Simulated Annealing.

Pada Simulated Annealing, hasil nilai akhir fungsi objektifnya mencapai nilai terbaik dan yang paling mendekati global optimum jika dibandingkan dengan algoritma search yang lainnya.

Pada Genetic Algorithm, hasilnya masih kurang optimal dan jauh dari Simulated Annealing maupun solusi global optimum. Hal tersebut juga dipengaruhi oleh pengaturan jumlah populasi individu dan iterasi yang dilakukan.

- **Perbandingan durasi proses pencarian tiap algoritma relatif terhadap algoritma lainnya**

Pada *Steepest Ascent Hill Climbing*, durasi atau waktu yang dibutuhkan selama proses pencarian termasuk yang paling cepat dengan kisaran waktu sekitar 3-5 detik. Hal tersebut juga berhubungan dengan kekurangan algoritma Steepest Ascent Hill Climbing yang sangat mudah terjebak pada solusi lokal optimum sehingga pencarian lebih cepat dihentikan.

Pada *Hill Climbing with Sideways Move*, durasi atau waktu yang dibutuhkan selama proses pencarian tidak jauh dengan Steepest Ascent Hill Climbing namun tetap lebih lama dengan durasi sekitar 5-7 detik. Hal tersebut disebabkan oleh algoritma ini tetap berjalan jika ditemukan nilai yang sama walaupun tetap mudah untuk terjebak pada *local optimum*.

Pada *Stochastic Hill Climbing*, durasi atau waktu yang dibutuhkan lebih lama jika dibandingkan dengan *Hill Climbing with Sideways Move* ataupun *Hill Climbing with Sideways Move*. Durasi yang dibutuhkan algoritma *Stochastic Hill Climbing* berkisar sekitar 53-55 detik. Durasi yang cukup lama ini terjadi karena algoritma melakukan iterasi dan pencarian solusi yang lebih luas dibandingkan dengan *Hill Climbing with Sideways Move* ataupun *Hill Climbing with Sideways Move.*

Pada *Random Restart Hill Climbing*, durasi atau waktu yang dibutuhkan lebih lama jika dibandingkan dengan *Hill Climbing with Sideways Move* ataupun *Hill Climbing with Sideways Move*. Durasi yang dibutuhkan algoritma *Random Restart Hill Climbing* berkisar sekitar 22-26 detik. Namun, durasi yang dibutuhkan algoritma ini juga tergantung dengan jumlah restart yang dilakukan selama proses pencarian.

Pada *Simulated Annealing*, durasi atau waktu yang dibutuhkan berkisar pada 54 detik. Durasi tersebut cukup lama dibandingkan dengan *Hill Climbing with Sideways Move, Hill Climbing with Sideways Move,* ataupun *Random Restart Hill Climbing* karena algoritma *Simulated Annealing* melibatkan temperatur dalam mekanisme pencariannya. Temperatur tersebut walaupun memiliki dampak yang baik dengan memungkinkan pencarian solusi yang lebih optimum, namun mengakibatkan durasi yang lama karena adanya pendinginan temperatur tersebut.

Pada *Genetic Algorithm*, durasi atau waktu yang dibutuhkan bergantung pada banyaknya populasi individu dan iterasi yang dilakukan.

- **Konsistensi hasil akhir yang didapatkan dari tiap-tiap eksperimen yang dilakukan**

  Pada *Steepest Ascent Hill Climbing*, algoritma menghasilkan nilai yang cukup konsisten karena pencarian solusi hanya dilakukan di sekitar *initial state*.

  Pada *Hill Climbing with Sideways Move*, algoritma menghasilkan nilai yang konsisten karena pencarian solusi hanya dilakukan di sekitar *initial state*.

  Pada *Stochastic Hill Climbing*, algoritma percobaan kali ini menghasilkan nilai yang cukup konsisten walaupun ada nilai yang cukup berbeda dengan percobaan lainnya. Hal tersebut terjadi karena Stochastic Hill Climbing melakukan pencarian secara acak dan tidak hanya mencari nilai atau solusi di sekitar *initial state* saja seperti Steepest Ascent Hill Climbing ataupun Hill Climbing with Sideways Move.

  Pada *Random Restart Hill Climbing*, algoritma ini menghasilkan nilai yang beragam, ada yang konsisten dengan percobaan lain namun ada yang mendapat nilai akhir fungsi objektif yang cukup jauh berbeda juga.. Hal tersebut terjadi karena Random Restart Hill Climbing melakukan pencarian secara acak dan tidak hanya mencari nilai atau solusi di sekitar *initial state* saja seperti Steepest Ascent Hill Climbing ataupun Hill Climbing with Sideways Move. Konsistensi Random Restart Hill Climbing juga tergantung *initial state* dan nilai *state* acaknya.

  Pada *Simulated Annealing*, algoritma ini menghasilkan nilai yang cukup konsisten. Namun, hasil tersebut juga tergantung pada hasil pendinginan temperatur selama proses pencarian berlangsung.

  Pada *Genetic Algorithm*, algoritma ini menghasilkan nilai yang tidak sepenuhnya konsisten. Hal ini terlihat dari kondisi di mana hasil akhir populasi lebih besar atau iterasi yang lebih banyak tidak menghasilkan perbaikan yang sebanding atau malah cukup lebih buruk meskipun pada eksperimen terdapat peningkatan jumlah iterasi atau populasi yang memberikan perbaikan pada nilai

objective function. Namun hal tersebut tidak selalu berlaku terutama jika jumlah populasi atau iterasinya tidak terlalu signifikan perbedaannya. Konsistensi juga terlihat dari durasi waktu di mana populasi dan iterasi yang lebih besar pasti menghabiskan waktu lebih lama walaupun dampaknya terhadap kualitas hasil tidak selalu sebanding. Populasi atau iterasi yang lebih kecil terkadang juga menghasilkan nilai *objective function* yang hampir sama atau bahkan lebih baik yang berarti optimasi tidak senantiasa memberikan peningkatan linier dan tak menjamin hasil terbaik hanya dengan menambah populasi atau iterasi.

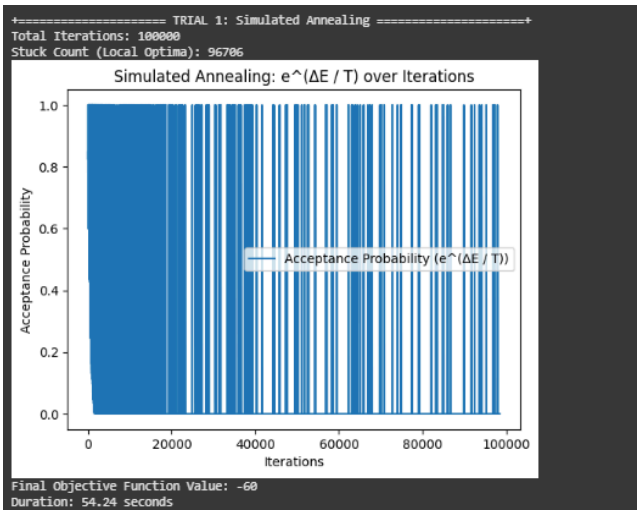
- **Pengaruh banyak iterasi dan jumlah populasi terhadap hasil akhir pencarian pada *Genetic Algorithm***

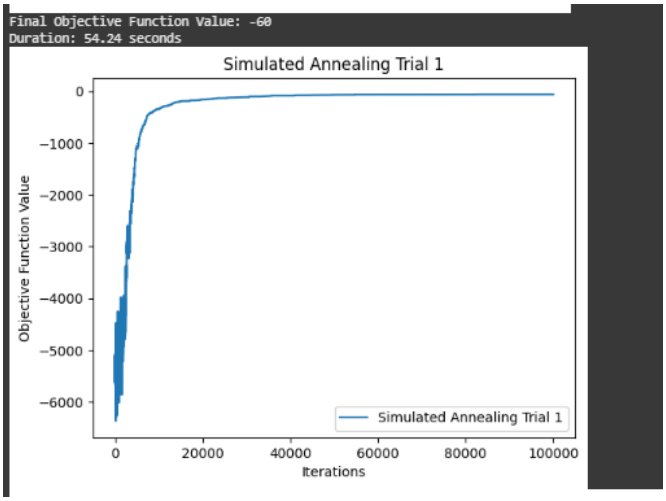
  Pada *Genetic Algorithm*, banyak iterasi dan jumlah populasi sangat berpengaruh terhadap hasil akhir pencarian. Semakin banyak jumlah iterasi, maka akan semakin besar juga kemungkinan algoritma ini untuk mendapatkan nilai akhir yang mendekati global optimum karena pada proses seleksi individu, *crossover*, dan mutasi dapat dilakukan lebih banyak lagi. Jumlah populasi juga berpengaruh dengan semakin besar jumlah populasi, maka semakin banyak juga variasi yang dapat digunakan dalam proses pencarian sehingga dapat meminimalisir algoritma untuk terjebak pada solusi *local optimum*. Namun, selain berdampak baik, jumlah populasi dan iterasi yang tinggi juga dapat mempengaruhi waktu atau durasi yang dibutuhkan pada proses pencarian, yaitu durasi pencarian menjadi semakin lama.

KESIMPULAN DAN SARAN

A. Kesimpulan

Berdasarkan hasil eksperimen dan analisis yang telah dilakukan menggunakan enam jenis algoritma pencarian (Steepest Ascent Hill Climbing, Hill Climbing with Sideways Move, Stochastic Hill Climbing, Random Restart Hill Climbing, Simulated Annealing, dan Genetic Algorithm), diperoleh hasil bahwa pencarian solusi *diagonal magic cube* dengan algoritma *local search* yang terbaik ialah *Simulated Annealing*. Hal ini dikarenakan setelah program dijalankan, kami mendapatkan nilai *final objective function* sebesar -60 dengan durasi selama 54,24 detik pada percobaan pertama (*trial* 1). Hasil dari algoritma *Simulated Annealing* ini adalah yang paling mendekati nilai nol (0) jika dibandingkan dengan percobaan yang dilakukan dengan algoritma *local search* lainnya sehingga percobaan pertama dengan algoritma *Simulated Annealing* tersebut merupakan hasil yang terbaik dibandingkan jenis algoritma *local search* lainnya yang telah diperoleh.





B. Saran

Saran dari pengerjaan laporan ini adalah sebaiknya dikerjakan dengan mencicilnya lebih awal supaya dapat memiliki waktu untuk mengimplementasikan bonus berupa *video player*. Selain itu, disarankan agar lebih mampu mengelola waktu lebih baik lagi ke depannya dalam pengerjaan tugas besar ini. Kemudian, disarankan juga

memanfaatkan algoritma *local search* berupa *Simulated Annealing* untuk mencari solusi *Diagonal Magic Cube* yang terbaik.

PEMBAGIAN TUGAS TIAP ANGGOTA KELOMPOK

NIM	Nama	Pembagian Tugas
18222002	Yasra Zhafirah	Membuat program source code, README.md
18222030	Vini Putiasa	Membuat fungsi objektif, mengerjakan pembahasan implementasi algoritma, pembahasan hasil eksperimen dan analisis pertanyaan, kesimpulan.
18222031	Benedicta Eryka Santosa	Mengerjakan pembahasan
18222090	Kerlyn Deslia Andeskar	Mengerjakan deskripsi persoalan, pembahasan, kesimpulan & saran, fungsi ascii

REFERENSI

Russell, S., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson Education.

Trump, W. (2003, June 19). *The Successful Search for the Smallest Perfect Magic Cube*. Retrieved September 30, 2024, from https://www.trump.de/magic-squares/magic-cubes/cubes-1.html

Anonymous. (n.d.). *Features of the Magic Cube*. magischvierkant.com. Retrieved September 28, 2024, from https://www.magischvierkant.com/three-dimensional-eng/magic-features/