TUGAS BESAR 1

Pencarian Solusi Diagonal Magic Cube dengan Local Search

IF3070

Dasar Inteligensi Buatan

Disusun oleh:

Audra Zelvania Putri Harjanto / 18222106

Rizqi Andhika Pratama / 18222118

Sekar Anindita Nurjadini / 18222125

Khayla Belva Annandira / 18222138

Institut Teknologi Bandung

2024

Daftar Isi

Deskripsi Persoalan

Tugas Besar 1 ini merupakan bagian dari mata kuliah Dasar Intelegensi Artifisial yang bertujuan untuk memberikan pemahaman mengenai cara mengimplementasikan algoritma *local search* dalam mencari solusi permasalahan optimasi pada diagonal magic cube berukuran 5x5x5.

Diagonal magic cube merupakan kubus yang disusun dari angka-angka 1 hingga n^3 pengulangan, dengan panjang sisi kubus adalah n. Setiap baris, kolom, dan tiang dari kubus tersebut harus memiliki jumlah angka yang sama dengan magic number yang ditentukan. Selain itu, jumlah angka pada diagonal ruang serta diagonal pada potongan bidang kubus juga harus sesuai dengan magic number tersebut.

Kami diharapkan untuk mengimplementasikan algoritma *local search* dengan beberapa pendekatan, yaitu Steepest Ascent Hill-climbing, Hill-climbing with Sideways Move, Random Restart Hill-climbing, Stochastic Hill-climbing, Simulated Annealing, dan Genetic Algorithm. Setiap algoritma dilakukan Pengujian dan dicatat hasilnya, termasuk *state* awal dan akhir dari kubus, nilai *objective function*, serta durasi proses pencarian. Untuk algoritma tertentu seperti Steepest Ascent Hill-Climbing dan Stochastic Hill-Climbing, perlu dicatat banyaknya iterasi hingga pencarian berhenti, sementara untuk Simulated Annealing, frekuensi kejadian 'stuck' di local optima juga perlu diamati. Khusus untuk Genetic Algorithm, dilakukan uji coba dengan variasi jumlah populasi dan iterasi untuk melihat pengaruhnya terhadap hasil pencarian.

Selain itu, hasil dari Pengujian ini perlu divisualisasikan untuk menunjukkan *state* awal dan akhir dari kubus, serta perubahan nilai *objective function* selama iterasi. Hal tersebut dilakukan untuk memberikan pengetahuan mengenai efektivitas dan efisiensi berbagai algoritma *local search*, serta faktor-faktor yang mempengaruhi kinerja pencarian solusi diagonal magic cube.

Pembahasan

1. Pemilihan Objective Function

1.1 Penjelasan

Objective function adalah fungsi yang digunakan untuk mengevaluasi seberapa baik suatu solusi dalam sebuah masalah. Dalam konteks tugas Diagonal Magic Cube, objective function berfungsi untuk mengukur seberapa dekat susunan angka di dalam kubus 5x5x5 dengan kondisi ideal, yaitu ketika semua baris, kolom, diagonal, dan bidang memiliki jumlah angka yang sama dengan magic number. Berikut adalah cara yang kami gunakan untuk menentukan objective function dari persoalan ini:

- **Menentukan magic number**. Magic number dalam konteks magic cube adalah jumlah konstan yang harus dicapai oleh semua baris, kolom, tiang, dan diagonal dalam kubus tersebut. Magic cube terdiri dari bilangan bulat yang diatur dalam susunan $n \times n \times n$, dan magic number memastikan bahwa jumlah dari angka-angka di setiap dimensi kubus (baris, kolom, tiang, dan diagonal) selalu sama. Magic number pada magic cube tiga dimensi dapat dihitung dengan rumus:

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

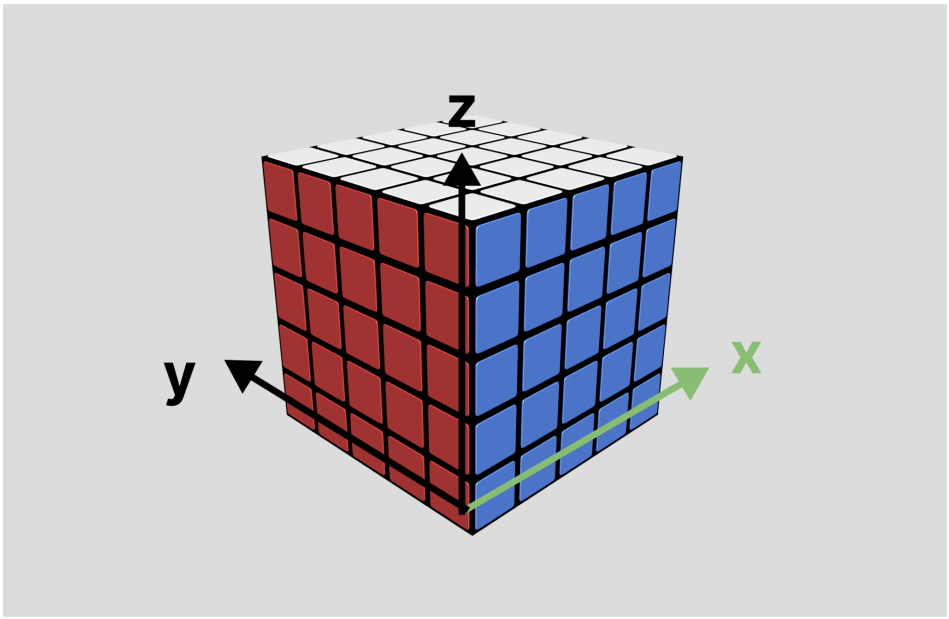
Kemudian, pada tugas ini dipaparkan sebuah magic cube 5 x 5 x 5 yang memiliki arti bahwa nilai n adalah 5. Oleh karena itu, kita dapat menyimpulkan bahwa magic number dari magic cube 5 x 5 x 5 ini adalah sebagai berikut:

$$Magic\ Number\ = \frac{5\,(5^3+1)}{2}\ = 315$$

Kemudian, kita dapat merumuskan objective function dengan magic number yang sudah kita dapatkan dengan memperhatikan syarat-syarat dari magic cube.

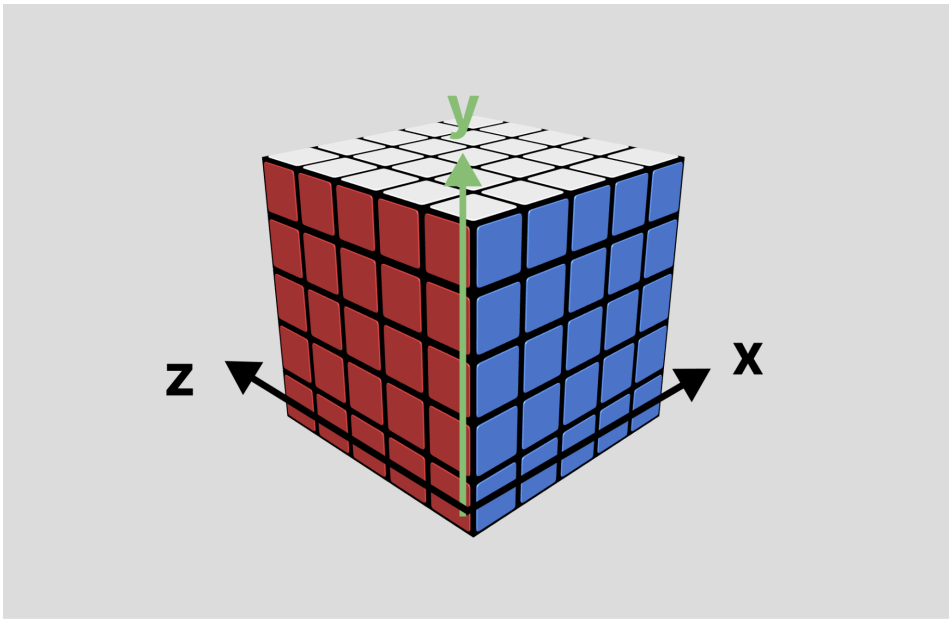
- **Evaluasi penjumlahan angka-angka pada setiap bagian dari syarat magic cube**. Objective function akan menghitung jumlah angka-angka pada:

  ❖ Setiap baris (horizontal)



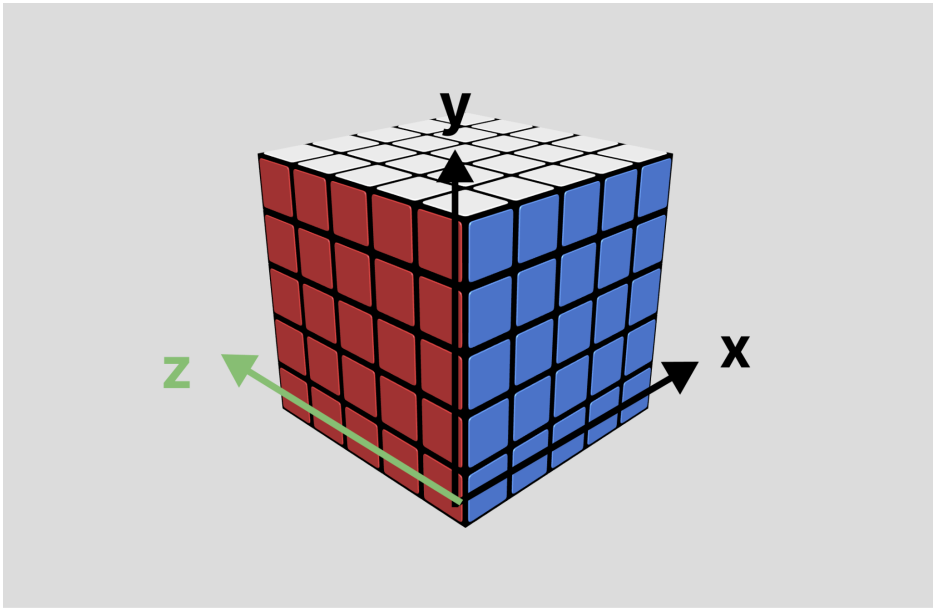
Gambar 1.1. Baris pada Cube (sumbu-x)

  ❖ Setiap kolom (vertikal)



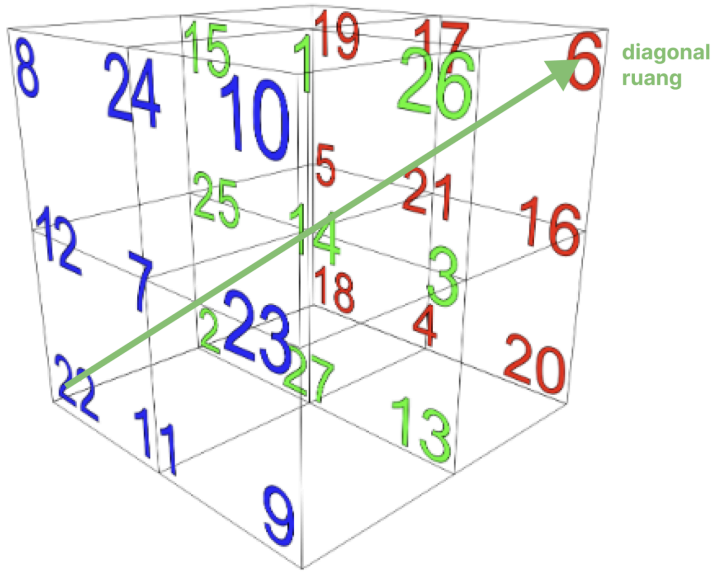
Gambar 1.2. Kolom pada Cube (sumbu-y)

- Setiap tiang (z-axis)



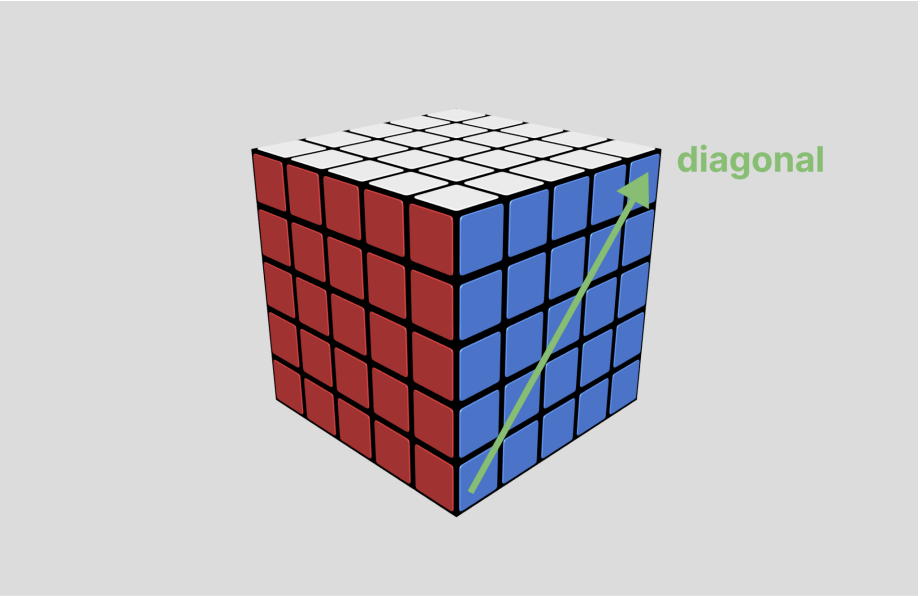
Gambar 1.3. Tiang pada Cube (sumbu-z)

- Seluruh diagonal ruang (melintasi kubus 3D)



Gambar 1.4. Diagonal Ruang pada Cube

- Seluruh diagonal pada suatu potongan bidang dari kubus



Gambar 1.5. Diagonal bidang pada Cube

- **Menghitung selisih**. Setelah menghitung jumlah setiap bagian, objective function akan membandingkan jumlah tersebut dengan magic number. Jika jumlahnya benar, maka selisihnya 0. Jika tidak benar, objective function akan menghitung seberapa besar perbedaannya. Berdasarkan pembahasan ini, objective function dapat dirumuskan sebagai berikut:

$$f(s) = \Sigma |M - S_{row}| + \Sigma |M - S_{col}| + \Sigma |M - S_{pillar}| + \Sigma |M - S_{diagonal}| + \Sigma |M - S_{slice}|$$

dengan:

- $f(s)$ adalah nilai dari objective function untuk state s
- M adalah magic number dengan nilai 315
- S_{row}, S_{col}, S_{pillar}, $S_{diagonal}$, dan S_{slice} masing-masing adalah jumlah dari angka-angka pada baris, kolom, tiang, diagonal ruang, dan potongan bidang.

Objective function ini akan memberikan nilai error yang menunjukkan seberapa jauh kubus tersebut dari kondisi magic cube ideal. Semakin kecil nilai objective function, semakin dekat state dengan kondisi yang diinginkan. Jika nilai objective function mencapai 0, berarti state yang ditemukan sudah benar, di mana setiap baris, kolom, diagonal, dan

bidang pada magic cube menjumlahkan angka-angka yang sama dengan magic number.

1.2 Implementasi

Implementasi dari *objective function* pada masalah Diagonal Magic Cube 5x5x5 bertujuan untuk mengukur sejauh mana konfigurasi angka-angka dalam kubus tersebut mendekati kondisi ideal dari sebuah magic cube.

Untuk penerapan *objective function* terdapat pada file cube.py

Tabel 1.2.1. Fungsi Objective Magic Cube

Nama Fungsi	objective_function(self)
Deskripsi Fungsi	Fungsi ini berguna untuk menghitung nilai objektif dari sebuah cube dan melihat seberapa jauh state cube dengan state ideal (nilai objektif = 0)
Kode	`def objective_function(self):` `  # Mengambil magic number` `  magic_number = self.magic_number` `  total_difference = 0 # Nilai objektif paling optimal` `  # Perbedaan jumlah di setiap baris` `  for layer in self.cube:` `  for row in layer:` `  row_sum = np.sum(row)` `  total_difference += abs(magic_number - row_sum)` `  # Perbedaan jumlah di setiap kolom` `  for layer in self.cube:` `  for col in range(self.n):` `  col_sum = np.sum(layer[:, col])` `  total_difference += abs(magic_number - col_sum)`

```
        # Perbedaan jumlah di setiap tiang
(melintasi layer)
        for row in range(self.n):
            for col in range(self.n):
                pillar_sum =
np.sum(self.cube[:, row, col])
                total_difference +=
abs(magic_number - pillar_sum)

        # Perbedaan jumlah di setiap diagonal
ruang
        diagonals = [
            np.sum([self.cube[i, i, i] for i
in range(self.n)]),
            np.sum([self.cube[i, i, self.n - i
- 1] for i in range(self.n)]),
            np.sum([self.cube[i, self.n - i -
1, i] for i in range(self.n)]),
            np.sum([self.cube[i, self.n - i -
1, self.n - i - 1] for i in range(self.n)]),
        ]
        total_difference +=
sum(abs(magic_number - diag_sum) for diag_sum
in diagonals)

        # Perbedaan jumlah di setiap diagonal
bidang
        for layer in self.cube:
            diagonal1 = np.sum([layer[i, i]
for i in range(self.n)])  # Diagonal dari kiri
atas ke kanan bawah di setiap layer
            diagonal2 = np.sum([layer[i,
self.n - i - 1] for i in range(self.n)])  #
Diagonal dari kanan atas ke kiri bawah di
setiap layer
            total_difference +=
```

```
abs(magic_number - diagonal1)
            total_difference +=
abs(magic_number - diagonal2)

        for row in range(self.n):
            diagonal1 = np.sum([self.cube[row,
i, i] for i in range(self.n)])  # Diagonal
dari kiri atas ke kanan bawah di setiap baris
(3D)
            diagonal2 = np.sum([self.cube[row,
i, self.n - i - 1] for i in range(self.n)])  #
Diagonal dari kanan atas ke kiri bawah di
setiap baris (3D)
            total_difference +=
abs(magic_number - diagonal1)
            total_difference +=
abs(magic_number - diagonal2)

        for col in range(self.n):
            diagonal1 = np.sum([self.cube[i,
i, col] for i in range(self.n)])  # Diagonal
dari kiri atas ke kanan bawah di setiap kolom
(3D)
            diagonal2 = np.sum([self.cube[i,
self.n - i - 1, col] for i in range(self.n)])
# Diagonal dari kanan atas ke kiri bawah di
setiap kolom (3D)
            total_difference +=
abs(magic_number - diagonal1)
            total_difference +=
abs(magic_number - diagonal2)

        return total_difference
```

2. Penjelasan Algoritma Local Search

2.1 Steepest Ascent Hill-Climbing

Steepest ascent hill climbing adalah salah satu metode optimasi berbasis pencarian lokal yang bertujuan untuk menemukan solusi terbaik dengan memperbaiki current state secara iteratif. Dalam konteks Magic Cube 5x5x5, algoritma ini diawali dengan initial state acak di mana jumlah angka pada beberapa baris, kolom, dan diagonal mungkin jauh dari magic number (315). Pada setiap iterasi, algoritma mengidentifikasi semua neighbor dari current state yang dihasilkan dari kemungkinan pertukaran dua angka dalam kubus. Setiap neighbor kemudian dievaluasi menggunakan objective function yang mengukur seberapa dekat konfigurasi tersebut dengan kondisi magic cube yang ideal. Algoritma memilih neighbor dengan nilai objective function terendah, yaitu yang paling mendekati 0, yang menandakan konfigurasi lebih optimal dibandingkan current state.

Proses ini berlanjut secara iteratif, di mana current state diperbarui dengan neighbor terbaik dari iterasi sebelumnya, dan algoritma kembali mencari neighbor yang memberikan penurunan nilai objective function terbesar. Algoritma berhenti ketika tidak ada lagi neighbor yang menawarkan hasil yang lebih baik dari current state saat ini, menghasilkan state terbaik yang ditemukan dalam proses tersebut. Namun, steepest ascent hill climbing memiliki keterbatasan dalam menjamin solusi global maximum karena algoritma ini dapat berhenti pada local maximum, yaitu konfigurasi yang tidak dapat diperbaiki lebih lanjut tetapi masih jauh dari konfigurasi ideal (global maximum). Misalnya, algoritma mungkin berhenti dengan nilai objective function 50, yang menunjukkan state optimal lokal tetapi bukan solusi ideal yang bernilai 0.

2.2 Hill-Climbing with Sideways Move

Sideways move hill climbing adalah varian dari hill climbing yang memperkenankan gerakan lateral ketika tidak ada neighbor yang secara langsung meningkatkan kualitas solusi, yaitu menurunkan nilai objective function. Dalam penerapannya pada masalah Magic Cube 5x5x5, algoritma dimulai dari initial state acak di mana jumlah angka pada beberapa baris, kolom, dan diagonal dapat sangat berbeda dari magic number (315). Seperti pada Steepest Ascent Hill Climbing, algoritma mengevaluasi semua

neighbor dari current state yang diperoleh dari kemungkinan pertukaran dua angka dalam kubus. Setiap neighbor dinilai berdasarkan nilai objective function untuk melihat sejauh mana state tersebut mendekati solusi ideal. Jika ada neighbor dengan nilai objective function lebih rendah dari current state, algoritma akan berpindah ke neighbor tersebut.

Namun, yang membedakan sideways move hill climbing adalah kemampuannya untuk melakukan gerakan lateral atau "sideways move" ketika tidak ada neighbor yang memperbaiki nilai objective function tetapi ada yang memiliki nilai yang sama. Dengan melakukan gerakan sideways, algoritma mencoba menghindari local maximum dengan eksplorasi yang lebih luas, berharap menemukan jalan keluar menuju solusi yang lebih baik. Misalnya, jika current state memiliki nilai objective function 430 dan tidak ada neighbor yang memiliki nilai lebih rendah, tetapi ada neighbor dengan nilai yang sama, algoritma akan memilih salah satu neighbor dengan nilai 430. Proses ini berlanjut hingga algoritma menemukan neighbor dengan nilai yang lebih kecil, misalnya 400, atau berhenti ketika tidak ada perbaikan lebih lanjut. Meskipun sideways moves meningkatkan peluang keluar dari local maximum sementara, algoritma ini masih memiliki risiko berhenti pada local maximum tanpa mencapai solusi optimal (global maximum) karena bergantung pada eksplorasi terbatas yang mungkin tidak selalu menemukan state dengan nilai objective function 0.

2.3 Random Restart Hill-Climbing

Random restart hill climbing adalah metode optimasi yang memperluas algoritma hill climbing untuk meningkatkan kemungkinan menemukan solusi optimal dalam ruang pencarian yang kompleks, seperti dalam permasalahan Magic Cube 5x5x5. Algoritma ini dimulai dengan initial state acak di mana jumlah angka pada baris, kolom, dan diagonal mungkin jauh dari nilai ideal (magic number 315). Pada setiap iterasi, algoritma memeriksa semua neighbor dari current state yang dihasilkan dari pertukaran dua angka pada kubus, lalu mengevaluasi setiap neighbor berdasarkan nilai objective function f(s). Jika ada neighbor dengan nilai objective function yang lebih rendah, algoritma bergerak ke neighbor tersebut sebagai current state baru. Jika tidak ada perbaikan langsung, algoritma dapat melakukan sideways move dengan memilih neighbor yang memiliki nilai objective function sama dengan current state saat ini.

Namun, berbeda dengan steepest ascent hill climbing, ketika algoritma terjebak pada local maximum (di mana tidak ada neighbor yang memiliki nilai objective lebih kecil), algoritma ini melakukan random restart, yaitu memulai kembali dari posisi acak baru. Proses ini memberi kesempatan untuk mengeksplorasi bagian lain dari ruang pencarian yang mungkin mengarah pada solusi yang lebih baik. Berdasarkan probabilitas keberhasilan p dalam menemukan solusi yang lebih baik setelah satu kali pencarian, jumlah restart yang diharapkan dihitung menggunakan rumus $Expected\ nb\ of\ restarts = \frac{1}{p}$,, sementara jumlah langkah yang diharapkan untuk mencapai solusi lebih baik dihitung dengan $Expected\ nb\ of\ steps = s + \frac{f\,(1-p)}{p} = 100 + \frac{50\,(1-0.2)}{0.2} = 300$, dengan s sebagai langkah pencarian sebelum mencapai local maximum dan f sebagai langkah tambahan setelah restart. Dengan metode ini, algoritma dapat mencoba lebih banyak kemungkinan konfigurasi state, sehingga meningkatkan peluang untuk menemukan global maximum, meskipun tetap tidak menjamin tercapainya nilai objective function 0.

2.4 Stochastic Hill-Climbing

Stochastic hill climbing adalah metode optimasi yang berbeda dari steepest ascent hill climbing karena tidak mengevaluasi semua neighbor dari current state. Dalam algoritma ini, hanya satu neighbor yang dipilih secara acak pada setiap iterasi untuk dievaluasi berdasarkan nilai objective function f(s), yang mengukur seberapa dekat konfigurasi tersebut dengan solusi ideal. Jika neighbor yang dipilih memiliki nilai objective function yang lebih baik dibandingkan current state, maka algoritma memperbarui current state ke neighbor tersebut. Pendekatan ini memungkinkan variasi dalam proses pencarian dan dapat membantu algoritma untuk keluar dari local maximum yang mungkin sulit dijangkau oleh metode hill climbing lain yang lebih deterministik.

Penggunaan pemilihan neighbor secara acak membuat stochastic hill climbing lebih fleksibel dalam mengeksplorasi ruang pencarian dan memberikan peluang yang lebih besar untuk menemukan solusi alternatif. Namun, algoritma ini juga memiliki batasan, karena tanpa seleksi yang sistematis, algoritma bisa terjebak pada local maximum dan pencarian menjadi kurang terarah. Untuk mencegah pencarian yang berkepanjangan tanpa perbaikan yang signifikan, batas jumlah iterasi dapat ditetapkan,

misalnya berhenti setelah nmax kali Pengujian tanpa peningkatan. Meskipun pendekatan acak ini tidak selalu menjamin solusi optimal, stochastic hill climbing dapat menjelajahi konfigurasi yang lebih luas dalam ruang pencarian, yang pada beberapa kasus memberikan peluang lebih tinggi untuk mendekati kondisi optimal daripada algoritma hill climbing konvensional yang hanya mencari solusi terbaik dari seluruh neighbor yang ada.

2.5 Simulated Annealing

Simulated annealing adalah metode optimasi yang memperkenalkan elemen probabilistik dalam proses pencarian solusi, bertujuan untuk menghindari jebakan local maximum dan menemukan solusi optimal atau mendekati optimal. Pada masalah Magic Cube 5x5x5, algoritma ini dimulai dari initial state acak, di mana susunan angka pada beberapa baris, kolom, dan diagonal mungkin jauh dari magic number (315). Setiap iterasi melibatkan pemilihan neighbor dari current state, yang diperoleh melalui pertukaran dua angka dalam kubus. Namun, tidak seperti algoritma hill climbing lainnya, simulated annealing memungkinkan pilihan neighbor dengan nilai objective function yang lebih buruk dari current state dengan probabilitas tertentu, yang dikendalikan oleh parameter "temperatur". Di awal pencarian, dengan temperatur yang tinggi, algoritma cenderung lebih fleksibel, sehingga menerima state yang lebih buruk untuk mendorong eksplorasi ruang pencarian yang lebih luas.

Seiring berjalannya iterasi, temperatur akan diturunkan secara bertahap, yang mengurangi probabilitas algoritma menerima neighbor dengan nilai objective function yang lebih tinggi dari current state. Penurunan temperatur ini membuat algoritma semakin selektif dalam memilih neighbor, berfokus hanya pada neighbor yang memperbaiki current state seiring mendekati solusi optimal. Proses ini berlanjut hingga temperatur mencapai batas minimum, iterasi maksimum tercapai, atau solusi optimal (global maximum) ditemukan. Pendekatan ini memungkinkan algoritma untuk menjelajahi lebih banyak alternatif di awal pencarian dan secara bertahap memfokuskan pencarian pada solusi yang lebih baik. Dengan simulasi penurunan temperatur yang tepat, simulated annealing memiliki potensi yang lebih besar dalam mencapai state mendekati global maximum dibandingkan metode optimasi lokal lain yang tidak memiliki mekanisme untuk menerima solusi sementara yang lebih buruk.

2.6 Genetic Algorithm

Genetic Algorithm (GA) merupakan algoritma optimasi yang terinspirasi oleh proses evolusi biologis, di mana solusi dalam bentuk kromosom mengalami seleksi, crossover, dan mutasi untuk mencapai solusi optimal atau mendekati optimal. Dalam penerapannya pada permasalahan Magic Cube 5x5x5, GA diawali dengan pembentukan initial state dalam bentuk deretan kromosom acak, misalnya dua kromosom (K=2), yang masing-masing merepresentasikan susunan angka 0 hingga 125, menggambarkan seluruh elemen dalam Magic Cube yang telah didekomposisi. Setiap kromosom dievaluasi menggunakan objective function untuk menilai seberapa dekat susunan tersebut dengan solusi optimal. Nilai objective function yang lebih tinggi mengindikasikan susunan kromosom yang kurang sesuai, dan sebaliknya. Berdasarkan hasil objective function, persentase probabilitas seleksi dihitung, yang memungkinkan kromosom dengan nilai terbaik memiliki peluang lebih besar untuk berpartisipasi dalam tahap crossover.

Setelah seleksi, tahap crossover dilakukan dengan menentukan satu titik crossover secara acak untuk menukar bagian dari kromosom sehingga tercipta kombinasi karakteristik dari kedua kromosom induk. Mekanisme mutasi diterapkan secara acak untuk mencegah hasil konvergensi terlalu cepat dan meningkatkan keragaman solusi. Mutasi ini dilakukan dengan mengganti elemen tertentu pada kromosom dengan angka baru secara acak, lalu nilai objective function dihitung kembali untuk kedua kromosom hasil crossover. Proses ini berulang hingga mencapai solusi global optimum atau iterasi maksimal. Namun, dalam konteks Magic Cube, GA memiliki keterbatasan. Constraint dari Magic Cube, seperti susunan angka unik dan properti diagonal yang harus terpenuhi, membuat peluang menemukan solusi yang valid melalui kombinasi kromosom sangat kecil. Selain itu, GA memerlukan waktu iterasi yang panjang, menjadikannya kurang efisien untuk menyelesaikan masalah Magic Cube secara langsung.

3. Implementasi Algoritma Local Search

3.1 Struktur Folder

Berikut adalah struktur folder yang diterapkan pada keseluruhan program Magic Cube Solver kami.

```
src
├── algorithm
│   ├── genetic_algorithm.py
│   ├── hill_climbing.py
│   └── simulated_annealing.py
├── driver
│   ├── genetic-driver.py
│   ├── hillclimb-driver.py
│   └── simulated-driver.py
├── handlers
│   └── algorithm.py
├── models
│   └── cube.py
├── main.py
└── Visual.py
```

3.2 Hill Climbing

Untuk penerapan genetic algorithm terdapat pada file hill_climbing.py

Tabel 3.2.1. Fungsi Get Neighbor Hill Climbing

Nama Fungsi	get_neighbor(self, cube, current_score)
Deskripsi Fungsi	Fungsi ini melakukan pencarian neighbor terbaik dari cube dengan melakukan berbagai kemungkinan

	pertukaran posisi pada setiap elemen cube. Fungsi mengembalikan cube dengan posisi elemen yang telah diubah sesuai dengan swap terbaik yang ditemukan.
Kode	

```
def get_neighbor(self, cube, current_score):

        # Inisialisasi variabel untuk menyimpan skor terendah dan swap terbaik
        lowest_score = current_score
        best_swap = None
        n = cube.n

        # Looping untuk semua kemungkinan swap pada setiap state
        for i in range(n):
            for j in range(n):
                for k in range(n):
                    for x in range(i, n):
                        for y in range(j if x == i else 0, n):
                            for z in range(k + 1 if x == i and y == j else 0, n):
                                # Melakukan swap pada kubus
                                cube.cube[i, j, k], cube.cube[x, y, z] = cube.cube[x, y, z], cube.cube[i, j, k]

                                # Menghitung skor dari kubus yang telah di-swap
                                new_score = cube.objective_function()

                                # Jika skor baru lebih rendah dari skor terendah, maka menyimpan swap tersebut
                                if new_score < lowest_score:
                                    lowest_score =
```

	`` new_score best_swap = (i, j, k, x, y, z) # Mengembalikan kubus ke keadaan semula cube.cube[i, j, k], cube.cube[x, y, z] = cube.cube[x, y, z], cube.cube[i, j, k] # Jika ditemukan swap terbaik, maka dilakukan swap tersebut pada kubus if best_swap: i, j, k, x, y, z = best_swap cube.cube[i, j, k], cube.cube[x, y, z] = cube.cube[x, y, z], cube.cube[i, j, k] else: cube = None # Mengembalikan kubus yang telah di-swap return cube ```

Tabel 3.2.2. Fungsi Get All Neighbors Hill Climbing

Nama Fungsi	get_all_neighbors(self, cube)
Deskripsi Fungsi	Fungsi ini menghasilkan semua kemungkinan neighbor dari suatu cube. Neighbor dihasilkan dengan melakukan swap antar elemen cube. Fungsi mengembalikan list yang mencakup semua kemungkinan neighbor yang dapat dihasilkan dari initial cube.
Kode	``` def get_all_neighbors(self, cube): # Inisialisasi variabel n = cube.n neighbors = [] # Looping untuk semua kemungkinan swap pada setiap state ```

```
        for i in range(n):
            for j in range(n):
                for k in range(n):
                    for x in range(i, n):
                        for y in range(j if x == i
else 0, n):
                            for z in range(k + 1 if x
== i and y == j else 0, n):
                                # Melakukan swap pada
kubus
                                cube.cube[i, j, k],
cube.cube[x, y, z] = cube.cube[x, y, z], cube.cube[i,
j, k]

                                # Membuat kubus baru
                                new_cube = Cube()
                                new_cube.cube =
np.copy(cube.cube)

                                # Menambahkan kubus
yang telah di-swap ke dalam list neighbors

neighbors.append(new_cube)

                                # Mengembalikan kubus
ke keadaan semula
                                cube.cube[i, j, k],
cube.cube[x, y, z] = cube.cube[x, y, z], cube.cube[i,
j, k]

        # Mengembalikan semua neighbors
        return neighbors
```

3.2.1 Steepest Ascent Hill Climbing

Untuk penerapan Steepest Ascent terdapat pada file hill_climbing.py

Tabel 3.2.1.1. Fungsi Steepest Ascent Hill Climbing

Nama Fungsi	steepest_ascent(self, cube, max_iterations)
Deskripsi Fungsi	Fungsi ini merupakan fungsi utama yang melakukan pencarian dengan algoritma Steepest Ascent Hill Climbing. Pada setiap iterasi, fungsi ini memanggil get_neighbor untuk mencari neighbor terbaik, yang kemudian dijadikan sebagai current state. Fungsi akan mengembalikan cube terbaik, nilai objective terbaik, jumlah iterasi, dan semua objective function.
Kode	`def steepest_ascent(self, cube, max_iterations):` `  # Inisialisasi variabel untuk menyimpan kubus saat ini dan objective function saat ini` `  current_state = cube` `  current_score = current_state.objective_function()` `  iterations = 0` `  scores = []` `  display_text = []` `  # Looping hingga mencapai maksimum iterasi` `  while iterations < max_iterations:` `  # Menyimpan skor pada iterasi saat ini` `  scores.append(current_score)` `  # Mendapatkan neighbor terbaik dari kubus saat ini` `  neighbor = self.get_neighbor(current_state, current_score)` `  # Jika tidak ditemukan neighbor terbaik, maka berhenti` `  if neighbor is None:`

```
                break  # Break loop ketika tidak ada neighbor yang lebih baik
            # Jika ditemukan neighbor terbaik, maka kubus saat ini diganti dengan neighbor terbaik
            else:
                current_state = neighbor
                current_score = current_state.objective_function()

            display_text = Visual.display_iteration(display_text, iterations, current_score, "stepest_ascent")

            iterations += 1

        # Mengembalikan kubus terbaik, nilai objektif terbaik, jumlah iterasi, dan semua objektif function
        return current_state, current_score, len(scores), scores
```

3.2.2 Sideways Move Hill Climbing

Untuk penerapan Sideways Move terdapat pada file hill_climbing.py

Tabel 3.2.2.1. Fungsi Sideways Move Hill Climbing

Nama Fungsi	sideways_move(self, cube, max_sideways)
Deskripsi Fungsi	Fungsi ini merupakan fungsi utama untuk melakukan searching menggunakan hill climbing with sideways move algorithm. Fungsi ini akan menerima input cube dan max_sideways dan memanggil get_neighbor untuk kemudian mengevaluasi objective function. Fungsi ini akan mengembalikan cube terbaik, nilai objective terbaik, jumlah iterasi, dan semua objective function.

Kode

```
def sideways_move(self, cube, max_sideways):
        # Inisialisasi variabel untuk menyimpan kubus saat ini dan objective function saat ini
        current_state = cube
        current_score = current_state.objective_function()
        scores = []
        display_text = []
        iterations = 0
        sideways_count = 0

        # Looping hingga ditemukan solusi optimal
        while True:

            # Menyimpan skor pada iterasi saat ini
            scores.append(current_score)

            # Mendapatkan neighbor terbaik dari kubus saat ini
            neighbor = self.get_neighbor(current_state, current_score)

            # Mengambil neighbor dan menghitung skor dari neighbor terbaik
            next_state = neighbor
            next_score = next_state.objective_function()

            # Jika skor dari neighbor lebih baik dari skor saat ini, maka kubus saat ini diganti dengan neighbor terbaik
            if next_score < current_score:
                current_state = next_state
                current_score = next_score
                sideways_count = 0
            # Jika skor dari neighbor sama dengan skor saat ini, maka kubus saat ini diganti dengan neighbor terbaik
            else:
```

```
                # Jika skor dari neighbor sama dengan
skor saat ini, maka kubus saat ini diganti dengan
neighbor terbaik
                if next_score == current_score:
                    current_state = next_state
                    current_score = next_score
                    sideways_count += 1  # Menambahkan
jumlah sideways
                # Jika skor dari neighbor lebih buruk
dari skor saat ini, maka berhenti
                else:
                    break

            display_text =
Visual.display_iteration(display_text, iterations,
current_score, "sideways_move")

            iterations += 1

            # Berhenti jika jumlah sideways lebih dari
maksimum yang ditentukan
            if sideways_count >= max_sideways:
                print("Reached max sideways moves,
stopping.")
                break

        # Mengembalikan kubus terbaik, nilai objektif
terbaik, jumlah iterasi, dan semua objektif function
        return current_state, current_score,
len(scores), scores
```

3.2.3 Random Restart Hill Climbing

Untuk penerapan Random Restart terdapat pada file hill_climbing.py

Tabel 3.2.3.1. Fungsi Random Restart Hill Climbing

Nama Fungsi	random_restart(self, cube, max_restarts,

	max_iterations)
Deskripsi Fungsi	Fungsi ini merupakan fungsi utama yang melakukan pencarian dengan algoritma Random Restart Hill Climbing. Fungsi ini akan menerima input max_restarts dan max_iterations, lalu memanggil fungsi steepest_ascent untuk mendapatkan state terbaik. Fungsi ini akan mengembalikan cube terbaik, nilai objective terbaik, jumlah iterasi, dan semua objective function.
Kode	`def random_restart(self, cube, max_restarts, max_iterations):` `  # Inisialisasi variabel untuk menyimpan kubus terbaik, nilai objektif terbaik, dan skor terbaik` `  best_state = cube` `  best_score = best_state.objective_function()` `  scores = []` `  display_text = []` `  # Looping untuk setiap restart` `  for restart in range(max_restarts):` `  # Inisialisasi kubus baru` `  cube_instance = Cube()` `  # Menjalankan algoritma steepest ascent` `  current_state, current_score, _, current_scores = self.steepest_ascent(cube_instance, max_iterations)` `  # Menyimpan skor dari iterasi saat ini` `  scores.extend(current_scores)`

```
            # Cek jika skor dari iterasi ini
lebih baik dari best_score
            if current_score < best_score:
                best_state = current_state
                best_score = current_score

            display_text =
Visual.display_iteration(display_text,
restart, current_score, "random_restart",
max_restarts)

            # Berhenti jika ditemukan solusi
optimal (skor 0)
            if current_score == 0:
                print("Optimal solution
found!")
                break

        # Mengembalikan hasil terbaik dan
semua skor dari semua iterasi
        return best_state, best_score,
len(scores), scores
```

3.2.4 Stochastic Hill Climbing

Untuk penerapan Stochastic terdapat pada file hill_climbing.py

Tabel 3.2.4.1. Fungsi Stochastic Hill Climbing

Nama Fungsi	stochastic(self, cube, max_iterations)
Deskripsi Fungsi	Fungsi ini merupakan fungsi utama untuk melakukan searching menggunakan stochastic hill climbing. Fungsi ini akan menerima input cube dan max_iterations dan memanggil get_all_neighbor untuk kemudian memilih neighbor secara acak dan mengevaluasi objective function. Fungsi ini akan mengembalikan cube terbaik, nilai objective terbaik,

	jumlah iterasi, dan semua objective function.
Kode	

```
def stochastic(self, cube, max_iterations):
        # Inisialisasi variabel untuk menyimpan kubus saat ini dan objective function saat ini
        current_state = cube
        current_score = current_state.objective_function()
        scores = []
        display_text = []

        # Looping hingga mencapai maksimum iterasi
        for iteration in range(max_iterations):

            # Menyimpan skor pada iterasi saat ini
            scores.append(current_score)

            # Mendapatkan neighbor terbaik dari kubus saat ini
            neighbors = self.get_all_neighbors(current_state)

            # Mengambil neighbor secara acak
            next_state = random.choice(neighbors)
            next_score = next_state.objective_function()

            # Jika skor dari neighbor lebih baik dari skor saat ini, maka kubus saat ini diganti dengan neighbor terbaik
            if next_score < current_score:
                current_state = next_state
                current_score = next_score
            # Jika skor dari neighbor sama dengan skor saat ini, maka kubus saat ini diganti dengan neighbor terbaik
            else:
                pass

            # Menampilkan skor dan iterasi saat ini
```

	display_text = Visual.display_iteration(display_text, iteration, current_score, "stochastic") # Mengembalikan kubus terbaik, nilai objektif terbaik, jumlah iterasi, dan semua objektif function return current_state, current_score, len(scores), scores

3.6 Simulated Annealing

Untuk penerapan Simulated Annealing terdapat pada file simulated_annealing.py

Tabel 3.6.1. Fungsi Konstruktor Simulated Annealing

Nama Fungsi	__init__(self)
Deskripsi Fungsi	Fungsi ini merupakan konstruktor dari Simulated Annealing untuk menginisialisasi variabel best_cube (state terbaik) dan best_value (nilai objektif terbaik).
Kode	def __init__(self): self.best_cube = None self.best_value = float('inf')

Tabel 3.6.2. Fungsi Set State Simulated Annealing

Nama Fungsi	set_state(self, cube, objective_value)
Deskripsi Fungsi	Fungsi ini menyimpan *current state* terbaik dan nilai objektifnya, serta memperbarui variabel best_cube dan best_value.
Kode	def set_state(self, cube, objective_value): cubeInstance = Cube() cubeInstance.cube = cube self.best_cube = cubeInstance self.best_value = objective_value

Tabel 3.6.3. Fungsi Generate Neighbor Simulated Annealing

Nama Fungsi	generate_neighbor(self, current_state)
Deskripsi Fungsi	Fungsi ini menghasilkan *neighbor state* dengan menukar dua angka secara acak pada kubus saat ini, serta menghitung nilai objektif dari *neighbor state* yang dihasilkan.
Kode	`def generate_neighbor(self, current_state, ):` `  # Melakukan copy pada current state` `  neighbor_state = np.copy(current_state)` `  # Melakukan swap pada dua angka secara acak` `  x1, y1, z1 = random.randint(0, 4), random.randint(0, 4), random.randint(0, 4)` `  x2, y2, z2 = random.randint(0, 4), random.randint(0, 4), random.randint(0, 4)` `  neighbor_state[x1, y1, z1], neighbor_state[x2, y2, z2] = neighbor_state[x2, y2, z2], neighbor_state[x1, y1, z1]` `  # Menghitung nilai objektif dari neighbor` `  cube_instance = Cube()` `  cube_instance.cube = neighbor_state` `  neighbor_objective = cube_instance.objective_function()` `  # Mengembalikan neighbor state dan objective` `  return neighbor_state, neighbor_objective`

Tabel 3.6.4. Fungsi Search Simulated Annealing

Nama Fungsi	simulated_annealing(self, cube_instance, initial_temperature, cooling_rate, min_temperature, max_iterations)
Deskripsi Fungsi	Fungsi ini merupakan fungsi utama untuk melakukan searching menggunakan algoritma Simulated Annealing. Fungsi ini menerima input *state* awal (cube_instance), initial_temperature, cooling_rate, min_temperature, dan max_iterations kemudian

	memanggil fungsi generate_neighbor. Fungsi ini mengembalikan state terbaik, nilai objektif terbaik, nilai objektif setiap iterasi, *acceptance probability neighbor* yang lebih buruk, dan jumlah iterasi ketika algoritma terjebak pada *local optimum.*
Kode	

```
def simulated_annealing(self, cube_instance, initial_temperature, cooling_rate, min_temperature, max_iterations):
        # Inisialisasi variabel
        T = initial_temperature
        current_state = np.copy(cube_instance.cube)
        current_score = cube_instance.objective_function()
        display_text = []

        # Set state awal
        self.set_state(current_state, current_score)

        # Inisialisasi list untuk menyimpan nilai objective function dan acceptance probability
        objective_values = []
        acceptance_probabilities = []

        # Inisialisasi counter stuck dan unchanged iterations
        stuck_counter = 0
        unchanged_iterations = 0

        # Melakukan iterasi sebanyak iterasi maksimal yang di-input
        for iteration in range(max_iterations):
            # Jika temperatur kurang dari temperatur minimum, maka stop iterasi
            if T < min_temperature:
                break

            # Generate neighbor dengan memanggil fugnsi
            neighbor_state, neighbor_objective =
```

```
self.generate_neighbor(current_state)

            # Menghitung ΔE
            delta_E = neighbor_objective -
current_score

            # Menghitung acceptance probability
            acceptance_probability = math.exp(-delta_E
/ T) if delta_E > 0 else 1

            # Menerima neighbor jika acceptance
probability lebih besar dari random uniform
            if random.uniform(0, 1) <
acceptance_probability:
                current_state = neighbor_state
                current_score = neighbor_objective

                # Update best state jika nilai
objective lebih kecil
                if current_score < self.best_value:
                    self.set_state(current_state,
current_score)



            # Menyimpan nilai objective function
            objective_values.append(current_score)

            # Menyimpan acceptance probability jika ΔE
> 0
            if delta_E > 0:

acceptance_probabilities.append(acceptance_probabilit
y)

            # Jika ΔE = 0, maka tambah unchanged
iterations dan stuck counter
            if delta_E == 0:
                unchanged_iterations += 1
                stuck_counter += 1
```

```
            else:
                unchanged_iterations = 0

            # Mengurangi temperatur dengan cooling
rate
            T *= cooling_rate

            # Menampilkan nilai objective function
setiap iterasi
            display_text =
Visual.display_iteration(display_text, iteration,
current_score, "simulated_annealing")

        # Mengembalikan kubus terbaik, nilai objektif
terbaik, nilai objektif, acceptance probability, dan
stuck counter
        return self.best_cube, self.best_value,
objective_values, acceptance_probabilities,
stuck_counter
```

3.7 Genetic Algorithm

Untuk penerapan Genetic Algorithm terdapat pada file genetic_algorithm.py

Tabel 3.7.1. Fungsi Konstruktor Genetic Algorithm

Nama Fungsi	__init__(self)
Deskripsi Fungsi	Fungsi ini merupakan konstruktor dari Genetic Algorithm untuk kemudian di-instansiasi di file algorithm.py.
Kode	`def __init__(self):` `  self.best_cube = Cube()` `  self.best_value = float('inf')` `  self.display_text = []`

Tabel 3.7.2. Fungsi Populate Genetic Algorithm

Nama Fungsi	populate(self, population_size)
Deskripsi Fungsi	Fungsi ini merupakan fungsi untuk melakukan inisiasi populasi berupa sejumlah cube sesuai inputan user.
Kode	`def populate(self, population_size):` `  population = [Cube() for _ in range(population_size)]` `  return population`

Tabel 3.7.3. Fungsi Selection Genetic Algorithm

Nama Fungsi	selection(self, population)
Deskripsi Fungsi	Fungsi ini merupakan fungsi untuk melakukan proses seleksi pada populasi yang di-input oleh user.
Kode	`def selection(self, population):` `  # Menghitung nilai objektif untuk setiap kubus dalam populasi` `  fitness_scores = [cube.objective_function() for cube in population]` `  # Menghitung probabilitas seleksi secara kumulatif` `  total_fitness = sum(fitness_scores)` `  selection_probs = [(total_fitness - score) / total_fitness for score in fitness_scores]` `  cumulative_probs = np.cumsum(selection_probs) / np.sum(selection_probs) * 100` `  # Memilih parents berdasarkan probabilitas kumulatif` `  parents = []` `  for cube in range(len(population)):` `  random_percentage =`

	random.uniform(0, 100) for i, prob in enumerate(cumulative_probs): if random_percentage <= prob: parents.append(population[i]) break # Mengembalikan parents yang telah dipilih return parents

Tabel 3.7.4. Fungsi Crossover Genetic Algorithm

Nama Fungsi	crossover(self, parents)
Deskripsi Fungsi	Fungsi ini merupakan fungsi untuk melakukan proses crossover pada parents hasil proses seleksi
Kode	# Crossover function ensuring unique elements in parentren def crossover(self, parents): new_population = [] # Proses crossover untuk setiap pasangan parents for i in range(0, len(parents), 2): parent1 = parents[i] parent2 = parents[i + 1] # Menentukan titik crossover secara acak crossover_point = random.randint(0, 124) # Membuat dua kubus baru sebagai child child1 = Cube()

```
            child2 = Cube()

            # Melakukan crossover pada titik
yang telah ditentukan

child1.cube.flat[crossover_point:],
child2.cube.flat[crossover_point:] =
parent2.cube.flat[crossover_point:],
parent1.cube.flat[crossover_point:]

            # Mengisi sisa elemen yang belum
terisi pada child1 dan memastikan elemen yang
diisi unik
            currentIndex = 0
            for j in range(crossover_point,
125):
                for k in range(currentIndex,
125):
                    if parent2.cube.flat[k]
not in child1.cube.flat:
                        child1.cube.flat[j] =
parent2.cube.flat[k]
                        currentIdx = k
                        break

            # Mengisi sisa elemen yang belum
terisi pada child2 dan memastikan elemen yang
diisi unik
            currentIndex = 0
            for j in range(crossover_point,
125):
                for k in range(currentIndex,
125):
                    if parent1.cube.flat[k]
not in child2.cube.flat:
                        child2.cube.flat[j] =
parent1.cube.flat[k]
```

	`` currentIdx = k break # Menambahkan parents yang telah di-cross-over ke populasi baru new_population.append(parent1) new_population.append(parent2) return new_population ```

Tabel 3.7.5. Fungsi Mutation Genetic Algorithm

Nama Fungsi	mutation(self, new_population)
Deskripsi Fungsi	Fungsi ini merupakan fungsi untuk melakukan proses mutasi pada populasi baru hasil dari proses crossover
Kode	``` # Proses Mutasi def mutation(self, new_population): for individual in new_population: # Melakukan swap pada dua elemen secara acak idx1, idx2 = random.randint(0, 124), random.randint(0, 124) individual.cube.flat[idx1], individual.cube.flat[idx2] = individual.cube.flat[idx2], individual.cube.flat[idx1] return new_population ```

Tabel 3.7.6. Fungsi Search Genetic Algorithm

Nama Fungsi	genetic_search(self, population_size, max_iterations)
Deskripsi Fungsi	Fungsi ini merupakan fungsi utama untuk melakukan searching menggunakan genetic algorithm. Fungsi ini akan memanggil fungsi populate, selection,

	crossover, dan mutation kemudian mengembalikan state terbaik, score terbaik, populasi akhir, kumpulan score terbaik, dan rata-rate score dari semua iterasi yang sudah dilalui.
Kode	

```
# Proses pencarian menggunakan algoritma genetik
    def genetic_search(self, population_size, max_iterations):
        # Inisialisasi array skor
        min_scores = []
        avg_scores = []

        # Inisialisasi populasi
        population = self.populate(population_size)

        for iteration in range(max_iterations):
            # Proses seleksi
            parents = self.selection(population)

            # Proses crossover
            crossover_result = self.crossover(parents)

            # Proses mutasi
            population = self.mutation(crossover_result)

            # Menghitung nilai objektif untuk setiap kubus dalam populasi
            scores = [cube.objective_function() for cube in population]
            min_score = min(scores)
            max_score = max(scores)
```

```
            # Memperbarui skor terbaik dan
cube terbaik
            if min_score < self.best_value:
                self.best_value = min_score
                self.best_cube =
deepcopy(population[scores.index(self.best_va
lue)])

            # Menyimpan skor minimum dan
rata-rata

min_scores.append(self.best_value)
            avg_scores.append(sum(scores) /
len(scores))

            # Menampilkan informasi iterasi
            self.display_text =
Visual.display_iteration(self.display_text,
iteration, self.best_value, "genetic")

        # Mengembalikan kubus terbaik, nilai
objektif terbaik, populasi, skor minimum, dan
skor rata-rata
        return self.best_cube,
self.best_value, population, min_scores,
avg_scores
```

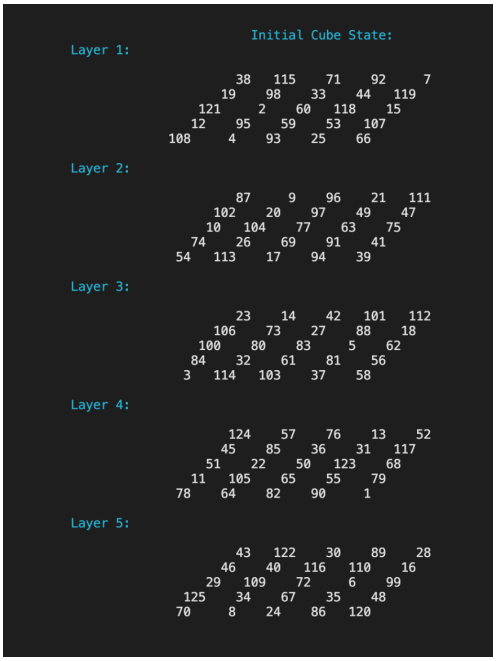
4. Hasil Eksperimen

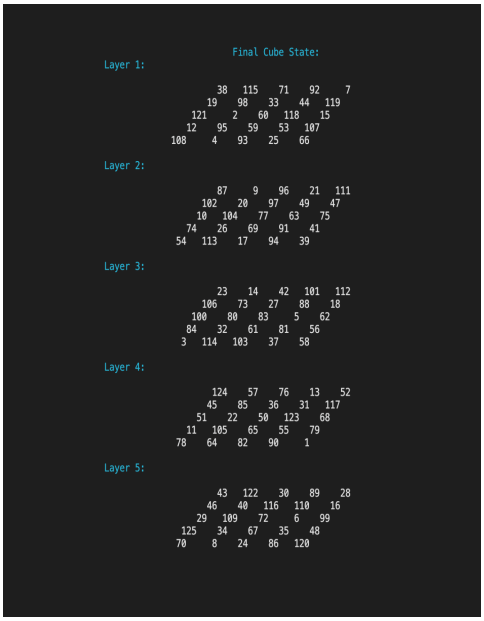
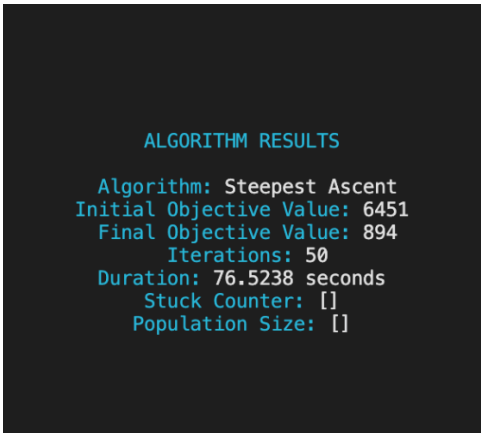
4.1 Hill Climbing

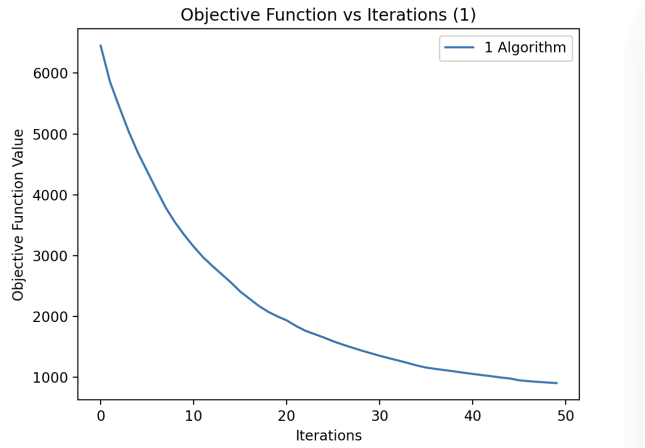
4.1.1 Steepest Ascent Hill Climbing

1. Pengujian 1 - Iterasi 50

Tabel 4.1.1.1. Pengujian 1 Steepest Ascent

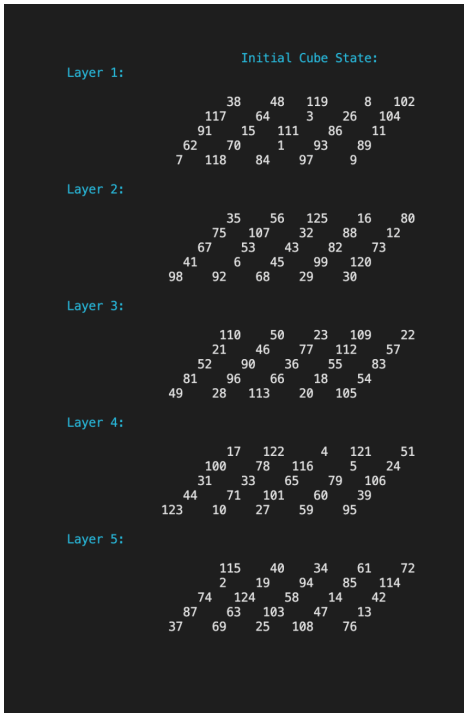
State Awal	
Objective Awal	6451

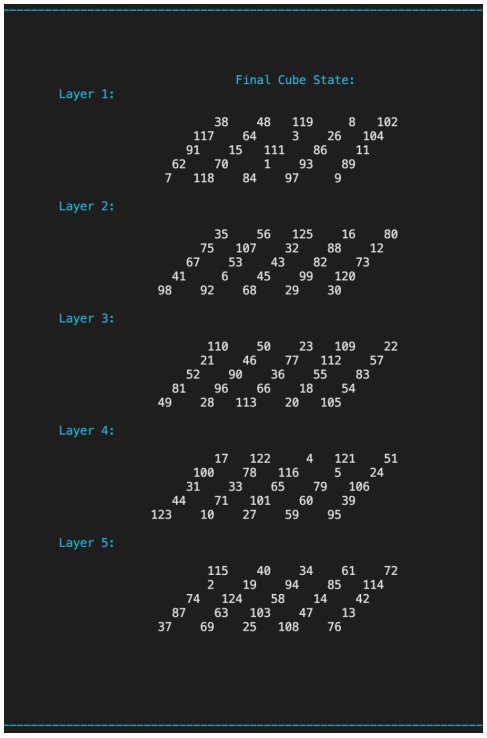
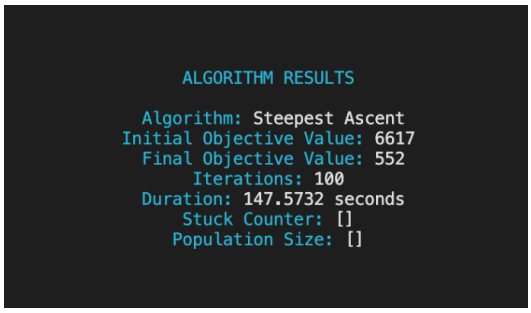
State Akhir	
Objective Akhir	894
Banyak Iterasi	50
Hasil Eksperimen	
Durasi Pencarian	76.5238 detik

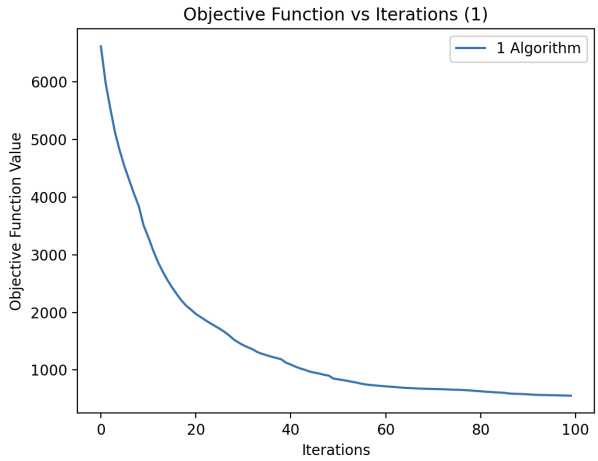
Plot Banyak Iterasi terhadap Objective Function	

2. Pengujian 2 - Iterasi 100

Tabel 4.1.1.2. Pengujian 2 Steepest Ascent

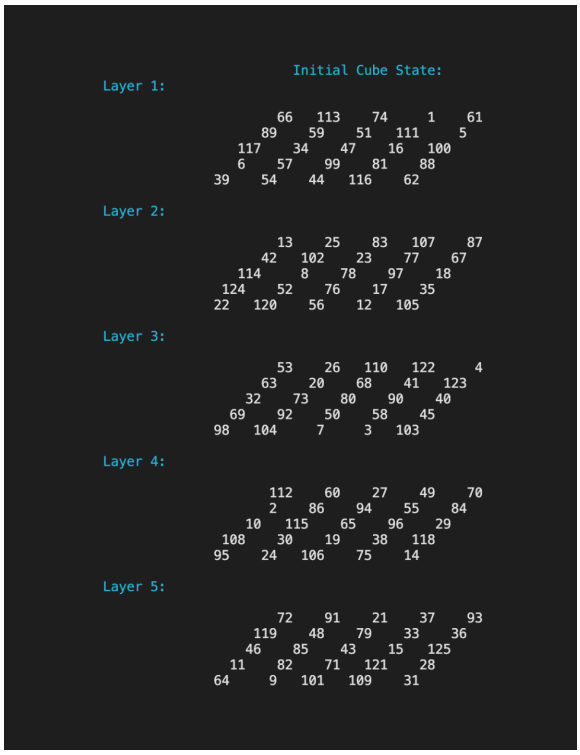
State Awal	
Objective Awal	6617

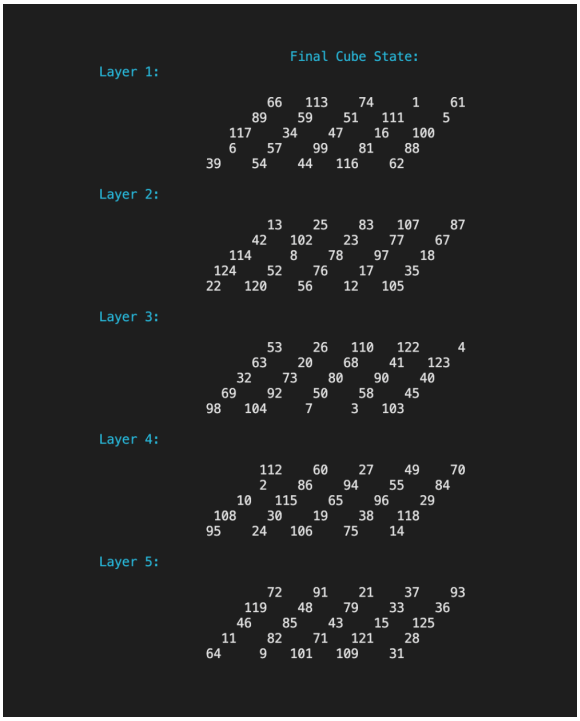
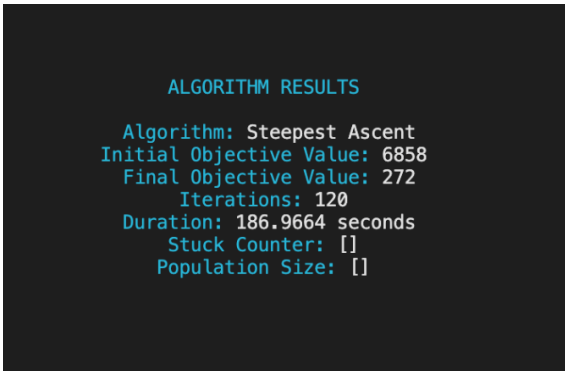
State Akhir	
Objective Akhir	552
Banyak Iterasi	100
Hasil Eksperimen	
Durasi Pencarian	147.5732

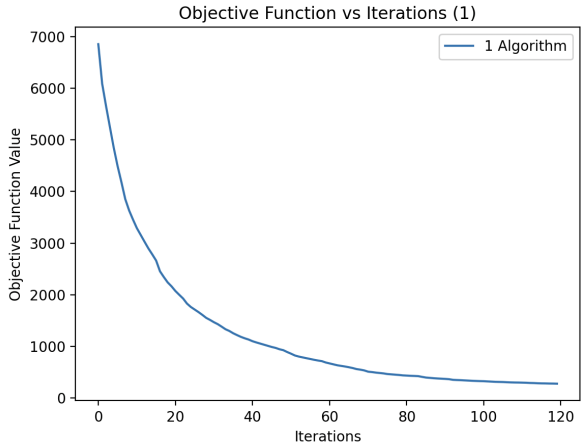
Plot Banyak Iterasi terhadap Objective Function	

3. Pengujian 3 - Iterasi 120

Tabel 4.1.1.3. Pengujian 2 Steepest Ascent

State Awal	

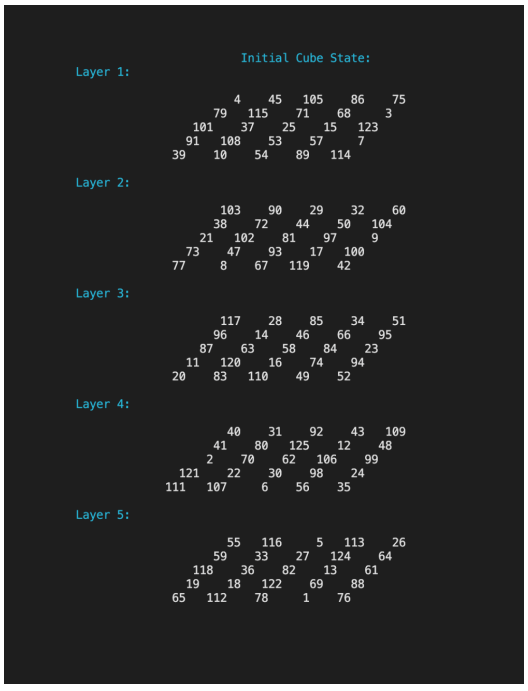
Objective Awal	6858
State Akhir	
Objective Akhir	272
Banyak Iterasi	120
Hasil Eksperimen	
Durasi Pencarian	186.9664 detik

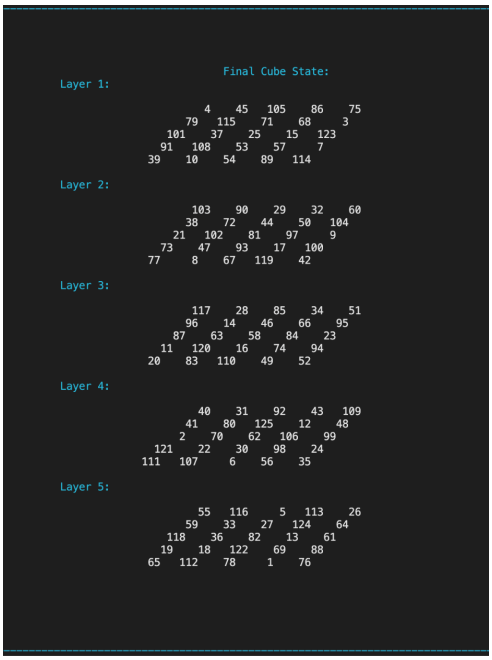
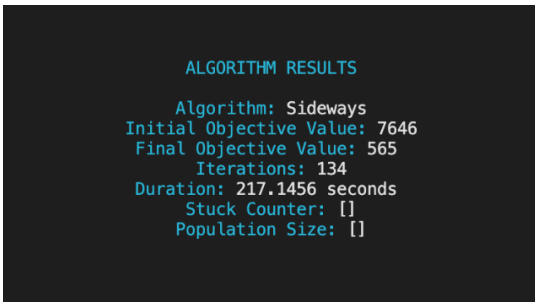
Plot Banyak Iterasi terhadap Nilai Objective Function	

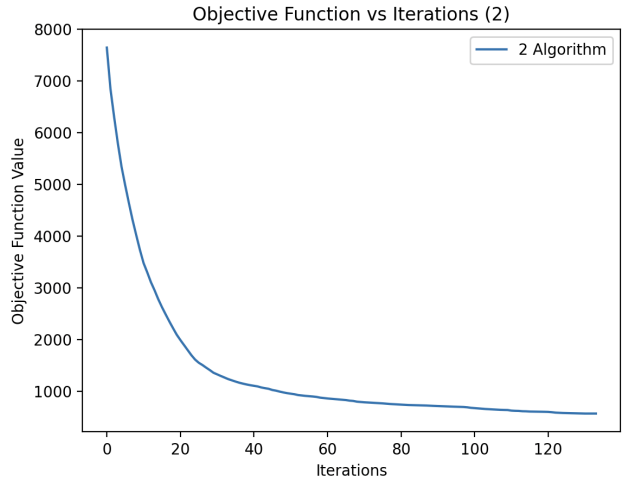
4.1.2 Hill Climbing with Sideways Move

1. Pengujian 1 - Iterasi Max Sideways 5

Tabel 4.1.2.1. Pengujian 1 Hill Climbing with Sideways Move

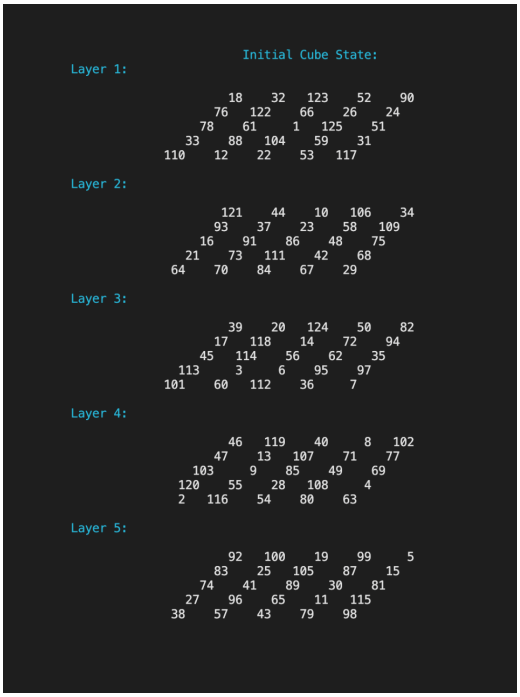
State Awal	

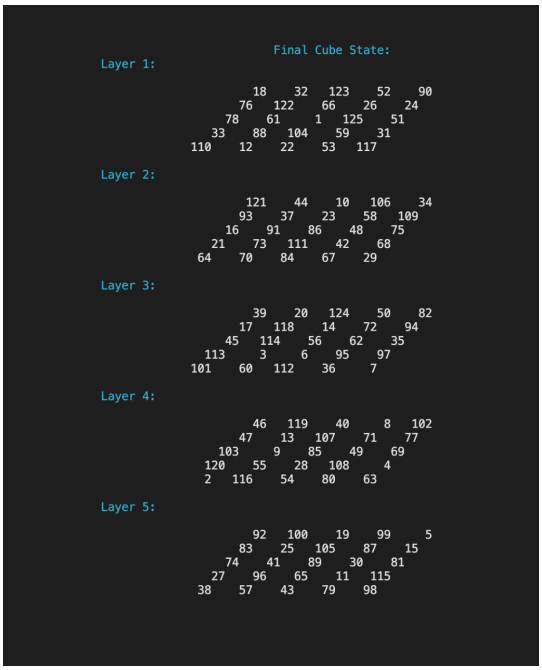
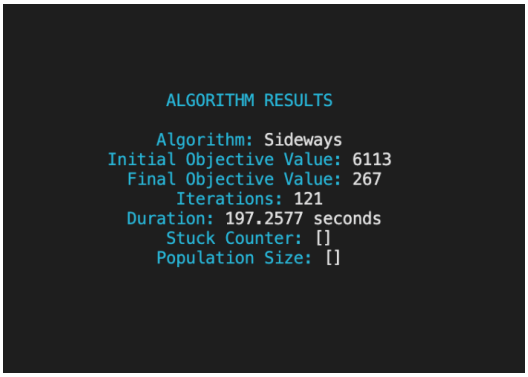
Objective Awal	7646
State Akhir	
Objective Akhir	565
Banyak Iterasi	134
Hasil Eksperimen	
Durasi Pencarian	217.1456

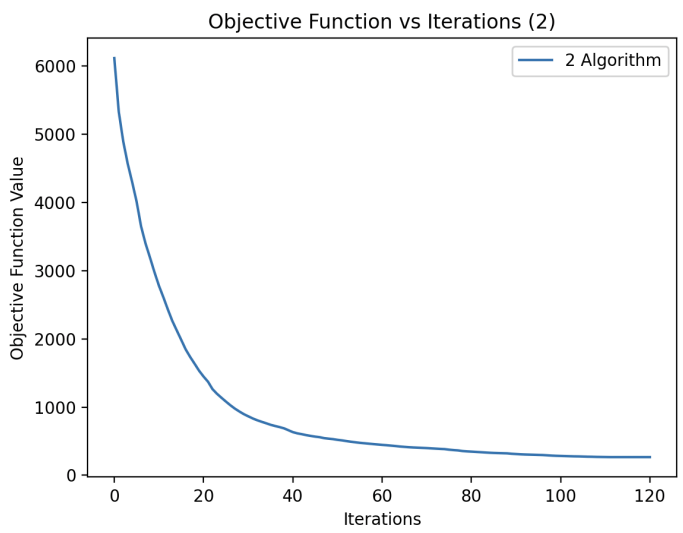
Plot Banyak Iterasi terhadap Objective Function	

2. Pengujian 2 - Iterasi Max Sideways 10

Tabel 4.1.2.2. Pengujian 2 Hill Climbing with Sideways Move

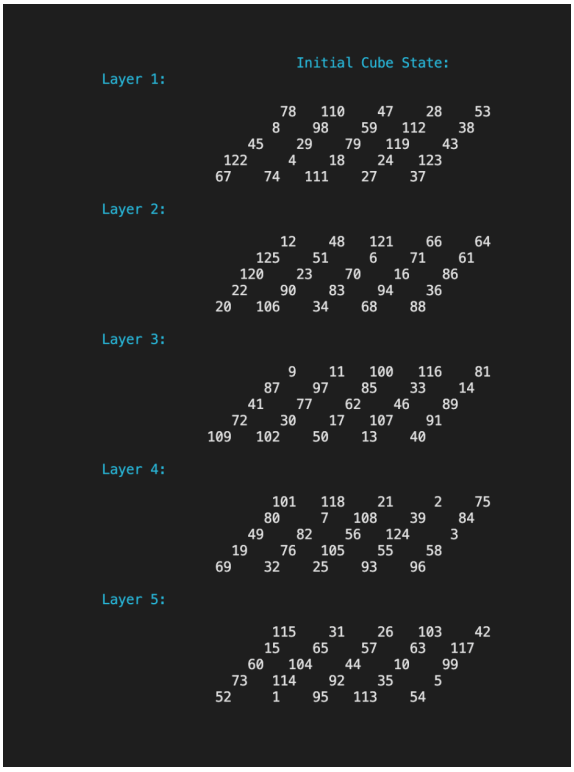
State Awal	
Objective Awal	6113

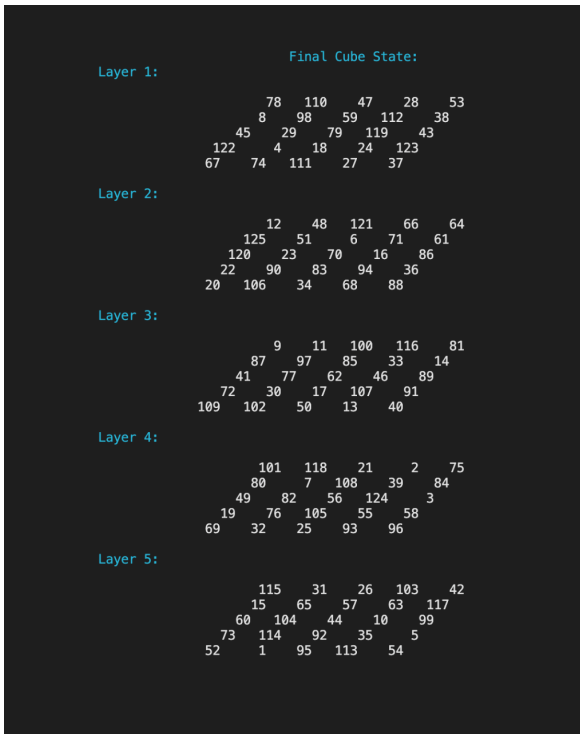
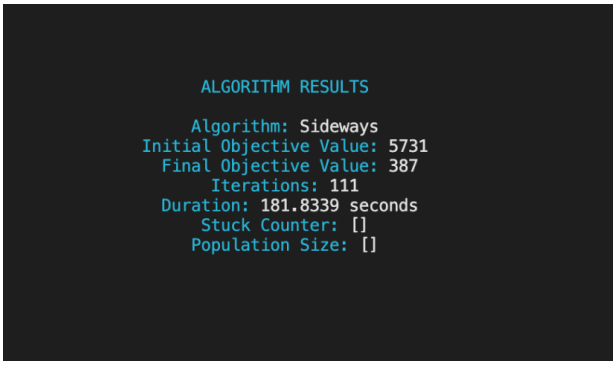
State Akhir	
Objective Akhir	267
Banyak Iterasi	121
Hasil Eksperimen	
Durasi Pencarian	197.2577 detik

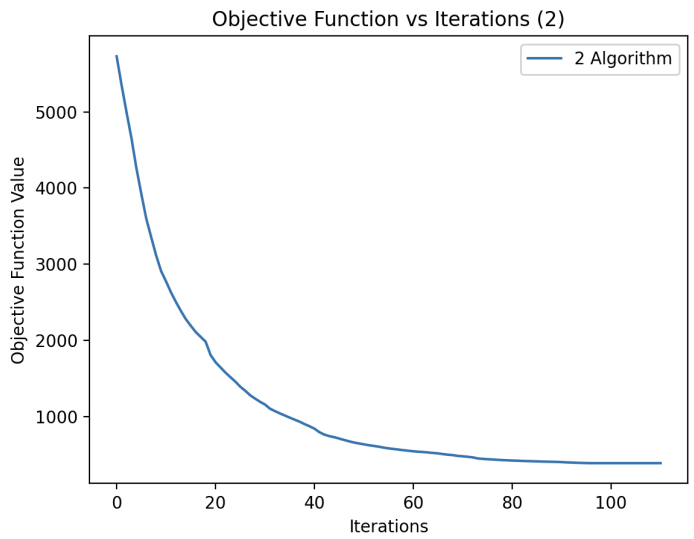
Plot Banyak Iterasi terhadap Objective Function	

3. Pengujian 3 - Iterasi Max Sideways 15

Tabel 4.1.2.3. Pengujian 3 Hill Climbing with Sideways Move

State Awal	

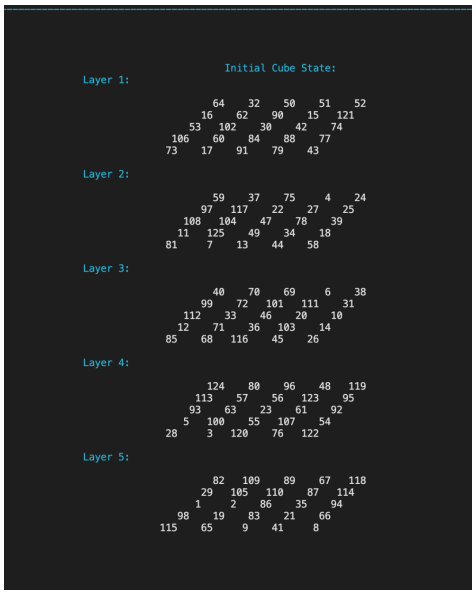
Objective Awal	5731
State Akhir	
Objective Akhir	387
Banyak Iterasi	111
Hasil Eksperimen	
Durasi Pencarian	181.8339 detik

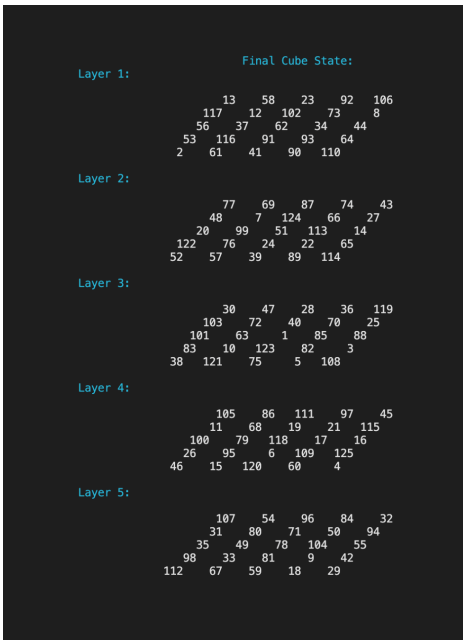
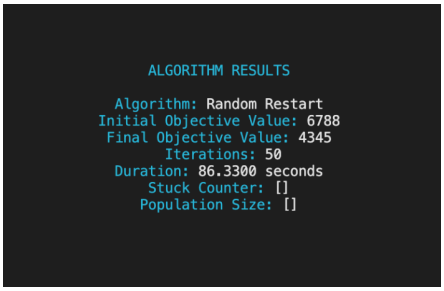
Plot Banyak Iterasi terhadap Objective Function	

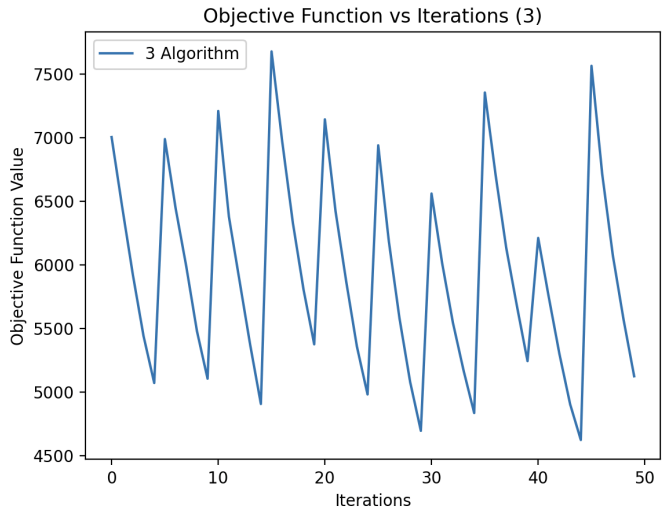
4.1.3 Random Restart Hill Climbing

1. Pengujian 1 - Restart 10 dengan 5 Iterasi

Tabel 4.1.3.1. Pengujian 1 Random Restart Hill Climbing

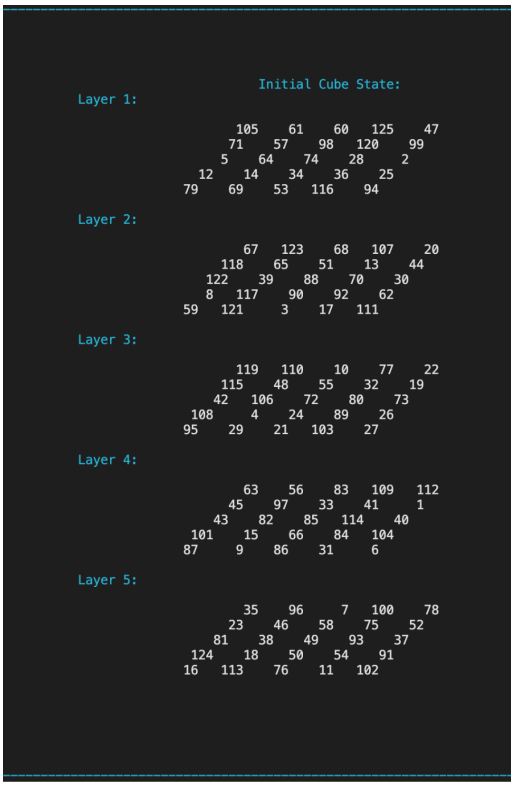
State Awal	
Objective Awal	6788

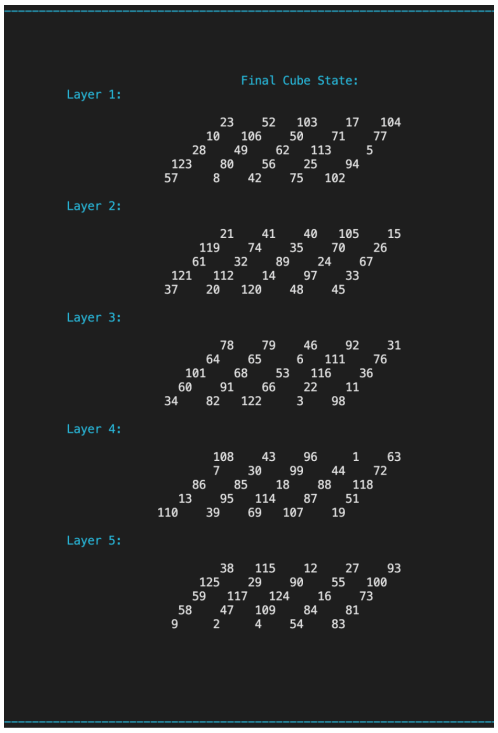
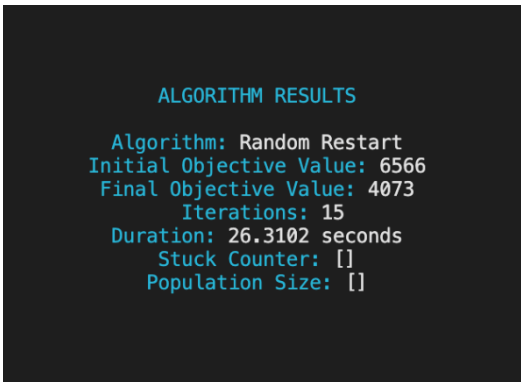
State Akhir	
Objective Akhir	4345
Banyak Iterasi	50
Hasil Eksperimen	
Durasi Pencarian	86.3300 detik

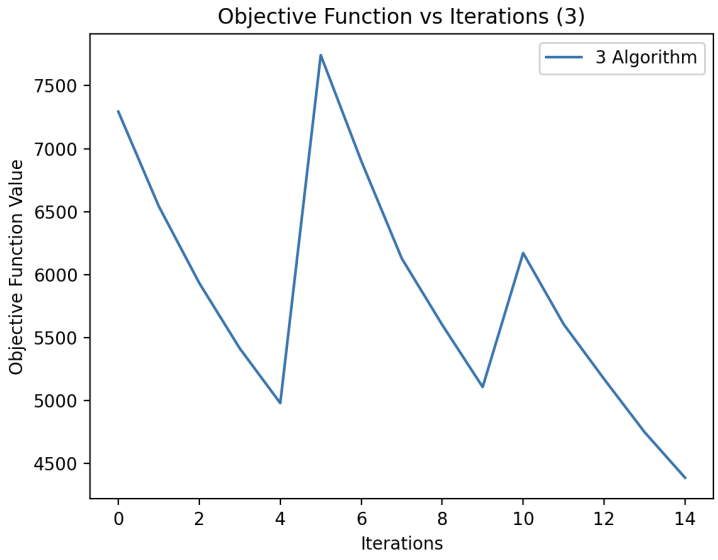
Plot Banyak Iterasi terhadap Objective Function	

2. Pengujian 2 - Restart 3 dengan 5 Iterasi

Tabel 4.1.3.2. Pengujian 2 Random Restart Hill Climbing

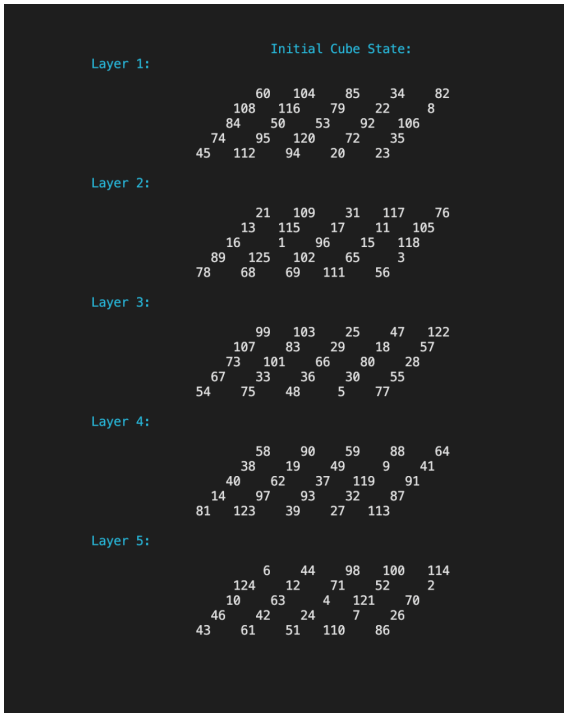
State Awal	

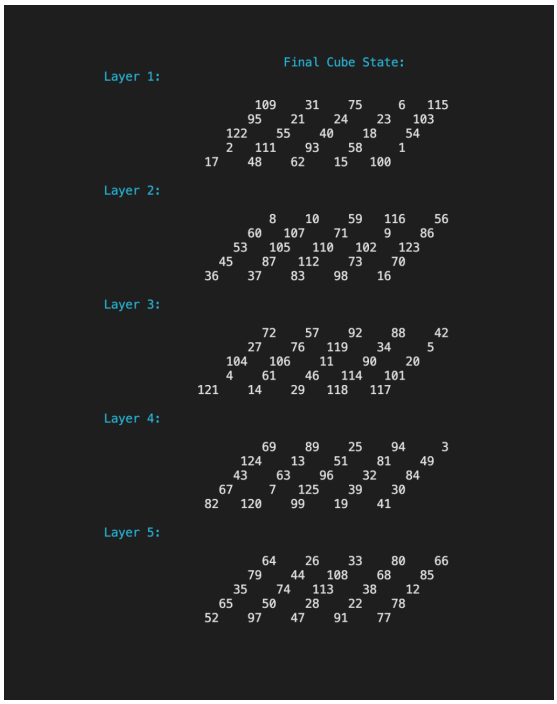
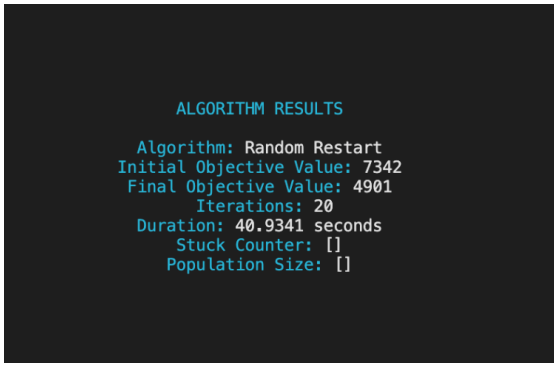
Objective Awal	6566
State Akhir	
Objective Akhir	4073
Banyak Iterasi	15
Hasil Eksperimen	
Durasi Pencarian	26.3102

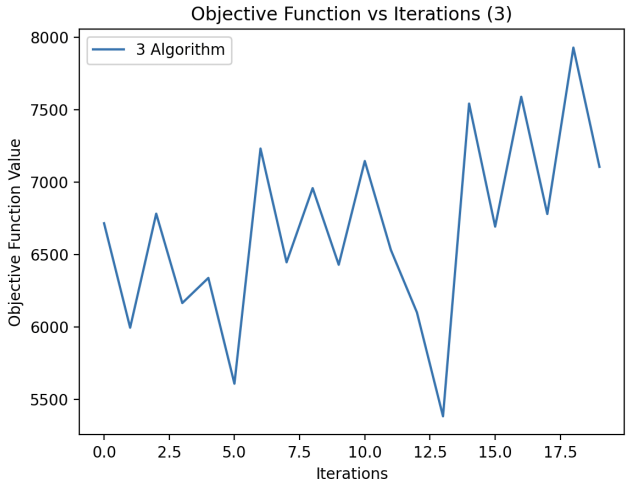
Plot Banyak Iterasi terhadap Objective Function	

3. Pengujian 3 - Restart 10 dengan 2 Iterasi

Tabel 4.1.3.3. Pengujian 3 Random Restart Hill Climbing

State Awal	
Objective Awal	7342

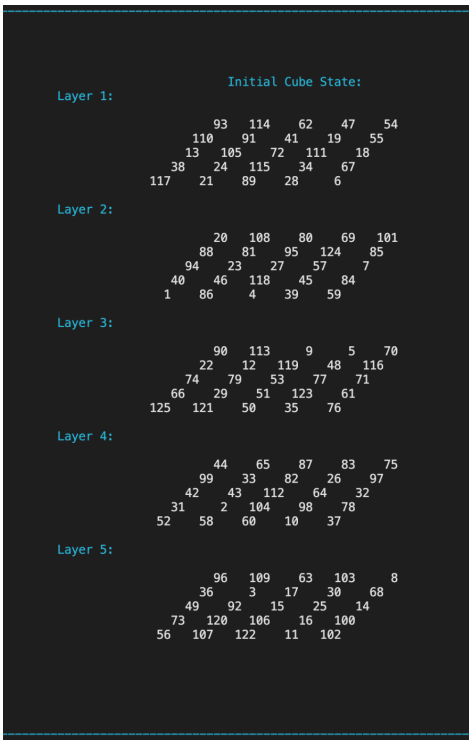
State Akhir	
Objective Akhir	4901
Banyak Iterasi	20
Hasil Eksperimen	
Durasi Pencarian	40.9341 detik

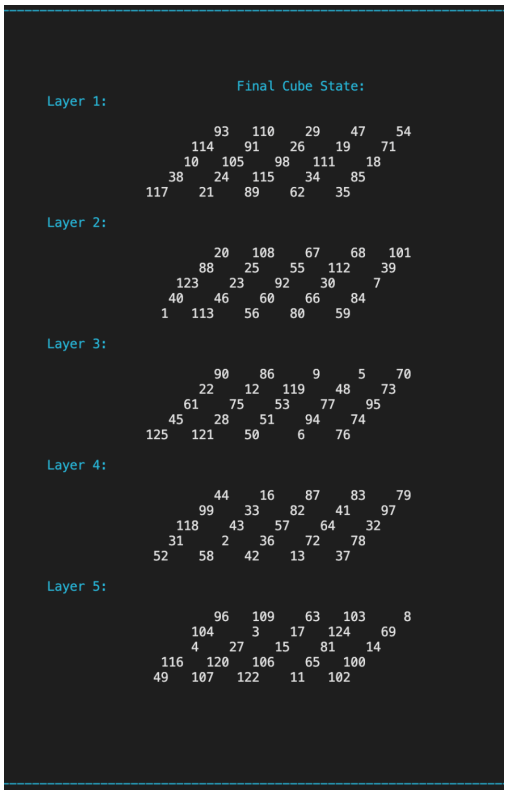
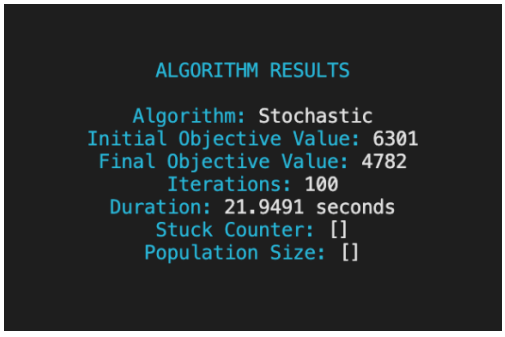
Plot Banyak Iterasi terhadap Objective Function	

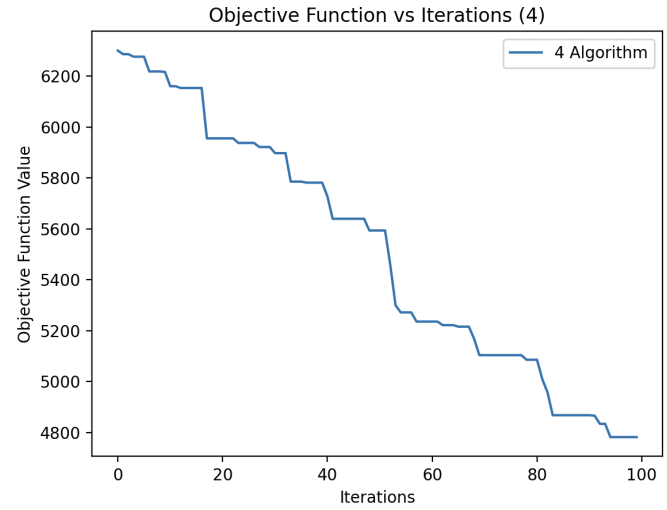
4.1.4 Stochastic Hill Climbing

1. Pengujian 1 - Iterasi 100

Tabel 4.1.4.1. Pengujian 1 Stochastic Hill Climbing

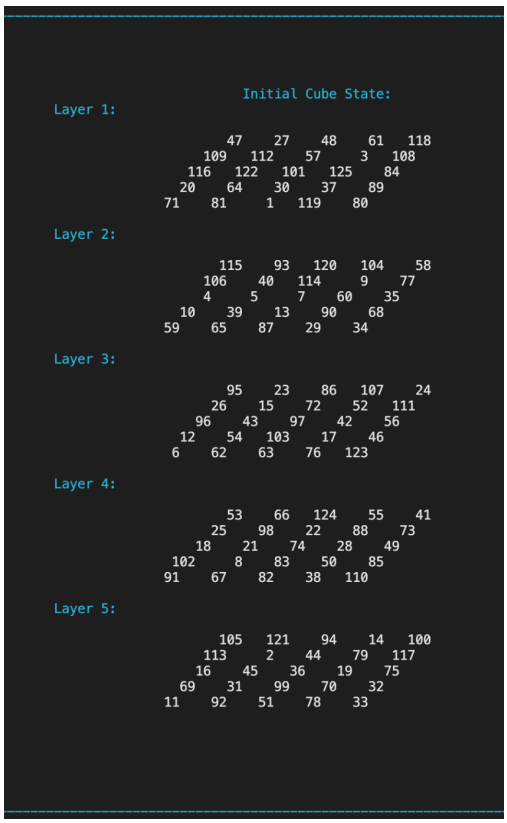
State Awal	
Objective Awal	6301

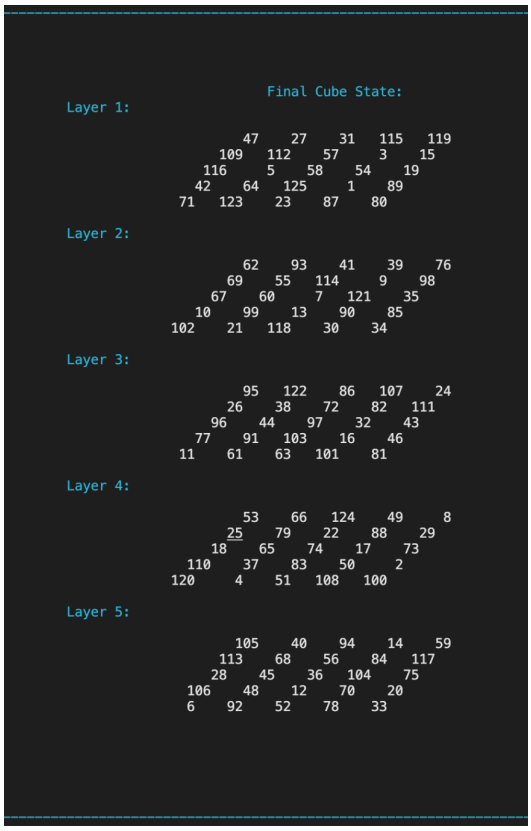
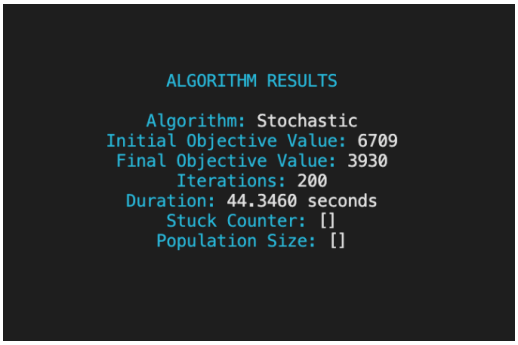
State Akhir	
Objective Akhir	4782
Banyak Iterasi	100
Hasil Eksperimen	
Durasi Pencarian	21.9491

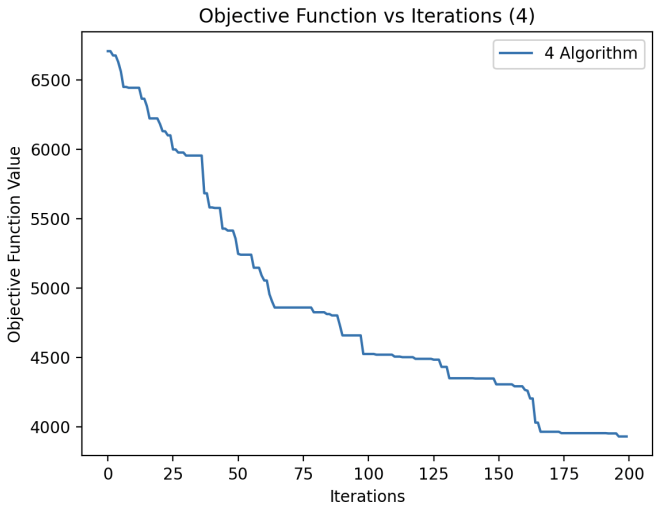
Plot Banyak Iterasi terhadap Objective Function	

2. Pengujian 2 - Iterasi 200

Tabel 4.1.4.2. Pengujian 2 Stochastic

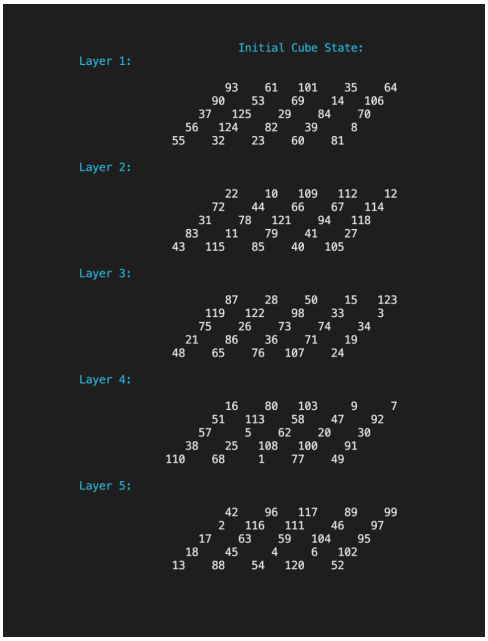
State Awal	

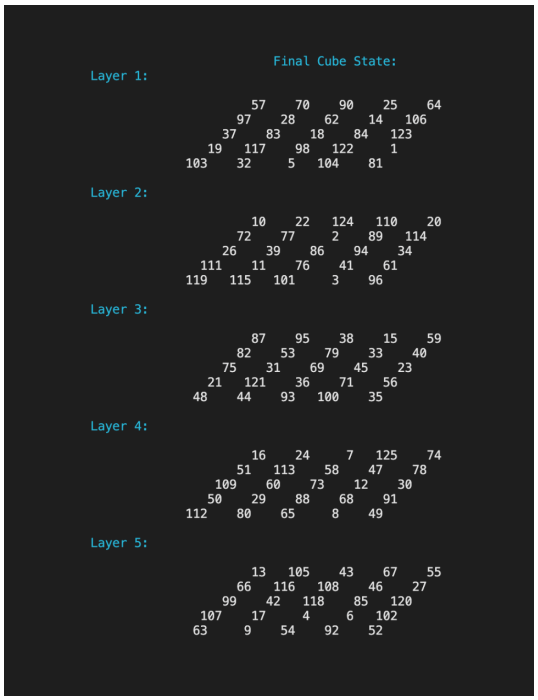
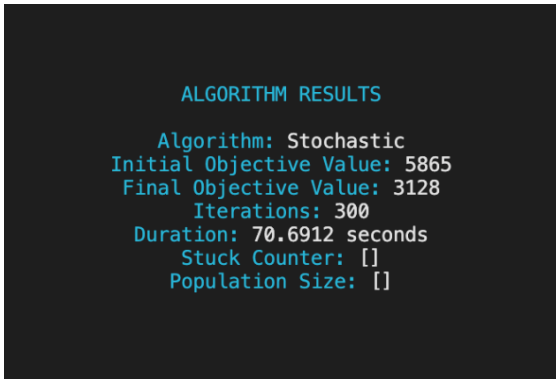
Objective Awal	6709
State Akhir	
Objective Akhir	3930
Banyak Iterasi	200
Hasil Eksperimen	
Durasi Pencarian	44.3460

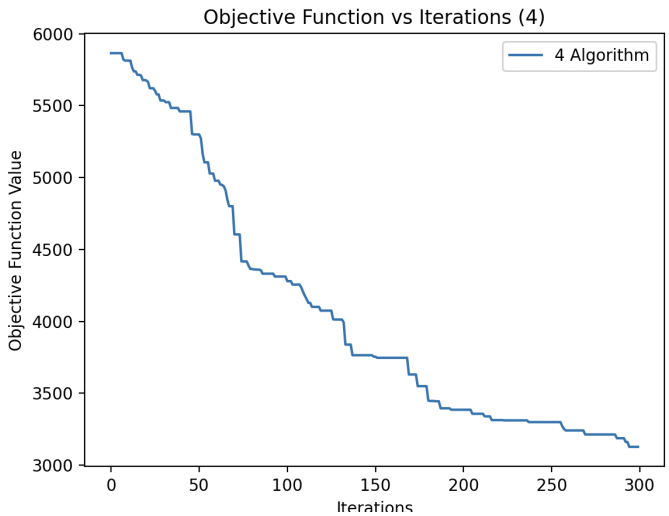
Plot Banyak Iterasi terhadap Objective Function	

3. Pengujian 3 - Iterasi 300

Tabel 4.1.4.3. Pengujian 3 Stochastic

State Awal	
Objective Awal	5865

State Akhir	
Objective Akhir	3128
Banyak Iterasi	300
Hasil Eksperimen	
Durasi Pencarian	70.6912 detik

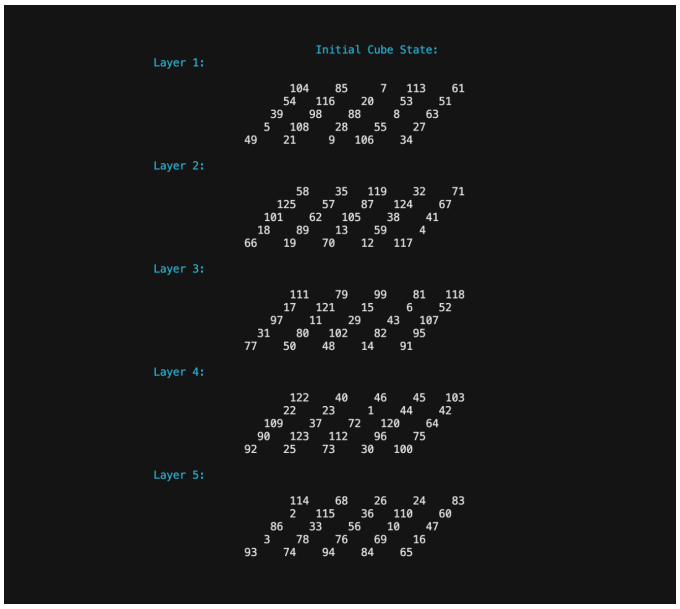
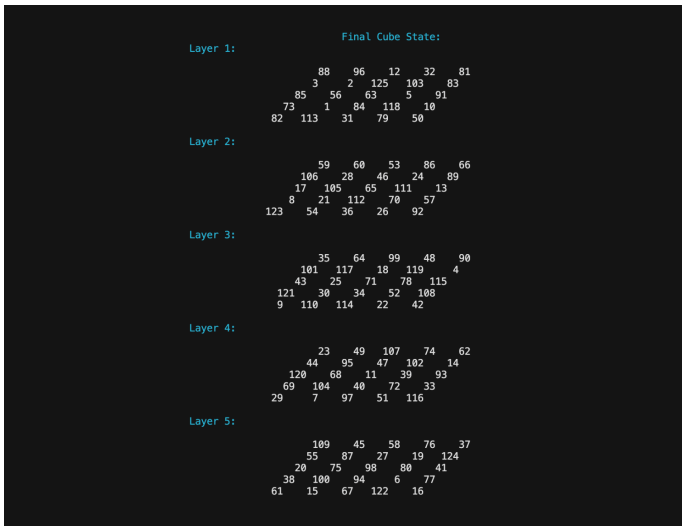
Plot Banyak Iterasi terhadap Objective Function	

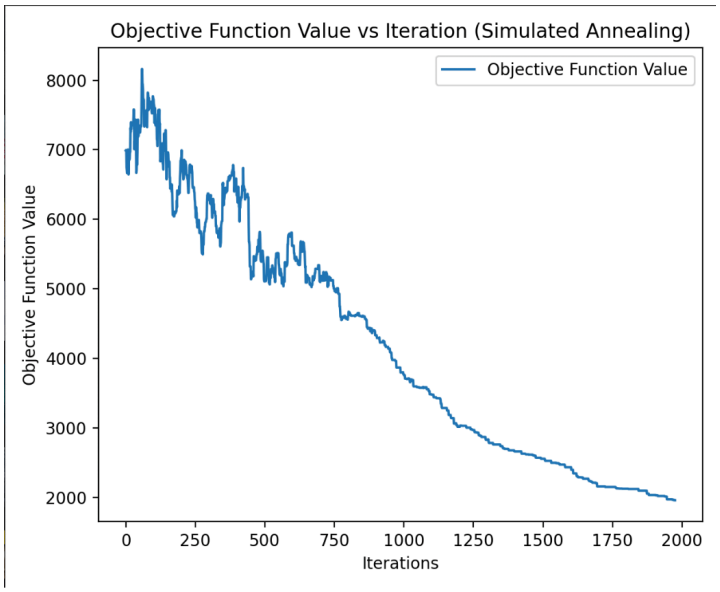
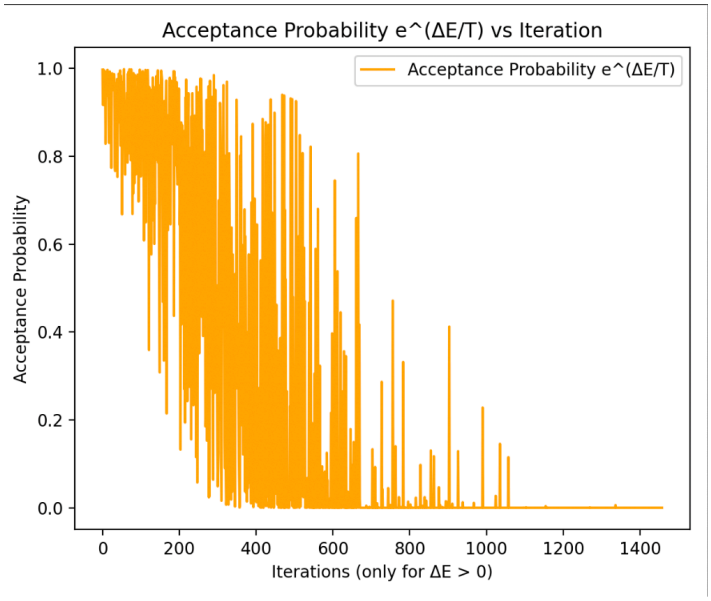
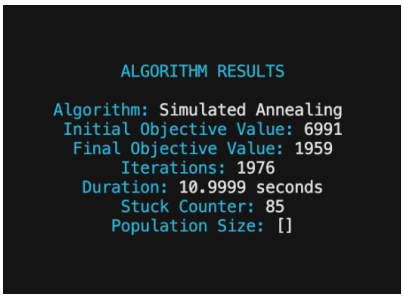
4.2 Simulated Annealing

1. Pengujian 1

Tabel 4.2.1. Pengujian 1 Simulated Annealing

Temperatur awal	2000
Cooling rate	0.995
Minimum temperature	0.1
Iterasi maksimum	10000

State Awal	
Objective Awal	6991
State Akhir	1959
Objective Akhir	

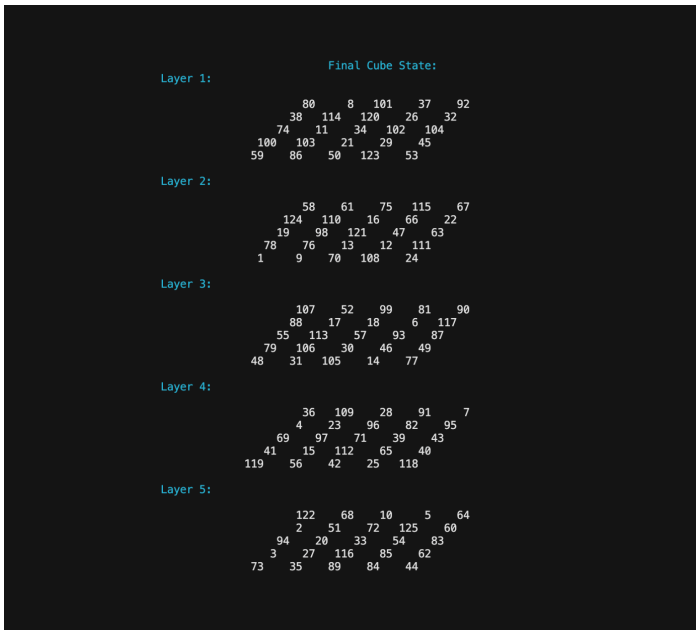
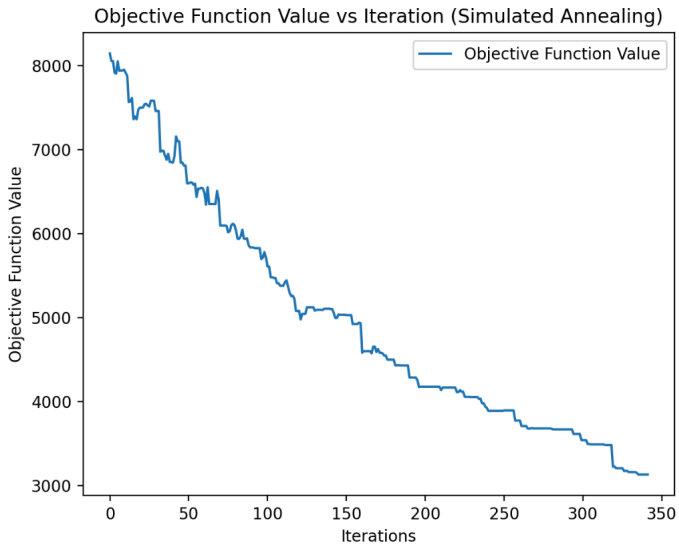
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	
Plot $e^{\Delta E/T}$ terhadap banyak iterasi	
Hasil Eksperimen	

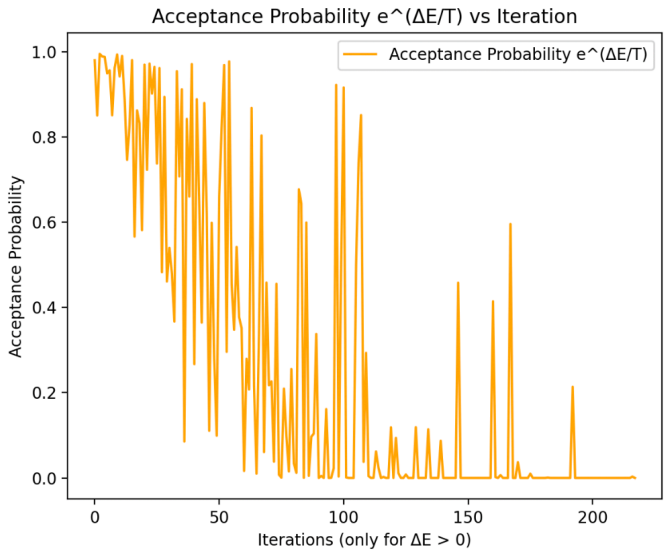
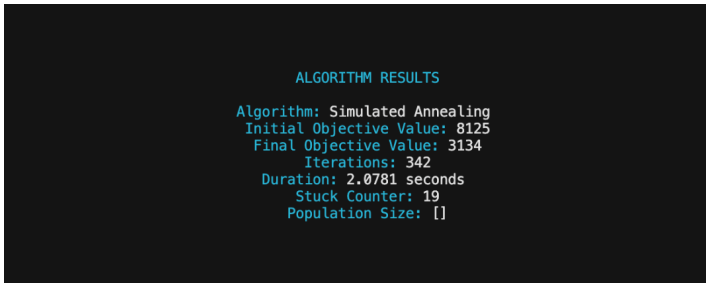
Banyak Iterasi	1976
Durasi Pencarian	10.999 detik
Jumlah Stuck	85

2. Pengujian 2

Tabel 4.2.2. Pengujian 2 Simulated Annealing

Temperatur awal	1000
Cooling rate	0.98
Minimum temperature	1
Iterasi maksimum	10000
State Awal	
Objective Awal	8125

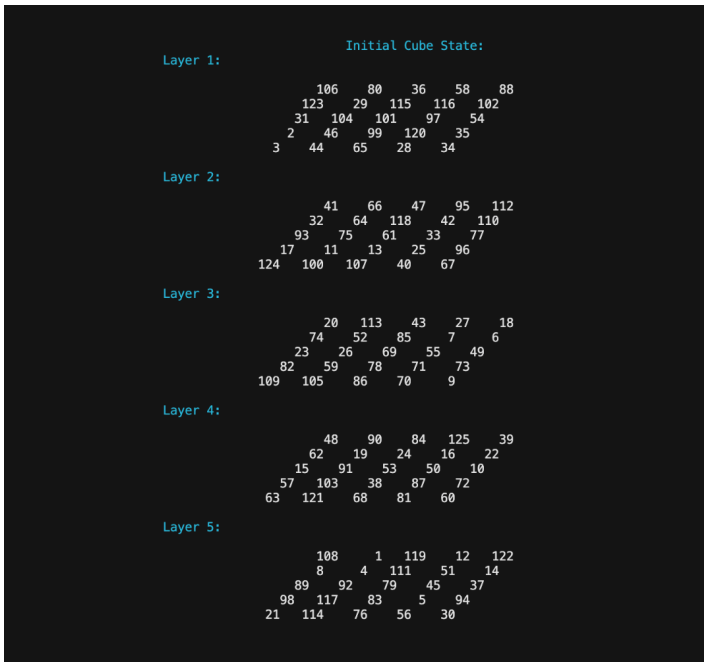
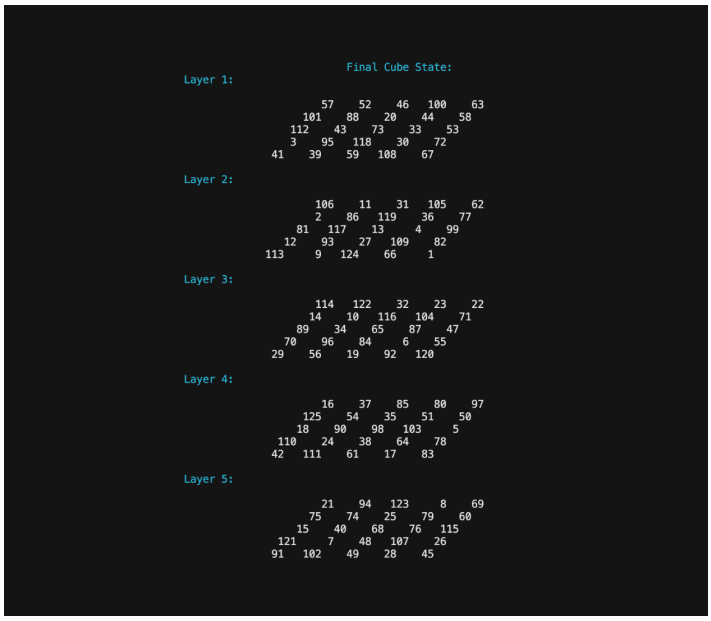
State Akhir	
Objective Akhir	3134
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	

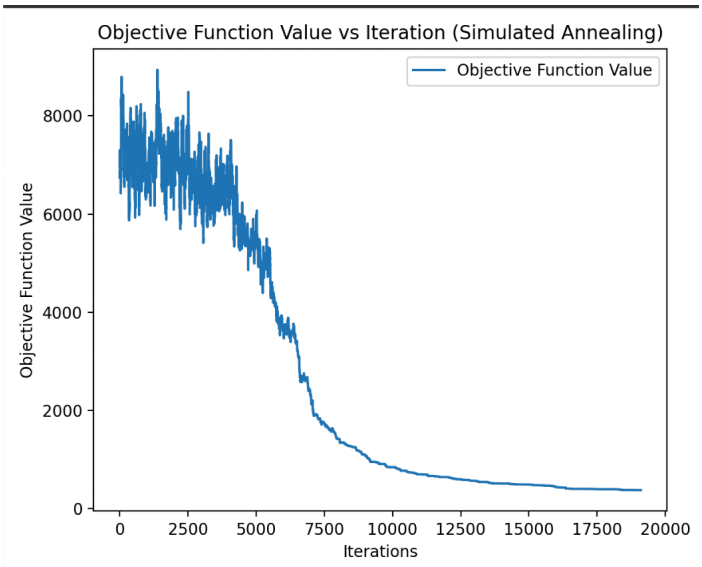
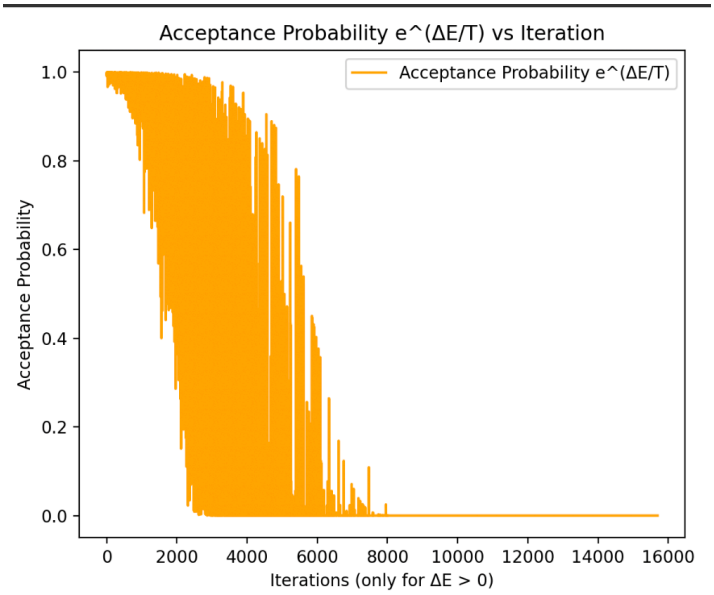
Plot $e^{\Delta E/T}$ terhadap banyak iterasi	
Hasil Eksperimen	
Banyak Iterasi	342
Durasi Pencarian	2.0781
Jumlah Stuck	19

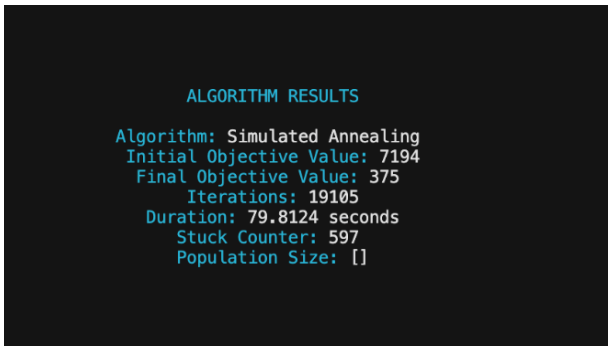
3. Pengujian 3

Tabel 4.2.3. Pengujian 3 Simulated Annealing

Temperatur awal	20000
Cooling rate	0.999
Minimum temperature	0.0001

Iterasi maksimum	50000
State Awal	
Objective Awal	7194
State Akhir	
Objective Akhir	375

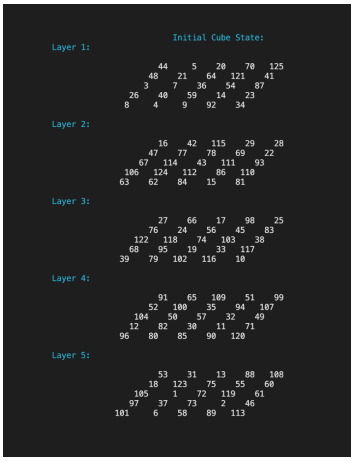
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	
Plot $e^{\Delta E/T}$ terhadap banyak iterasi	

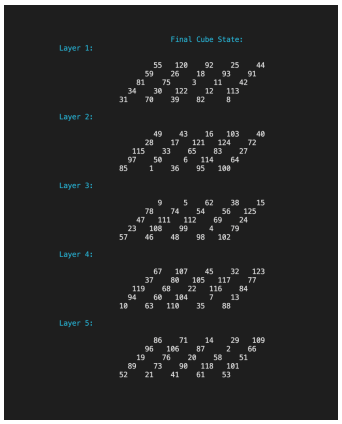
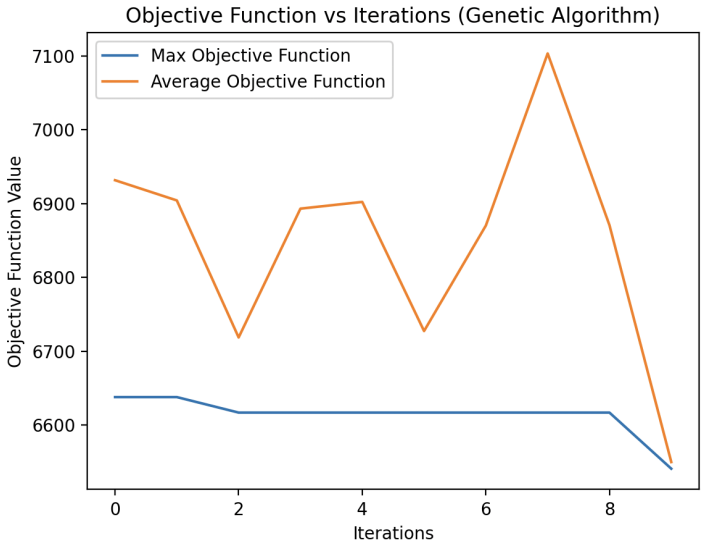
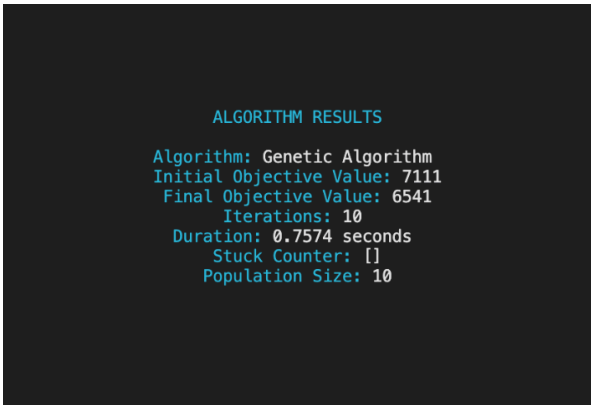
Hasil Eksperimen	
Banyak Iterasi	19105
Durasi Pencarian	79.8124 detik
Jumlah Stuck	597

4.3 Genetic Algorithm

1. Pengujian 1 - Populasi sebagai Kontrol dengan 10 Iterasi

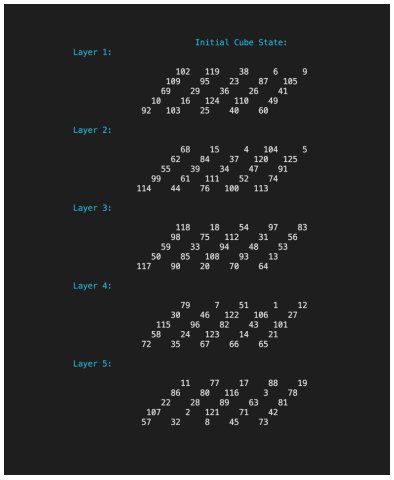
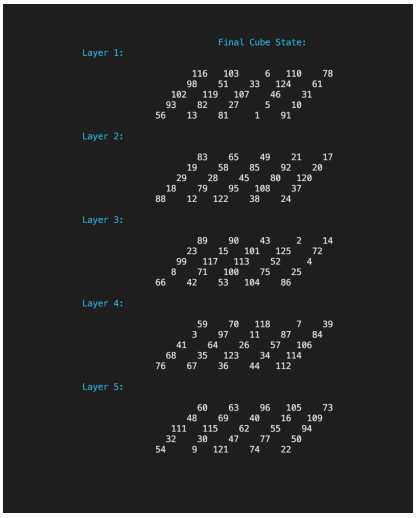
Tabel 4.3.1. Pengujian 1 Populasi sebagai Kontrol

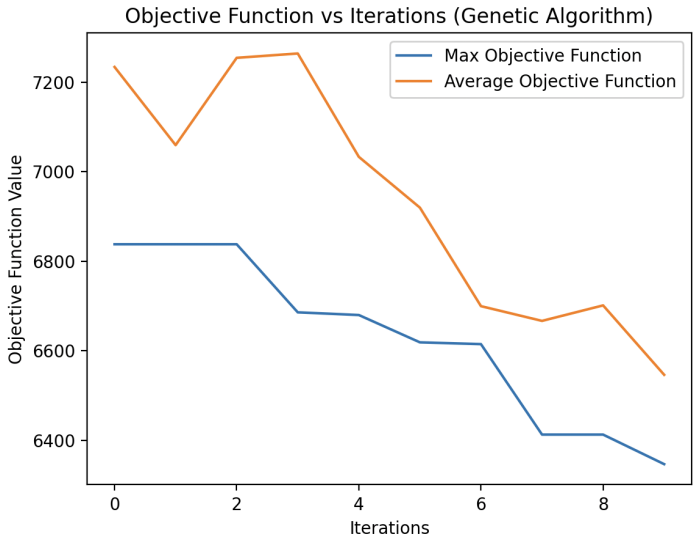
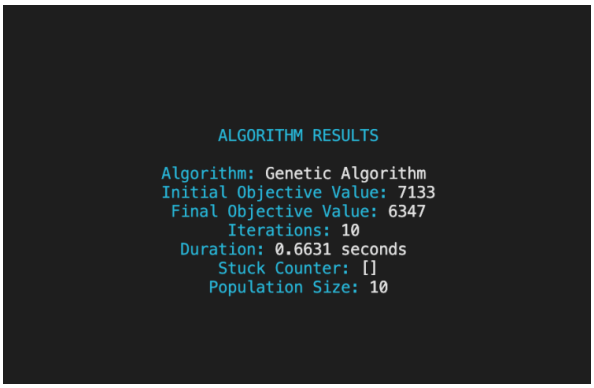
State Awal	
Objective Awal	7111

State Akhir	
Objective Akhir	6541
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	
Hasil Eksperimen	

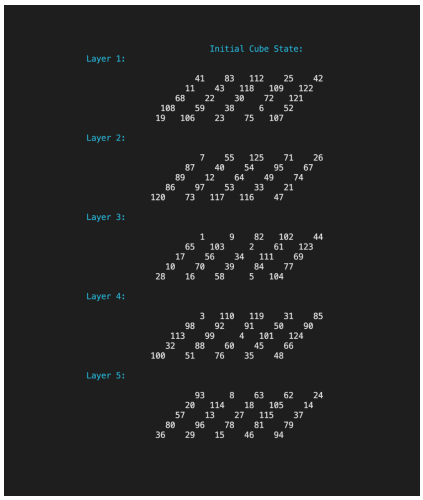
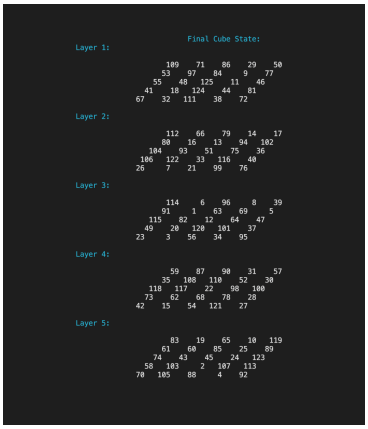
Jumlah Populasi	10
Banyak Iterasi	10
Durasi Pencarian	0.7574 detik

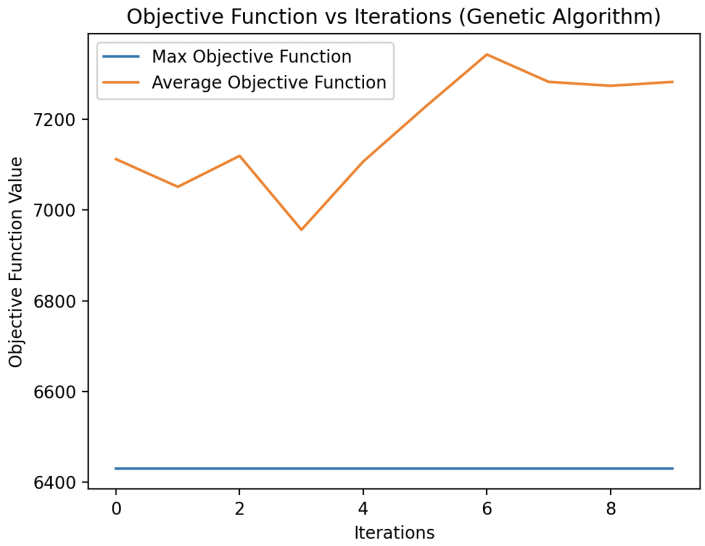
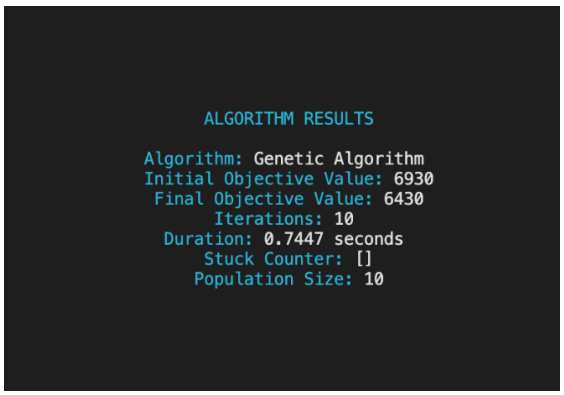
Tabel 4.3.2. Pengujian 1 Populasi sebagai Kontrol

State Awal	
Objective Awal	7133
State Akhir	
Objective Akhir	6347

Plot nilai objective function terhadap banyak iterasi yang telah dilewati	
Hasil Eksperimen	
Jumlah Populasi	10
Banyak Iterasi	10
Durasi Pencarian	0.6631 detik

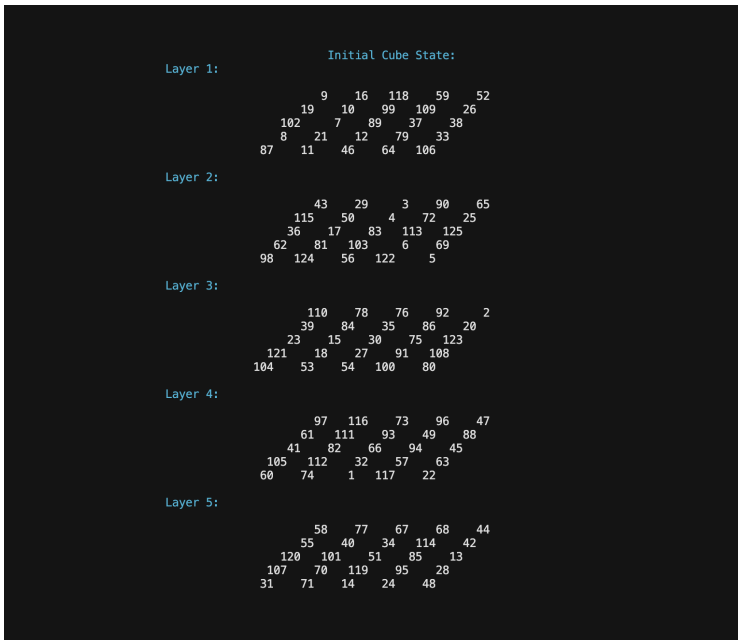
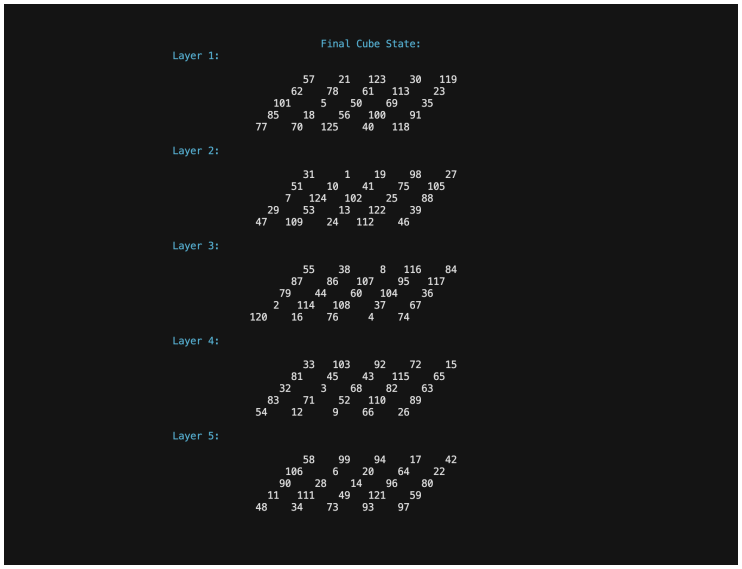
Tabel 4.3.3. Pengujian 1 Populasi sebagai Kontrol

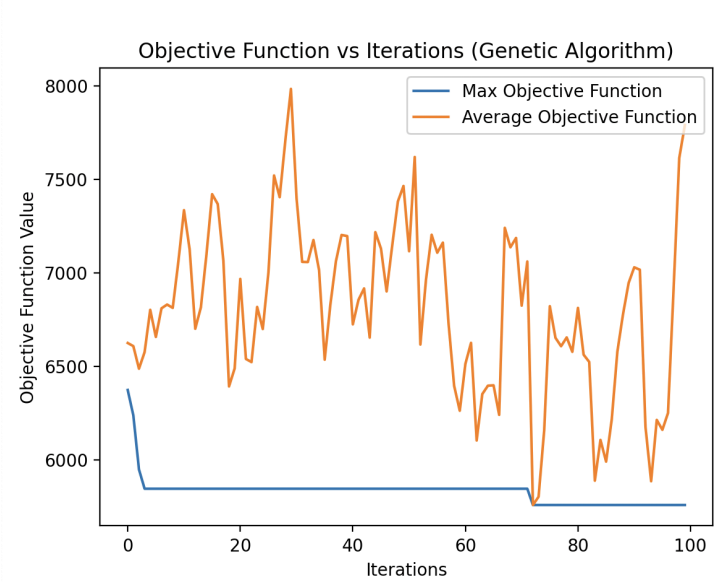
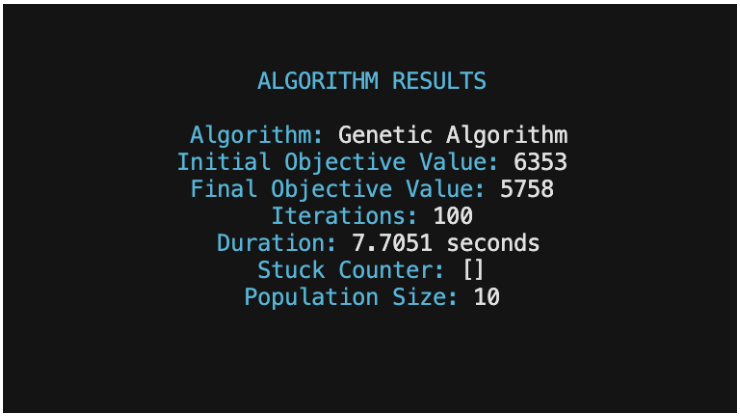
State Awal	
Objective Awal	6930
State Akhir	
Objective Akhir	6430

Plot nilai objective function terhadap banyak iterasi yang telah dilewati	
Hasil Eksperimen	
Jumlah Populasi	10
Banyak Iterasi	10
Durasi Pencarian	0.7447 detik

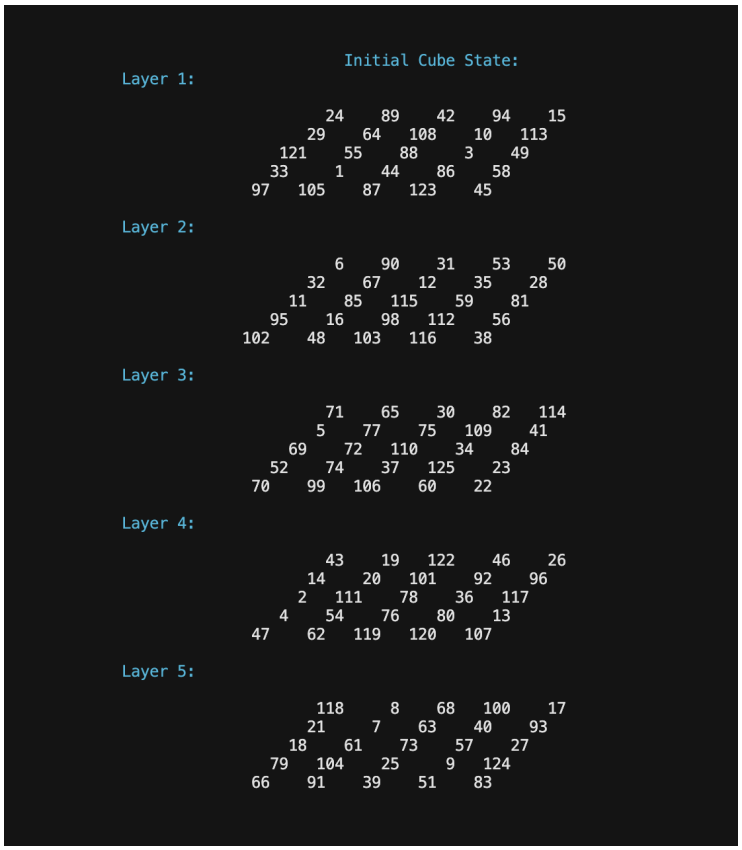
2. Pengujian 2 - Populasi sebagai Kontrol dengan 100 Iterasi

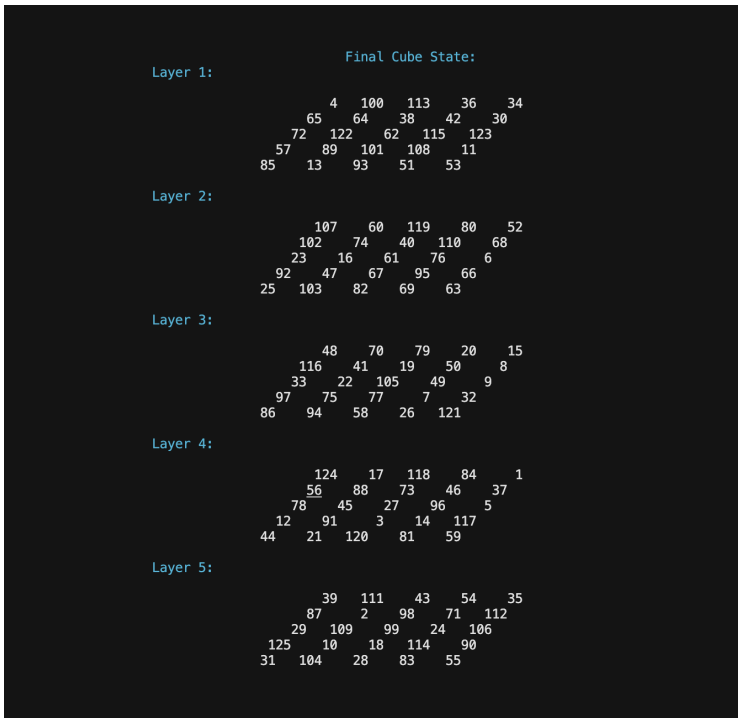
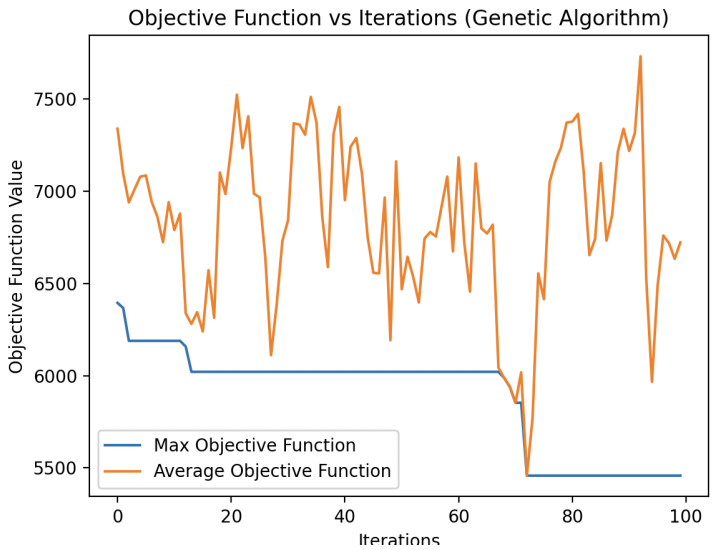
Tabel 4.3.4. Pengujian 2 Populasi sebagai Kontrol

State Awal	
Objective Awal	6353
State Akhir	
Objective Akhir	5758

Plot nilai objective function terhadap banyak iterasi yang telah dilewati	
Hasil Eksperimen	
Jumlah Populasi	10
Banyak Iterasi	100
Durasi Pencarian	7.7051 detik

Tabel 4.3.5. Pengujian 2 Populasi sebagai Kontrol

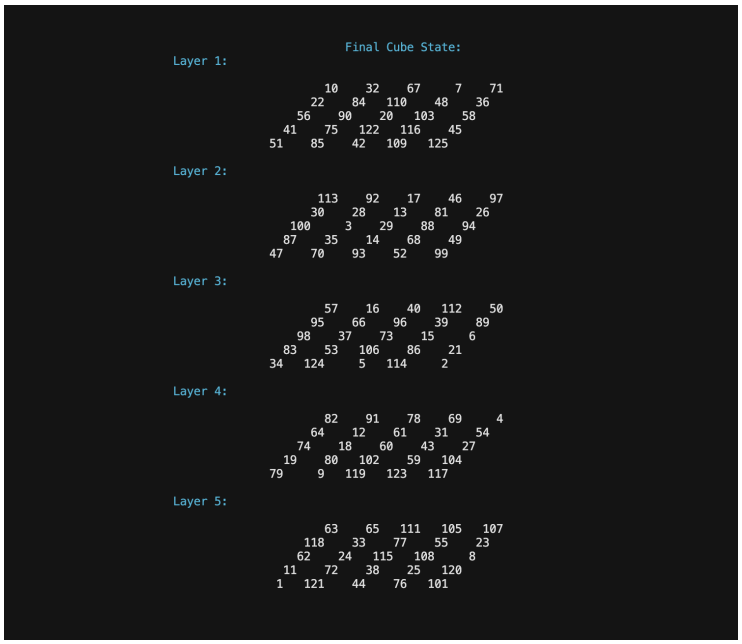
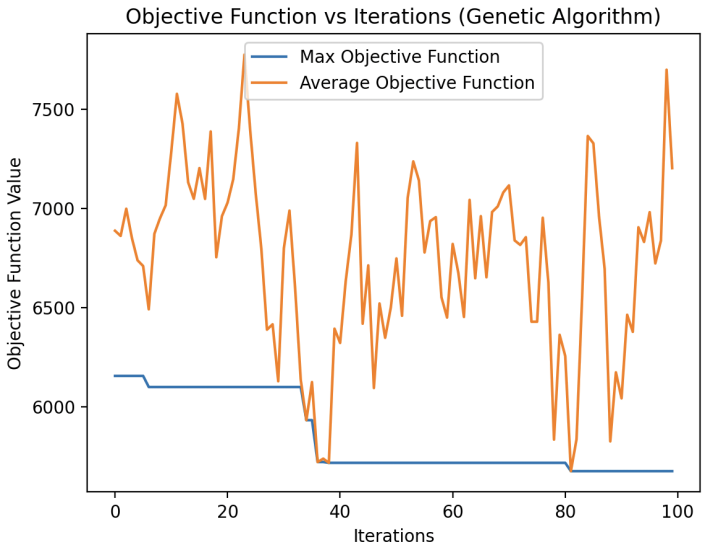
State Awal	
Objective Awal	6632

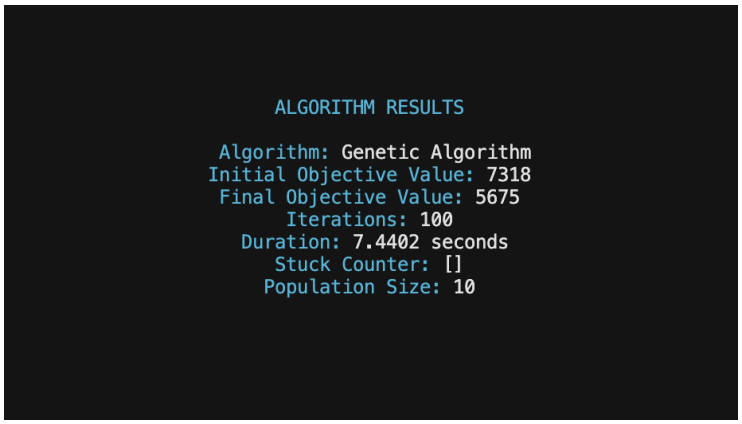
State Akhir	
Objective Akhir	5458
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	

Hasil Eksperimen	ALGORITHM RESULTS Algorithm: Genetic Algorithm Initial Objective Value: 6632 Final Objective Value: 5458 Iterations: 100 Duration: 7.5638 seconds Stuck Counter: [] Population Size: 10
Jumlah Populasi	10
Banyak Iterasi	100
Durasi Pencarian	7.5683 detik

Tabel 4.3.6. Pengujian 2 Populasi sebagai Kontrol

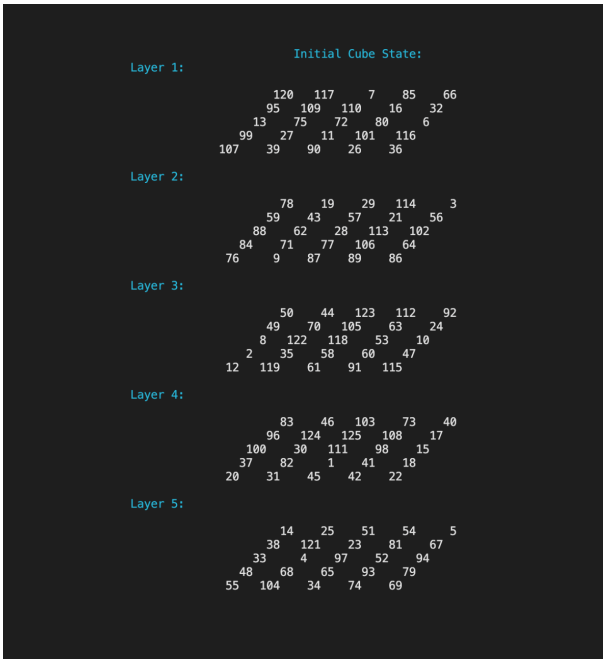
State Awal	Initial Cube State: Layer 1: 30 50 106 41 122 83 47 76 124 31 37 99 34 43 72 22 111 52 101 38 69 13 54 53 44 Layer 2: 14 57 25 40 79 66 107 1 39 70 24 112 63 117 75 121 77 46 118 10 87 51 81 119 89 Layer 3: 71 11 59 114 4 74 61 29 91 18 115 58 26 96 78 113 116 90 21 64 32 42 55 110 82 Layer 4: 45 80 100 5 97 68 104 67 84 48 35 2 108 8 60 7 9 95 56 88 103 109 92 102 94 Layer 5: 19 65 20 49 3 36 93 73 62 123 6 98 23 28 125 12 105 120 15 85 16 86 17 27 33
Objective Awal	7318

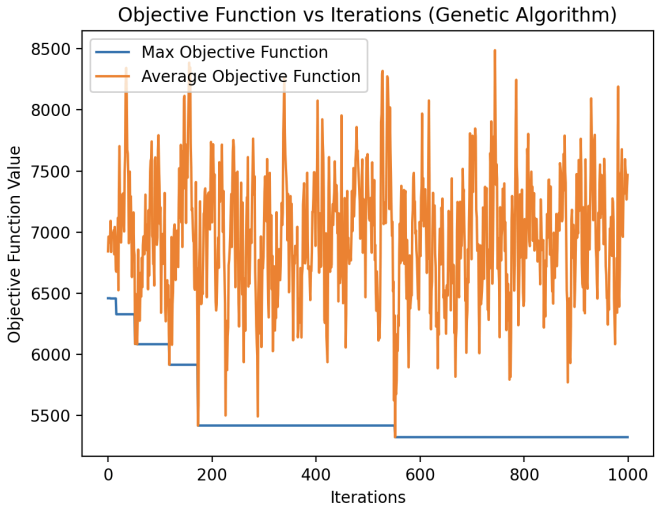
State Akhir	
Objective Akhir	5675
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	

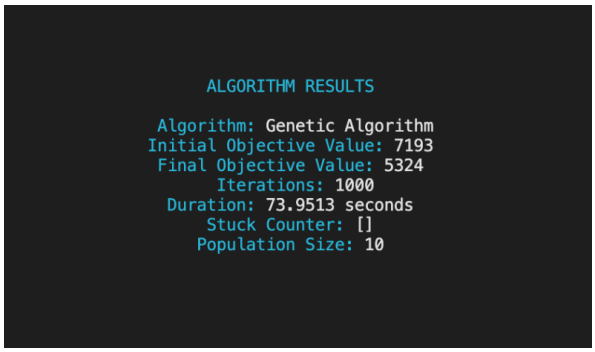
Hasil Eksperimen	
Jumlah Populasi	10
Banyak Iterasi	100
Durasi Pencarian	7.4402 detik

3. Pengujian 3 - Populasi sebagai Kontrol dengan 1000 Iterasi

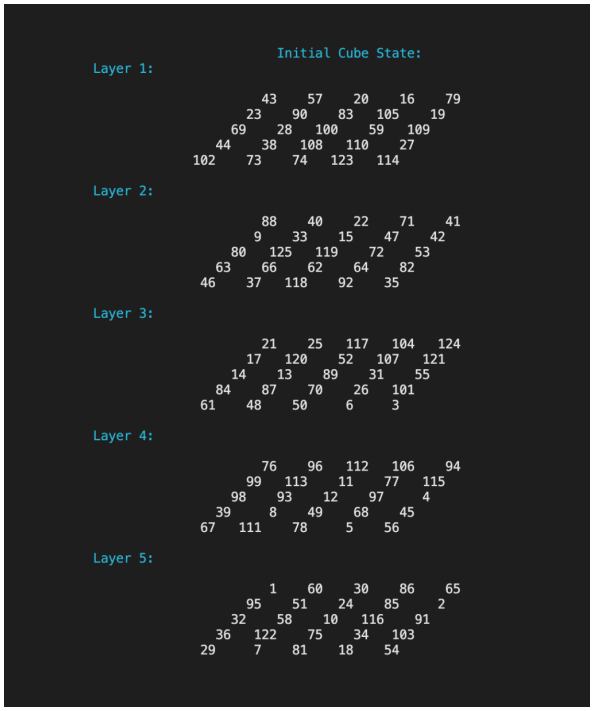
Tabel 4.3.7. Pengujian 3 Populasi sebagai Kontrol

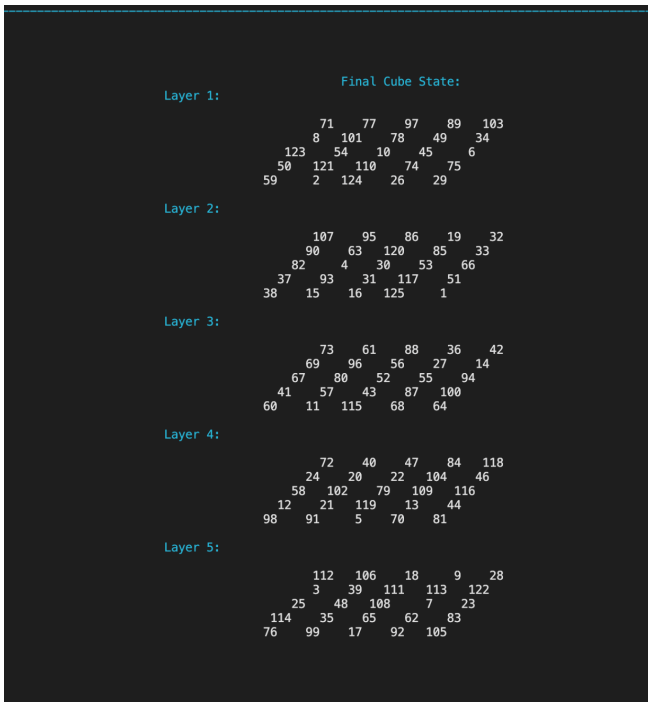
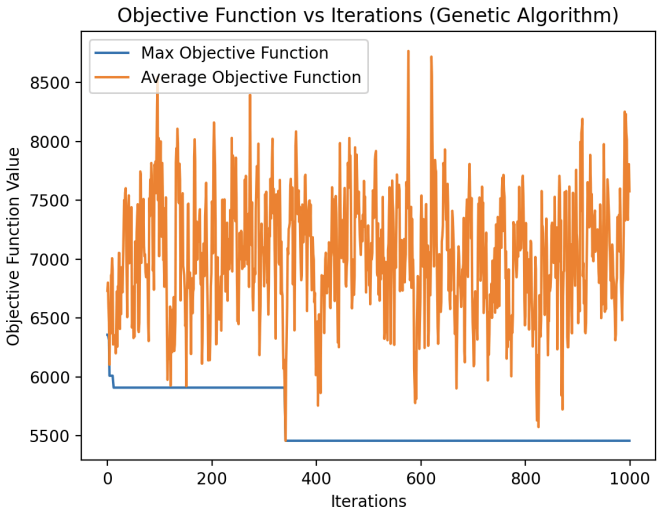
State Awal	
Objective Awal	7193

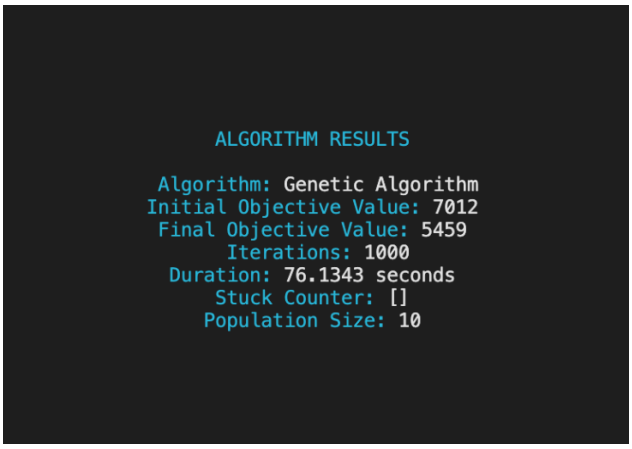
State Akhir	Final Cube State: Layer 1: 52 41 9 60 15 103 64 12 95 75 82 99 59 70 2 119 108 120 58 40 45 80 97 81 88 Layer 2: 90 123 3 61 71 46 62 115 11 31 34 20 100 125 26 19 21 53 8 13 67 102 5 18 109 Layer 3: 68 44 113 84 47 38 63 94 35 72 76 86 83 96 66 91 121 4 98 57 16 39 114 49 104 Layer 4: 87 111 118 54 6 122 110 1 124 56 22 25 27 33 23 32 107 17 101 106 14 92 65 10 69 Layer 5: 7 37 89 30 43 77 74 79 50 85 51 117 105 29 73 55 28 24 112 48 116 36 78 93 42
Objective Akhir	5324
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	

Hasil Eksperimen	
Jumlah Populasi	10
Banyak Iterasi	1000
Durasi Pencarian	73.9513 detik

Tabel 4.3.8. Pengujian 3 Populasi sebagai Kontrol

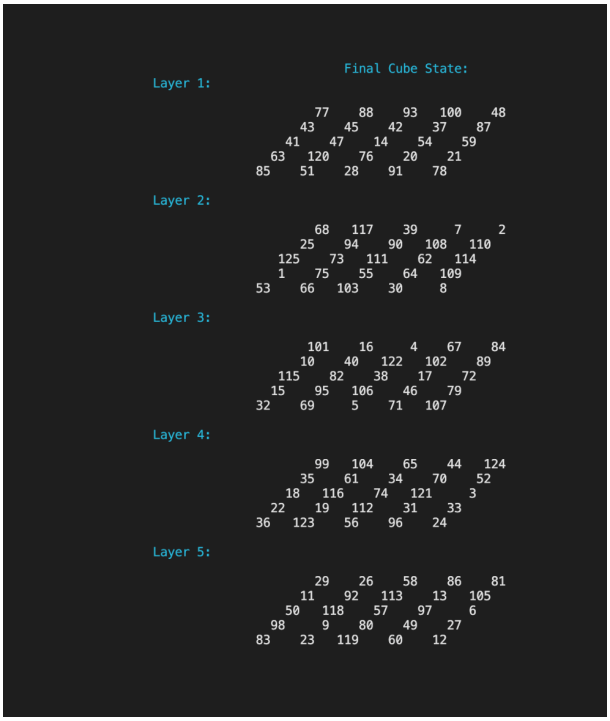
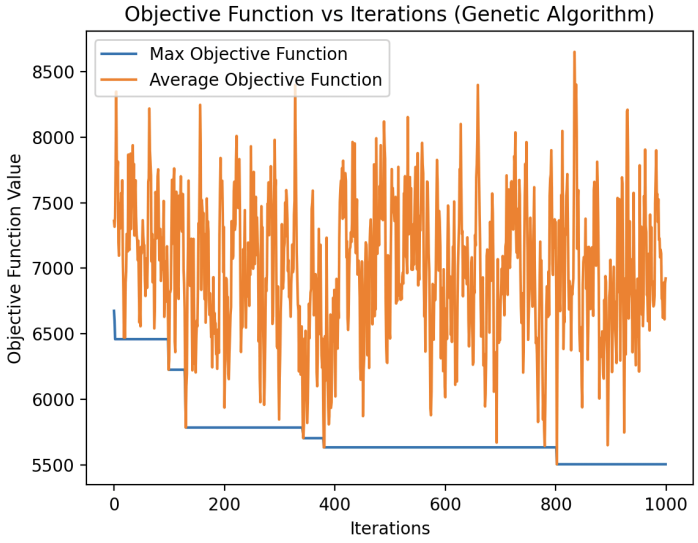
State Awal	
Objective Awal	7012

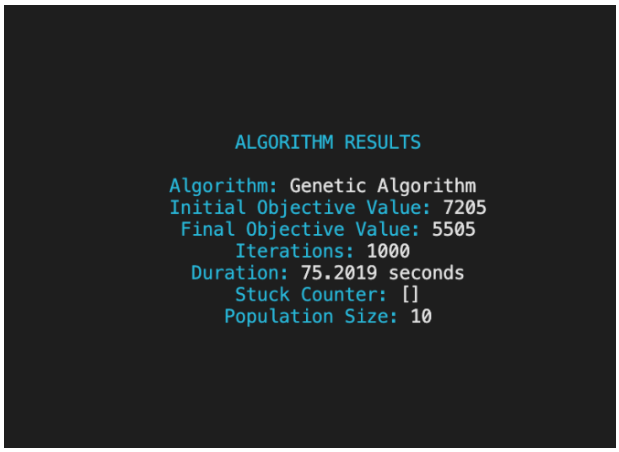
State Akhir	
Objective Akhir	5459
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	

Hasil Eksperimen	
Jumlah Populasi	10
Banyak Iterasi	1000
Durasi Pencarian	76.1343 detik

Tabel 4.3.9. Pengujian 3 Populasi sebagai Kontrol

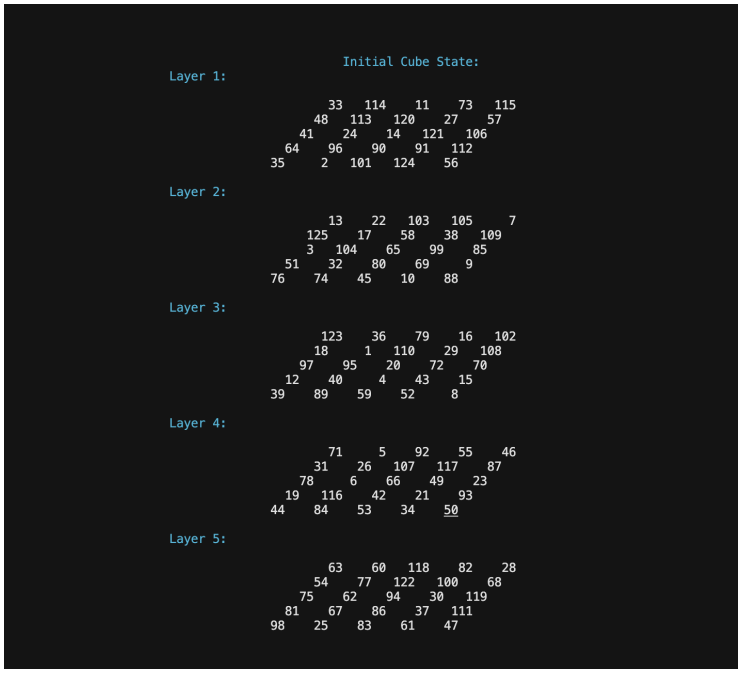
State Awal	

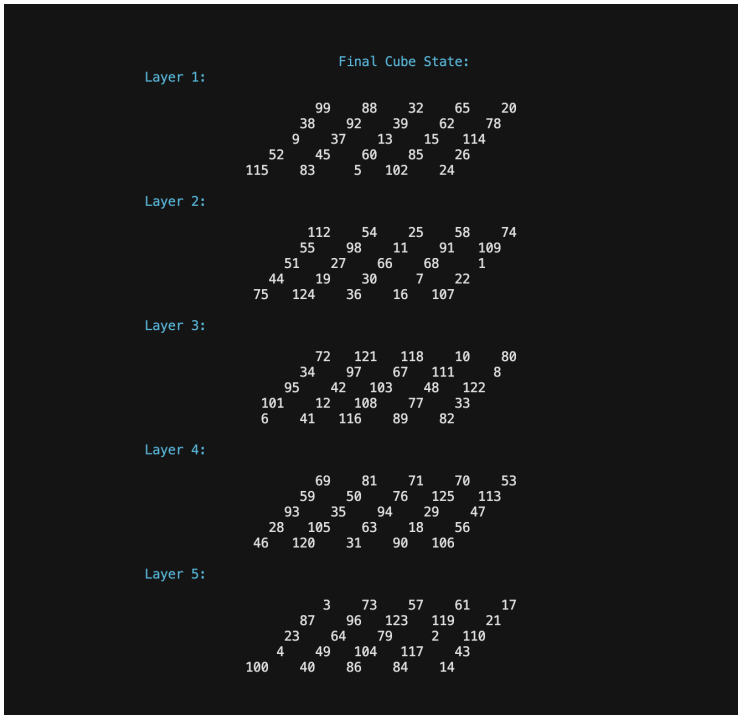
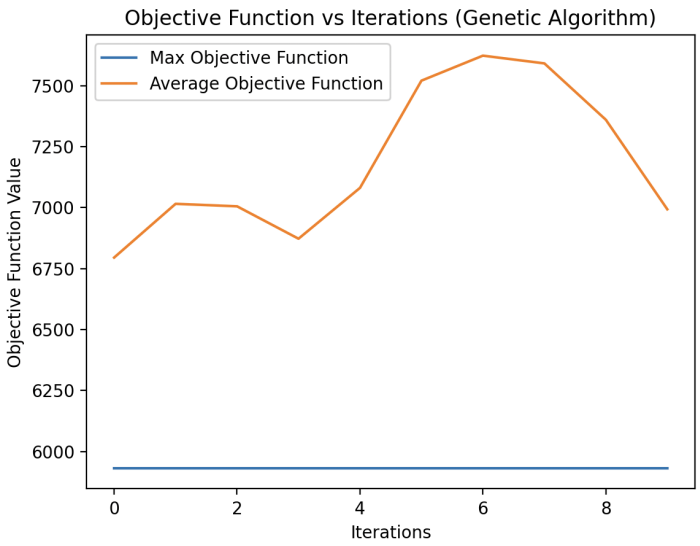
Objective Awal	7205
State Akhir	
Objective Akhir	5505
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	

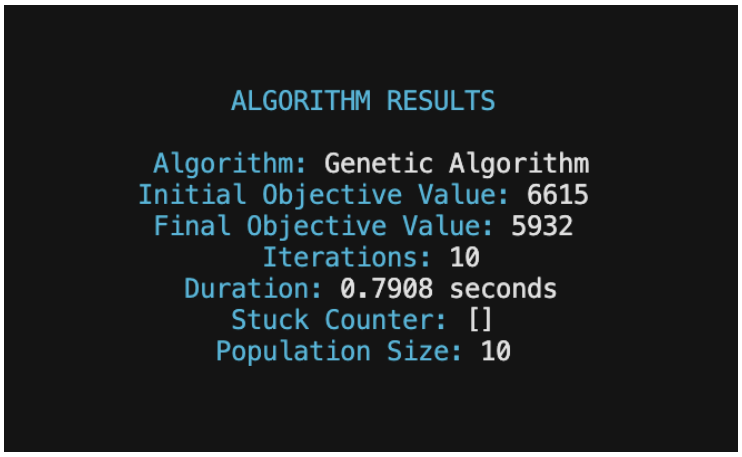
Hasil Eksperimen	
Jumlah Populasi	10
Banyak Iterasi	1000
Durasi Pencarian	75.2019 detik

4. Pengujian 4 - Iterasi sebagai Kontrol dengan 10 Populasi

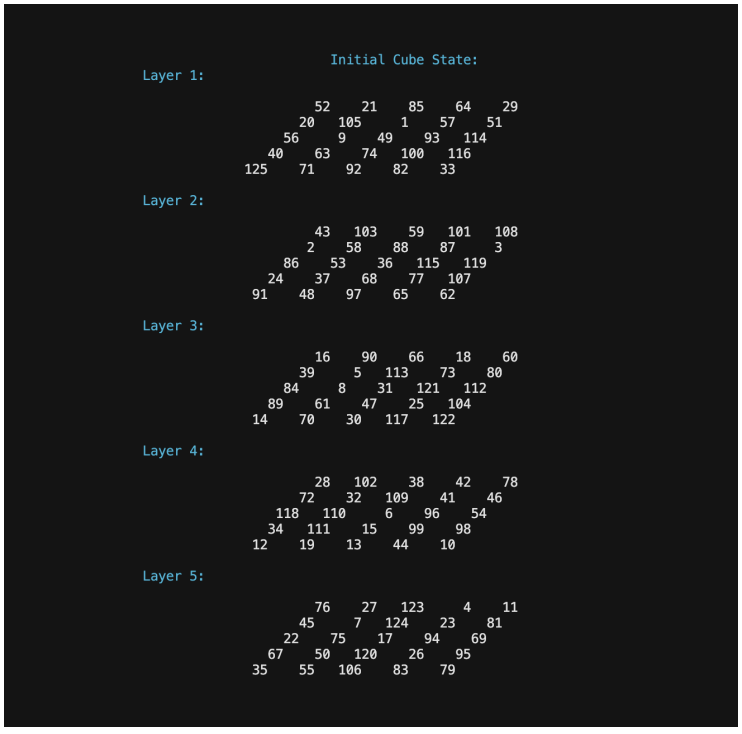
Tabel 4.3.10. Pengujian 4 Iterasi sebagai Kontrol

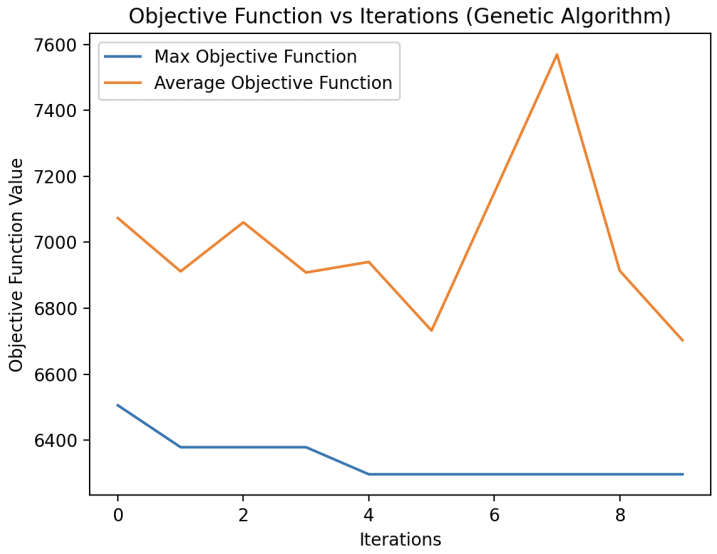
State Awal	

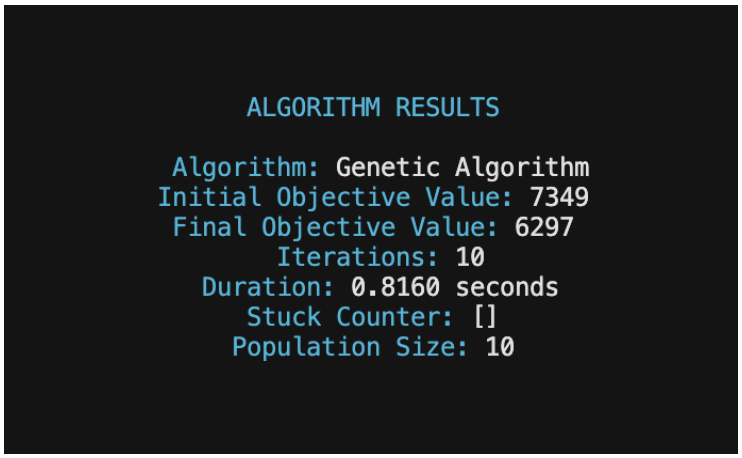
Objective Awal	6615
State Akhir	
Objective Akhir	5932
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	

Hasil Eksperimen	
Jumlah Populasi	10
Banyak Iterasi	10
Durasi Pencarian	0.7908 detik

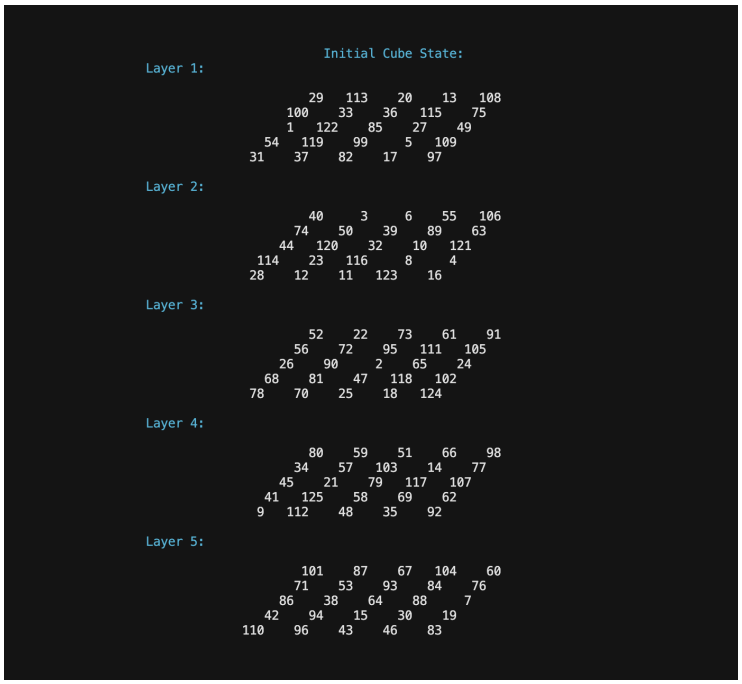
Tabel 4.3.11. Pengujian 4 Iterasi sebagai Kontrol

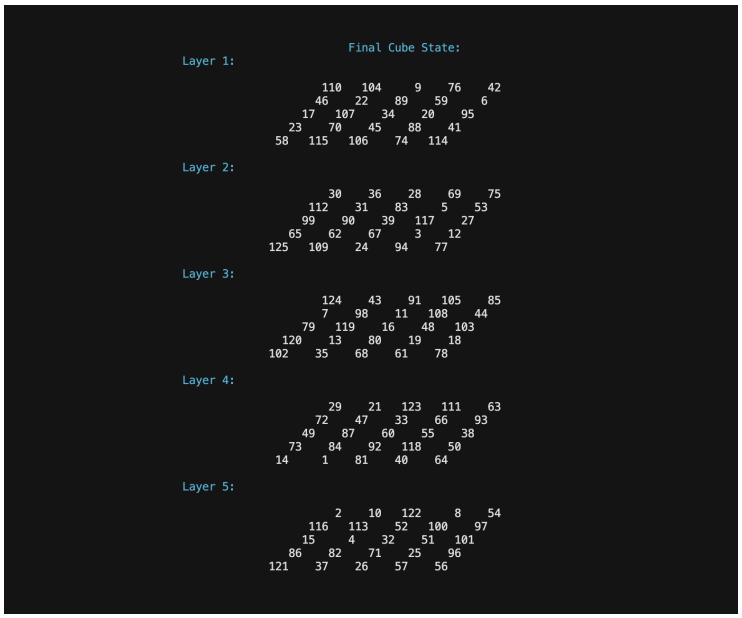
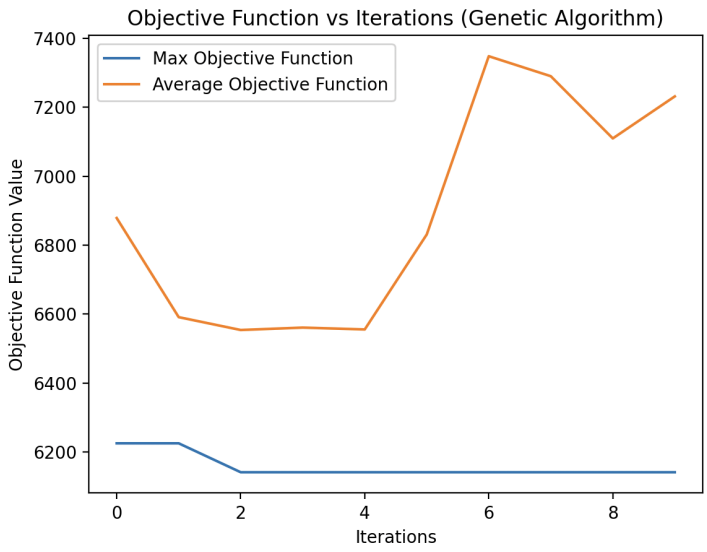
State Awal	

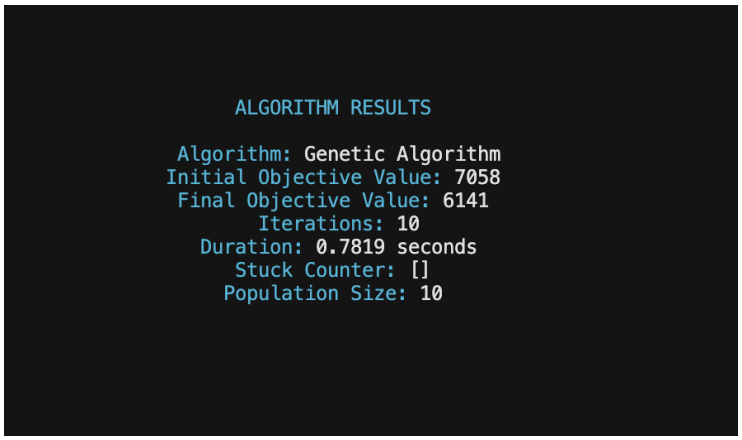
Objective Awal	7349
State Akhir	Final Cube State: Layer 1: 86 68 11 44 54 9 119 102 98 81 60 111 6 2 66 125 56 76 118 24 115 101 19 35 112 Layer 2: 89 4 121 1 84 22 64 85 79 82 20 36 15 94 107 46 55 29 34 7 52 59 99 43 45 Layer 3: 50 38 47 70 61 87 114 51 103 62 108 3 90 104 110 71 83 73 63 123 113 25 92 12 39 Layer 4: 97 67 117 33 58 95 16 31 109 78 14 32 75 18 57 93 77 53 80 26 30 96 65 40 17 Layer 5: 10 88 120 13 28 105 21 5 8 37 48 116 100 106 74 41 124 42 72 69 122 23 27 91 49
Objective Akhir	6297
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	

Hasil Eksperimen	
Jumlah Populasi	10
Banyak Iterasi	10
Durasi Pencarian	0.8160 detik

Tabel 4.3.12. Pengujian 4 Populasi sebagai Kontrol

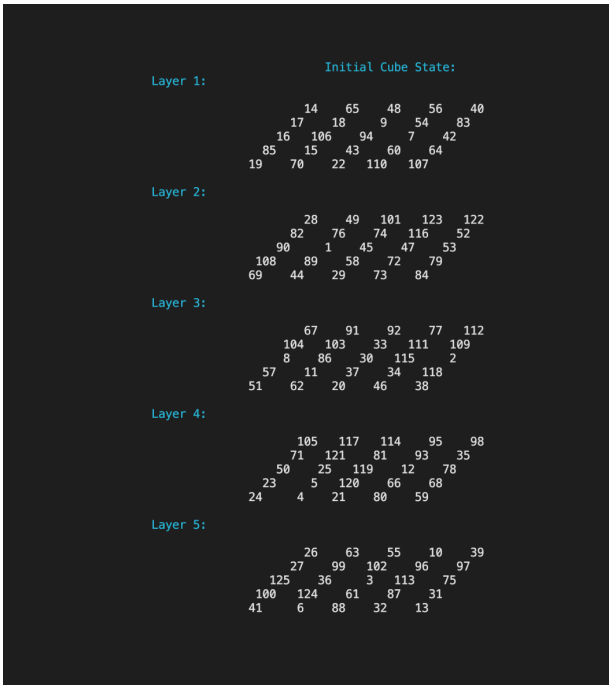
State Awal	
Objective Awal	7058

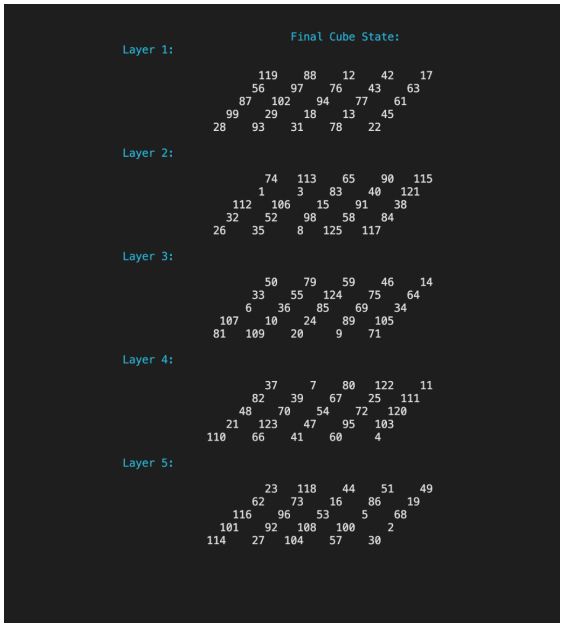
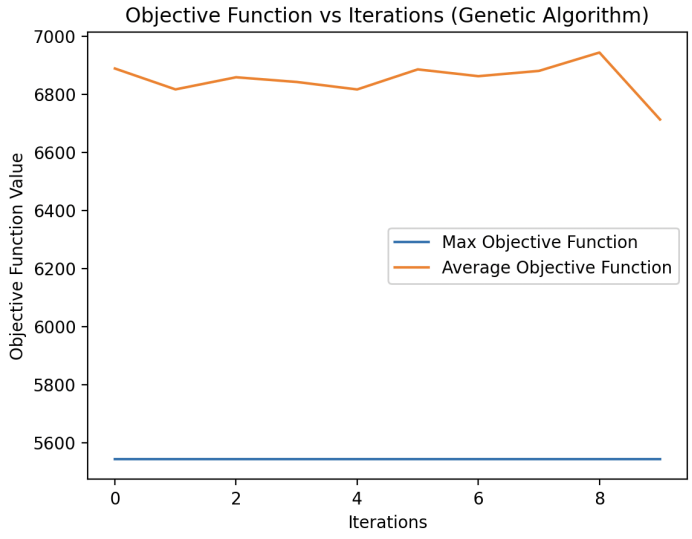
State Akhir	
Objective Akhir	6141
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	

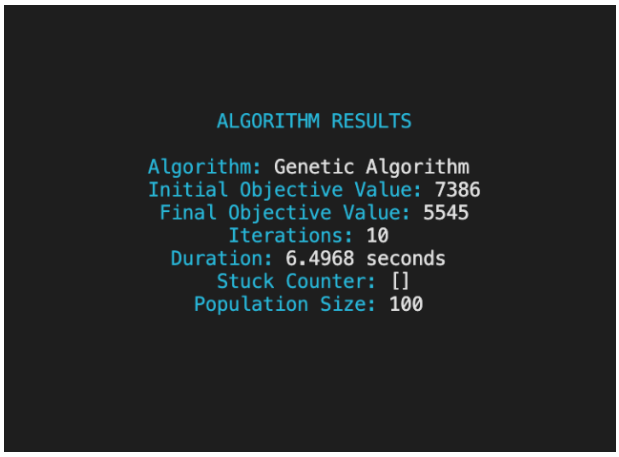
Hasil Eksperimen	
Jumlah Populasi	10
Banyak Iterasi	10
Durasi Pencarian	0.7819 detik

5. Pengujian 5 - Iterasi sebagai Kontrol dengan 100 Populasi

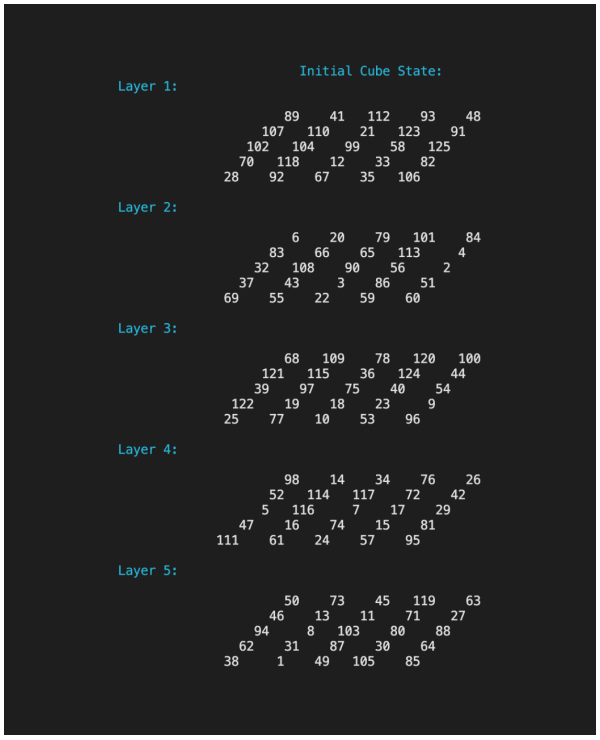
Tabel 4.3.13. Pengujian 5 Iterasi sebagai Kontrol

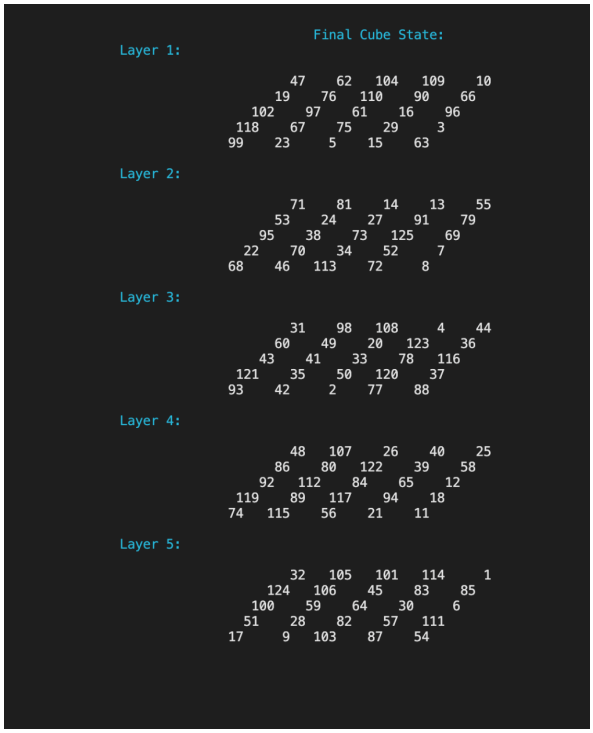
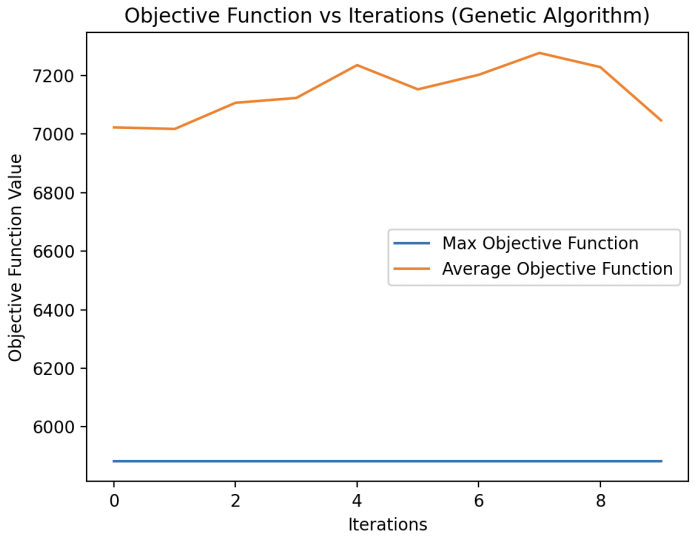
State Awal	

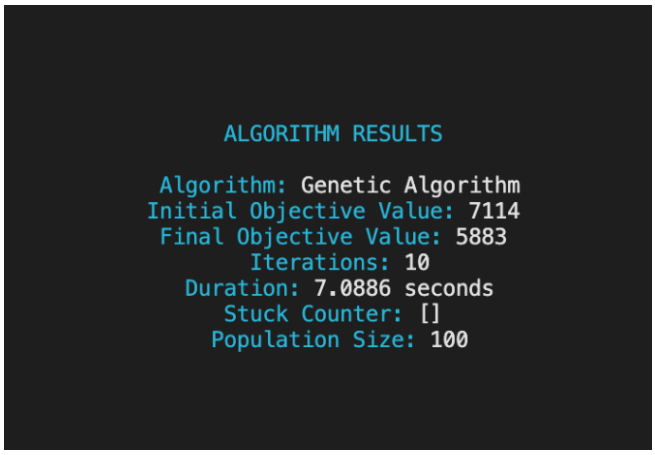
Objective Awal	7386
State Akhir	
Objective Akhir	5545
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	

Hasil Eksperimen	
Jumlah Populasi	100
Banyak Iterasi	10
Durasi Pencarian	6.4968 detik

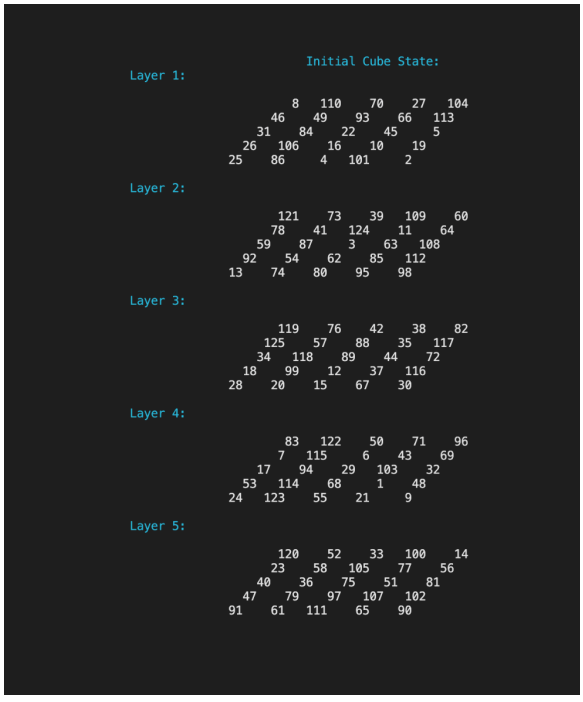
Tabel 4.3.14. Pengujian 5 Iterasi sebagai Kontrol

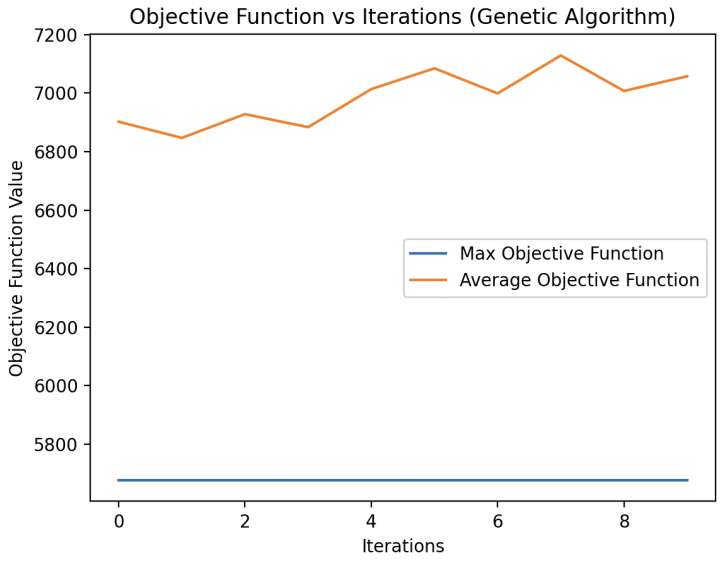
State Awal	

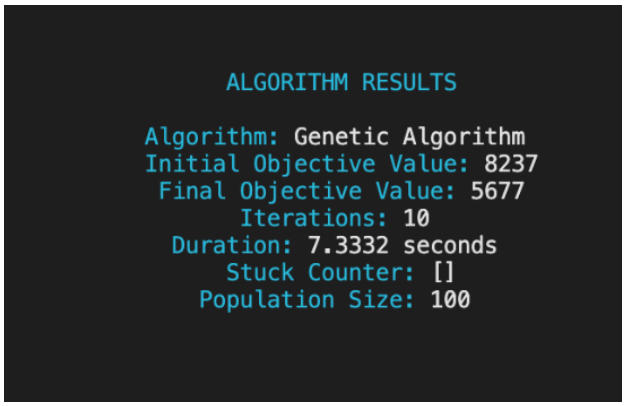
Objective Awal	7114
State Akhir	
Objective Akhir	5883
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	

Hasil Eksperimen	
Jumlah Populasi	100
Banyak Iterasi	10
Durasi Pencarian	7.0886 detik

Tabel 4.3.15. Pengujian 5 Iterasi sebagai Kontrol

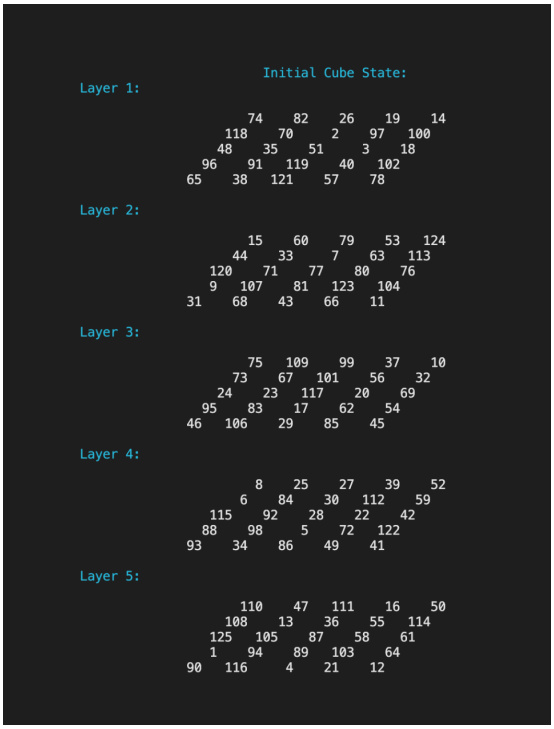
State Awal	
Objective Awal	8237

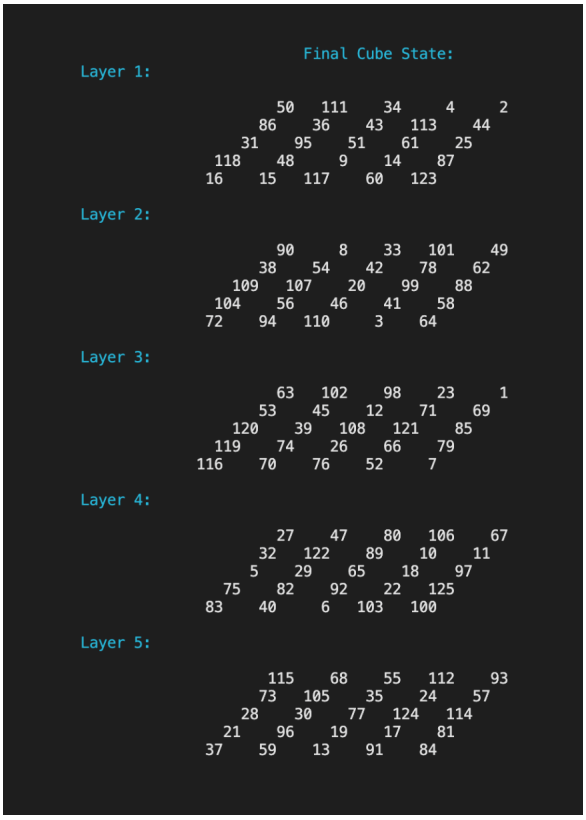
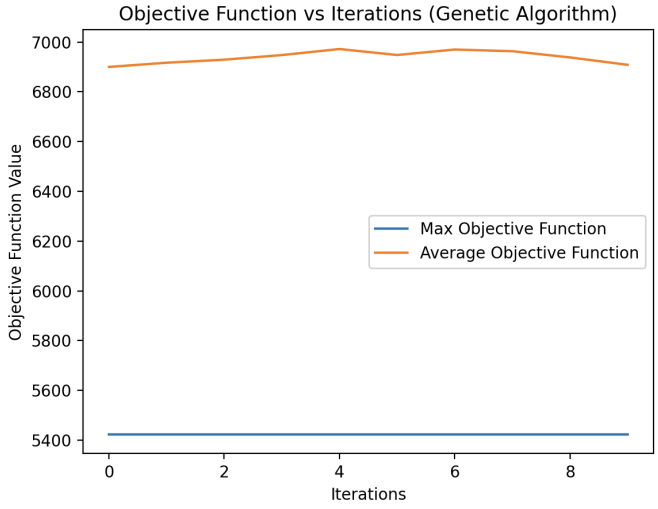
State Akhir	Final Cube State: Layer 1: 37 60 63 86 51 49 48 42 106 14 12 7 98 114 95 75 115 46 3 68 27 119 107 43 32 Layer 2: 71 121 104 90 50 88 74 59 65 116 17 8 55 56 4 96 33 85 77 76 72 87 22 57 30 Layer 3: 11 73 35 61 84 117 29 93 19 105 89 41 36 38 118 125 1 18 83 40 99 78 39 62 101 Layer 4: 67 123 25 20 52 9 81 6 110 44 100 69 26 31 91 10 15 103 108 70 34 113 102 82 122 Layer 5: 94 21 120 23 80 64 45 92 58 124 111 24 13 112 66 16 79 54 28 53 109 97 5 47 2
Objective Akhir	5677
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	

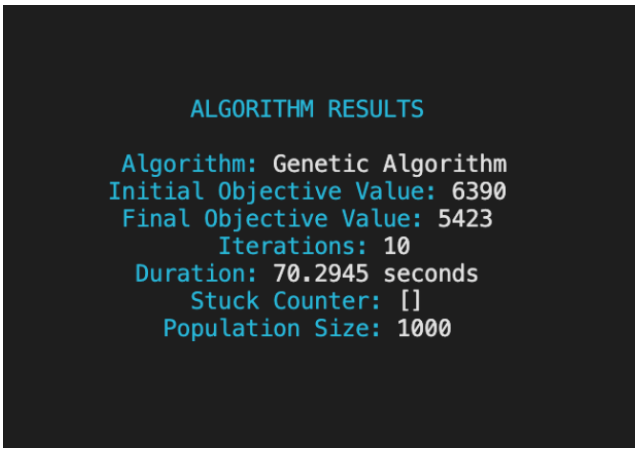
Hasil Eksperimen	
Jumlah Populasi	100
Banyak Iterasi	10
Durasi Pencarian	7.3332 detik

6. Pengujian 6 - Iterasi sebagai Kontrol dengan 1000 Populasi

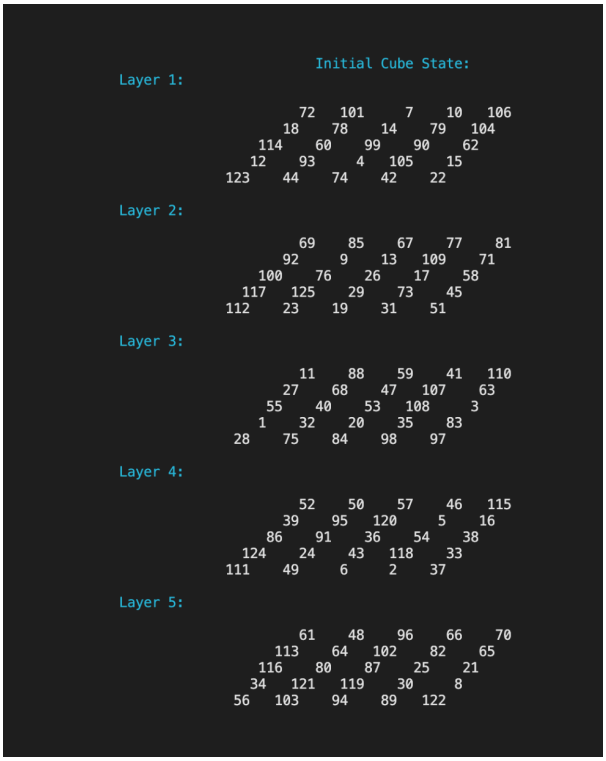
Tabel 4.3.16. Pengujian 6 Iterasi sebagai Kontrol

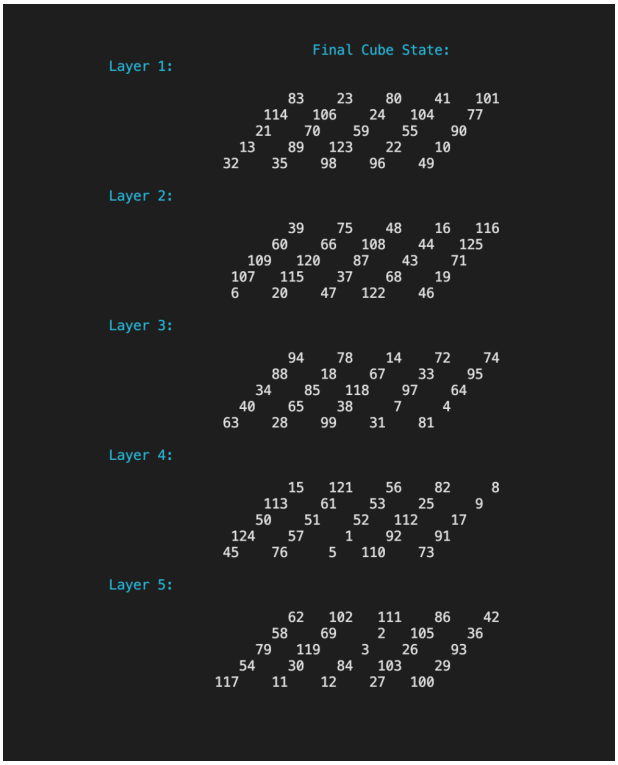
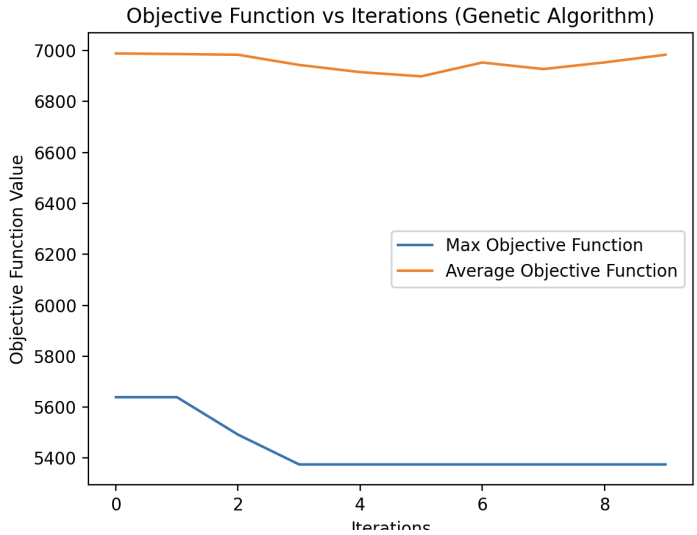
State Awal	

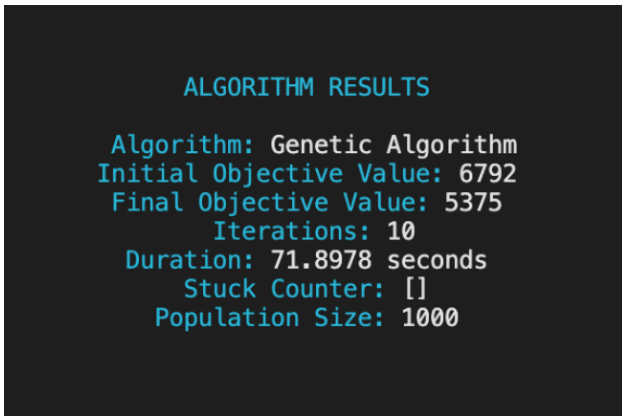
Objective Awal	6390
State Akhir	
Objective Akhir	5423
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	

Hasil Eksperimen	
Jumlah Populasi	1000
Banyak Iterasi	10
Durasi Pencarian	70.2945 detik

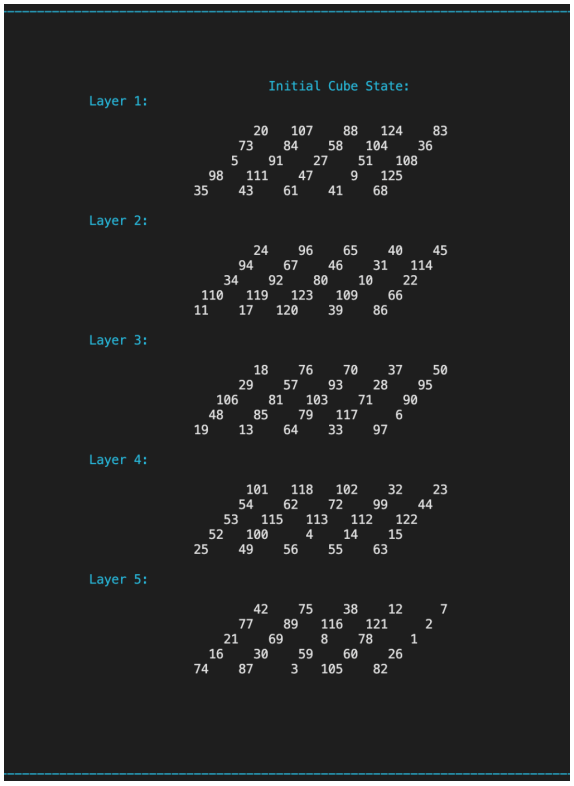
Tabel 4.3.17. Pengujian 6 Iterasi sebagai Kontrol

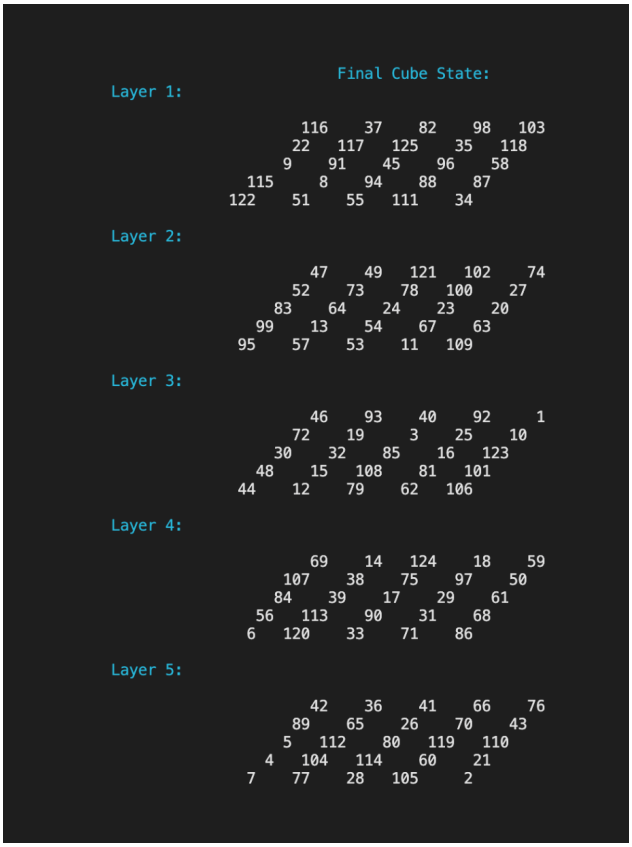
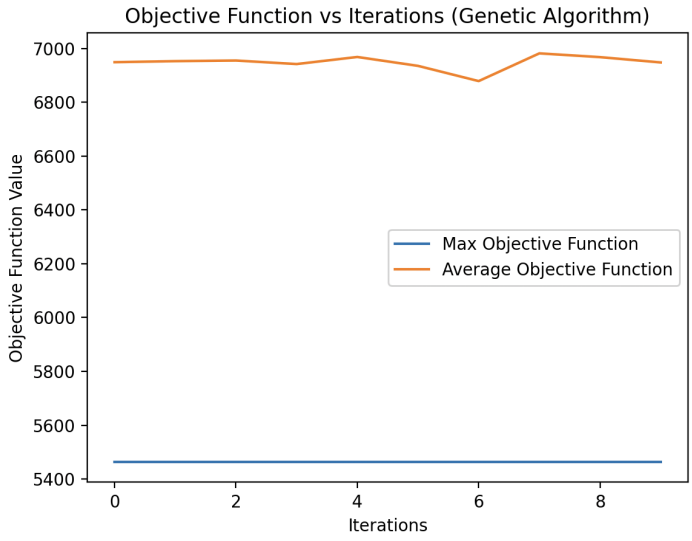
State Awal	

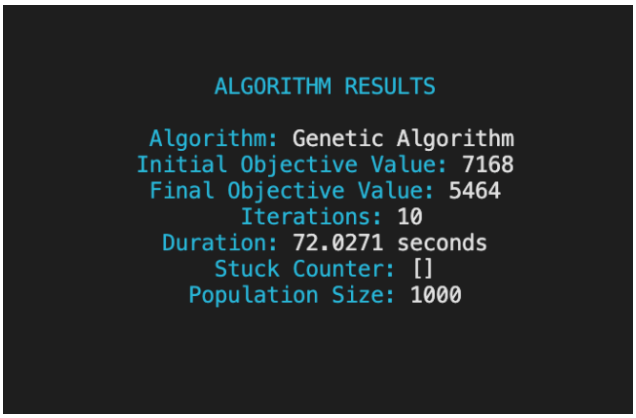
Objective Awal	6792
State Akhir	
Objective Akhir	5375
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	

Hasil Eksperimen	
Jumlah Populasi	1000
Banyak Iterasi	10
Durasi Pencarian	71.8978 detik

Tabel 4.3.18. Pengujian 6 Iterasi sebagai Kontrol

State Awal	

Objective Awal	7168
State Akhir	
Objective Akhir	5464
Plot nilai objective function terhadap banyak iterasi yang telah dilewati	

Hasil Eksperimen	
Jumlah Populasi	1000
Banyak Iterasi	10
Durasi Pencarian	72.0271 detik

5. Analisis

5.1 Tingkat Kedekatan Tiap Algoritma terhadap Global Optima

Simulated Annealing dan **Steepest Ascent Hill Climbing** adalah dua algoritma yang menghasilkan nilai objektif paling dekat dengan global optima. Namun, Simulated Annealing memiliki kelebihan dalam mencari ruang kemungkinan solusi yang lebih luas karena pengaturan suhu tinggi dan *cooling rate* yang lambat, tetapi membutuhkan waktu yang lama. Lalu, Sideways Move menunjukkan hasil yang baik dalam mendekati global optima karena dapat memilih *neighbor* yang memiliki *objective function* bernilai sama ketika terjebak di local optima. Dengan maksimum *sideways* yang cukup besar, Sideways Move mendekati global optima lebih baik dibanding algoritma lainnya, tetapi membutuhkan waktu yang lebih lama. Random Restart, Stochastic, dan Genetic Algorithm cenderung kurang optimal dalam mendekati global optima dalam kondisi yang ditentukan, tetapi Genetic Algorithm memiliki potensi lebih baik jika diterapkan dengan ukuran populasi dan iterasi yang cukup besar.

5.2 Perbandingan Hasil Pencarian Tiap-Tiap Algoritma Dengan Algoritma Local Search yang Lain

a. **Simulated Annealing dan Steepest Ascent Hill Climbing** lebih optimal dalam mencapai nilai objektif terendah, terutama ketika jumlah iterasi banyak. **Simulated Annealing** lebih fleksibel karena dapat memilih *neighbor* dengan nilai *objective* yang lebih buruk untuk mengeksplorasi kemungkinan yang lebih luas, sedangkan **Steepest Ascent** memerlukan lebih banyak waktu dalam proses *searching* tetapi menghasilkan nilai *objective* rendah yang lebih pasti.
b. **Sideways Move** berada di antara Simulated Annealing dan Steepest Ascent Hill Climbing dengan kemampuan memilih *neighbor* yang memiliki nilai *objective* sama dengan *current*, sehingga dapat menghindari *stuck* di *local* minima tetapi cenderung menghabiskan waktu yang lebih lama dan tidak menemukan solusi yang lebih baik dibandingkan Steepest Ascent.
c. **Random Restart** dan **Stochastic Hill Climbing** membutuhkan waktu yang lebih sedikit dalam proses *searching*, tetapi tidak optimal untuk mendapatkan solusi terbaik dibandingkan algoritma Hill Climbing yang lain.

d. **Genetic Algorithm** membutuhkan waktu yang lebih sedikit tetapi menghasilkan nilai *objective* yang lebih tinggi sehingga algoritma ini lebih mengutamakan kecepatan daripada nilai *objective* yang optimal.

5.3 Perbandingan Durasi Proses Pencarian Tiap Algoritma Terhadap Algoritma Lainnya

a. **Steepest Ascent** berkisar antara 76.5 detik sampai 186.9 detik. Dengan peningkatan jumlah iterasi, waktu yang dibutuhkan semakin lama. Iterasi 50 membutuhkan 76.5 detik, sedangkan iterasi 120 membutuhkan waktu hampir tiga kali lipat lebih lama (186.9 detik).
b. **Random Restart** berkisar antara 26.3 detik sampai 86.3 detik. Waktu pencarian meningkat seiring dengan peningkatan jumlah restarts dan iterasi di setiap *restart*. Contohnya, 10 *restart* dengan 5 iterasi membutuhkan 86.3 detik, sedangkan 3 *restart* dengan 5 iterasi hanya membutuhkan 26.3 detik. Random Restart membutuhkan durasi yang relatif panjang dibandingkan dengan Simulated Annealing, namun lebih cepat dibandingkan Steepest Ascent ketika iterasi rendah.
c. **Sideways Move** berkisar antara 181.8 detik sampai 217.1 Sideways move merupakan algoritma dengan durasi paling lama karena adanya kemampuan memilih *neighbor* yang memiliki nilai *objective function* sama sehingga eksplorasi solusi lebih luas.
d. **Stochastic** berkisar antara 21.9 detik sampai 70.6 detik. Meningkatkan jumlah iterasi akan menambah durasi. Contohnya, iterasi 100 membutuhkan 21.9 detik dan iterasi 300 membutuhkan waktu 70.6 detik. Stochastic termasuk cepat dibandingkan dengan Steepest Ascent, SIdeways Move, dan Simulated Annealing, terutama ketika jumlah iterasi sedikit.
e. **Simulated Annealing** berkisar antara 2 detik hingga 79.8 detik, bergantung pada parameter suhu, *cooling rate*, dan iterasi maksimum. Algoritma dengan suhu awal yang tinggi (20000) dan *cooling rate* yang lambat (0.999) memiliki durasi terlama (79.8 detik) karena eksplorasi solusi yang lebih luas dan proses pendinginan yang lambat. Sedangkan algoritma dengan suhu awal lebih rendah (1000) dan *cooling rate* lebih cepat (0.98) memiliki durasi paling singkat (2 detik).
f. **Genetic Algorithm** berkisar dari 0.6631 detik sampai 72.027 detik, tergantung pada jumlah iterasi dan ukuran populasi. Durasi paling singkat (0.6631 detik) yaitu pada iterasi 10 dan populasi 10. Sedangkan durasi

paling lama (72.027 detik) terjadi pada iterasi 10 dengan populasi 1000. Ketika jumlah iterasi sedikit dan populasi kecil, Genetic Algorithm membutuhkan waktu yang jauh lebih singkat daripada algoritma yang lain.

Dapat disimpulkan bahwa:

- Algoritma tercepat: **Genetic Algorithm** dengan iterasi rendah dan populasi kecil.
- Algoritma yang cukup cepat: **Stochastic** pada iterasi rendah dan **Random Restart** dengan sedikit *restart* dan iterasi rendah.
- Algoritma dengan durasi sedang: **Simulated Annealing** tergantung pada pengaturan suhu dan *cooling rate*.
- Algoritma lambat: **Steepest Ascent** dan **Sideways Move**, terutama pada iterasi tinggi atau jumlah pergerakan *sideways* yang besar.

5.4 Konsistensi Hasil Akhir yang Didapatkan dari Tiap-Tiap Eksperimen yang Dilakukan

a. **Steepest Ascent Hill Climbing** menunjukkan konsistensi dalam menghasilkan nilai *objective function* yang rendah, terutama ketika jumlah iterasi tinggi. Hasil eksperimen menunjukkan bahwa algoritma ini terus menurunkan nilai *objective function* secara signifikan dengan peningkatan iterasi, misalnya pada iterasi ke-120, nilai *objective function* dapat turun hingga 272. Namun, algoritma ini membutuhkan waktu yang lebih lama untuk mencapai hasil tersebut, karena mengevaluasi semua *neighbor* di setiap iterasi. Steepest Ascent cenderung konsisten dalam mencapai nilai yang rendah, tetapi karena kurangnya fleksibilitas dalam menerima solusi sementara yang lebih buruk, algoritma ini rentan terjebak di local minima.
b. **Sideways Move Hill Climbing** menunjukkan tingkat konsistensi yang cukup baik dalam menurunkan nilai *objective function*, khususnya saat batas maksimum *sideways* yang diperbolehkan lebih tinggi. Pada eksperimen dengan batas *sideways* 10, algoritma berhasil mencapai nilai *objective* 267, menunjukkan kemampuan untuk menghindari *local* minima yang lebih dalam. Konsistensi hasil akhir juga dipengaruhi oleh jumlah *sideways* yang diperbolehkan untuk meningkatkan kemungkinan mencapai nilai *objective function* yang lebih rendah. Secara keseluruhan, Sideways Move menunjukkan konsistensi dalam mencapai solusi yang baik meskipun dengan waktu *searching* yang lebih lama.

c. **Random Restart Hill Climbing** menunjukkan hasil akhir yang lebih bervariasi dan kurang konsisten dalam mencapai nilai *objective function* yang rendah. Konsistensi algoritma ini sangat bergantung pada jumlah *restart* dan iterasi per *restart* yang diterapkan. Misalnya, pada eksperimen dengan 10 *restart* dan 5 iterasi per *restart*, nilai *objective function* tercatat di angka 4345. Walaupun *restart* membantu algoritma keluar dari *local* minima, hasil akhirnya masih kurang optimal dibandingkan algoritma lain seperti Simulated Annealing atau Steepest Ascent. Dengan demikian, Random Restart kurang konsisten dalam menghasilkan solusi terbaik, terutama jika *restart* atau iterasi per restart terlalu sedikit.
d. **Stochastic Hill Climbing** memperlihatkan tingkat konsistensi yang lebih rendah dibandingkan algoritma lainnya, dengan nilai *objective function* akhir yang cenderung tetap tinggi meskipun iterasi ditingkatkan. Karena *neighbor* dipilih secara acak tanpa memperhatikan seluruh ruang pencarian, algoritma ini menghasilkan hasil akhir yang bervariasi dan kurang optimal. Misalnya, meskipun dijalankan hingga 300 iterasi, nilai *objective function* hanya turun hingga 3128. Hal ini menunjukkan bahwa pencarian secara acak pada Stochastic Hill Climbing mengakibatkan kurangnya konsistensi dalam mencapai solusi terbaik dan lebih rentan terhadap ketidakkonsistenan antar iterasi.
e. **Simulated Annealing** menunjukkan hasil yang relatif konsisten dalam mencapai nilai *objective function* yang rendah. Algoritma ini memanfaatkan parameter temperatur dan *cooling rate* sehingga lebih fleksibel dalam menerima solusi sementara yang lebih buruk di awal pencarian. Konsistensi ini terlihat dari eksperimen dengan variasi temperatur tinggi dan *cooling rate* lambat sehingga menghasilkan penurunan nilai *objective function* yang signifikan meskipun dengan waktu *searching* yang lebih lama. Misalnya, pada eksperimen dengan temperatur awal 20000 dan cooling rate 0.999, nilai *objective* dapat diturunkan menjadi 375. Penyesuaian ini menunjukkan bahwa Simulated Annealing efektif dalam mencapai hasil yang lebih konsisten dan mendekati global optima dengan waktu yang lebih cepat dari Steepest Ascent.
f. **Genetic Algorithm** menunjukkan konsistensi yang cukup baik dalam menghasilkan nilai akhir. Algoritma ini memberikan hasil yang konsisten dalam jangka pendek, tetapi kualitas solusinya terbatas oleh ukuran populasi dan jumlah iterasi. Genetic Algorithm lebih berfokus pada kecepatan dibandingkan eksplorasi ruang pencarian yang optimal,

sehingga hasil akhirnya konsisten pada nilai akhir yang buruk dibandingkan algoritma lainnya.

5.5 Pengaruh Jumlah Iterasi dan Jumlah Populasi Terhadap Hasil Akhir Pencarian pada Genetic Algorithm

1. **Pengaruh Iterasi**

   Berdasarkan hasil yang diperoleh, terlihat bahwa peningkatan jumlah iterasi pada Genetic Algorithm memberikan perubahan positif terhadap penurunan nilai *objective function*, namun juga berbanding lurus dengan durasi pencarian. Pada variasi iterasi pertama, yaitu 10 kali iterasi, nilai *objective function* turun dari 7133 menjadi 6297 dengan durasi pencarian sebesar 0,6631 detik. Ini menunjukkan bahwa Genetic Algorithm mampu melakukan eksplorasi awal untuk mencari solusi yang lebih baik dalam waktu singkat, tetapi penurunan nilai *objective function* yang dihasilkan tidak signifikan. Saat variasi iterasi dinaikkan menjadi 100, nilai *objective function* berkurang lebih jauh menjadi 5458, dengan durasi pencarian yang juga meningkat menjadi 7,5638 detik. Hal ini menunjukkan bahwa jumlah iterasi yang lebih banyak memberi kesempatan algoritma untuk melakukan seleksi, *crossover*, dan mutasi lebih intensif, sehingga berpotensi menghasilkan solusi yang lebih optimal dengan *trade-off* berupa waktu yang lebih lama.

   Pada iterasi ke-1000, nilai *objective function* berhasil diturunkan lebih signifikan lagi menjadi 5324 dengan durasi yang meningkat menjadi 73,9 detik. Hasil ini memperlihatkan bahwa semakin banyak iterasi yang dijalankan, algoritma memiliki peluang lebih besar untuk mengoptimalkan kualitas solusi tetapi tetap tidak signifikan. Peningkatan jumlah iterasi juga memperpanjang waktu yang dibutuhkan sehingga terjadi *trade-off* antara kualitas solusi dan efisiensi waktu.

2. **Pengaruh Populasi**

   Berdasarkan hasil yang diperoleh, terlihat bahwa peningkatan jumlah populasi pada Genetic Algorithm memberikan perubahan positif terhadap penurunan nilai *objective function*, namun juga diiringi dengan peningkatan durasi pencarian. Pada variasi pertama, yaitu dengan populasi sebesar 10 individu dan 10 kali iterasi, nilai *objective function* turun dari 6615 menjadi 5932, dengan durasi pencarian sebesar 0,7908 detik. Hasil ini menunjukkan bahwa populasi yang lebih kecil

memungkinkan algoritma untuk mencapai solusi yang lebih baik dalam waktu singkat, meskipun ruang pencarian terbatas sehingga kualitas solusi mungkin belum optimal. Ketika ukuran populasi dinaikkan menjadi 100 individu dengan jumlah iterasi yang sama, nilai *objective function* berhasil turun lebih jauh menjadi 5545 dengan durasi pencarian yang meningkat menjadi 6,4968 detik. Hal ini mengindikasikan bahwa populasi yang lebih besar menyediakan lebih banyak variasi solusi, yang meningkatkan kemungkinan algoritma menemukan solusi yang lebih optimal, meskipun membutuhkan waktu yang lebih lama.

Pada variasi populasi terbesar, yaitu 1000 individu dengan iterasi yang sama, nilai *objective function* berhasil diturunkan lebih signifikan menjadi 5375, namun durasi pencarian meningkat drastis menjadi 71,8978 detik. Hasil ini memperlihatkan bahwa semakin besar ukuran populasi, algoritma memiliki peluang lebih tinggi untuk menghasilkan solusi yang lebih baik karena lebih banyak individu yang dapat diolah dalam proses seleksi, *crossover*, dan mutasi. Akan tetapi, semakin besar ukuran populasi juga berbanding lurus dengan peningkatan durasi pencarian, sehingga terjadi *trade-off* antara kualitas hasil dan efisiensi waktu.

6. Kesimpulan dan Saran

6.1 Kesimpulan

Berdasarkan analisis yang dilakukan pada berbagai algoritma optimasi, disimpulkan bahwa setiap algoritma memiliki keunggulan dan kelemahan dalam mencapai global optima, durasi eksekusi, dan konsistensi hasil. Algoritma seperti Simulated Annealing dan Steepest Ascent Hill Climbing menunjukkan kinerja yang baik dalam mendekati global optima, terutama dalam kondisi dengan banyak iterasi. Simulated Annealing fleksibel dalam menerima solusi sementara yang kurang baik di awal pencarian, sehingga lebih efektif dalam eksplorasi solusi yang luas meskipun memerlukan waktu lebih lama. Steepest Ascent, meskipun cenderung konsisten tetapi rentan terhadap jebakan local minima, sehingga diperlukan iterasi yang banyak untuk mencapai global optima.

Sementara itu, Sideways Move mampu mencapai solusi yang cukup baik dengan kemampuan menghindari jebakan *local* minima melalui pemilihan *neighbor* yang sama, tetapi membutuhkan waktu yang lebih lama. Random Restart dan Stochastic Hill Climbing, di sisi lain, cenderung lebih cepat, tetapi kurang konsisten dalam mencapai solusi yang optimal.

Genetic Algorithm memberikan hasil yang kurang baik dalam durasi yang lebih singkat, terutama pada populasi dan iterasi rendah. Kemudian, kualitas hasil akhir bergantung pada ukuran populasi dan iterasi, di mana jumlah yang lebih besar meningkatkan peluang mendapatkan solusi yang lebih baik tetapi memperpanjang waktu *searching*.

Secara keseluruhan, performa algoritma ini dipengaruhi oleh faktor-faktor seperti jumlah iterasi, ukuran populasi, dan parameter spesifik pada masing-masing algoritma. *Trade-off* antara kecepatan dan kualitas hasil perlu dipertimbangkan dalam pemilihan algoritma yang sesuai dengan kebutuhan masalah.

6.2 Saran

1. **Pilih algoritma berdasarkan kebutuhan optimalisasi.** Jika kualitas hasil lebih diutamakan daripada waktu, Simulated Annealing atau Steepest Ascent Hill Climbing dengan jumlah iterasi tinggi dapat dipertimbangkan untuk dipilih. Dari sisi kecepatan, Genetic Algorithm dengan populasi dan iterasi rendah dapat dipilih.

2. **Atur parameter algoritma secara tepat.** Untuk mendapatkan hasil terbaik, lakukan tuning parameter seperti suhu dan *cooling rate* pada Simulated Annealing atau jumlah populasi dan iterasi pada Genetic Algorithm. Eksperimen dengan berbagai parameter dapat membantu menemukan solusi yang optimal.

3. **Sesuaikan jumlah iterasi dengan batasan waktu yang ada.** Jika waktu *searching* menjadi kendala, algoritma yang lebih cepat seperti Stochastic Hill Climbing atau Random Restart dapat dipertimbangkan dengan jumlah iterasi yang cukup rendah.

7. Pembagian Tugas Tiap Anggota Kelompok

Berikut adalah tabel rincian mengenai pendistribusian kerja oleh anggota kelompok:

Tabel 7.1. Pembagian Tugas

No	NIM Anggota	Nama Anggota	Deskripsi
1	18222106	Audra Zelvania Putri Harjanto	Algoritma Hill Climbing, Laporan
2	18222118	Rizqi Andhika Pratama	Algoritma Genetic Algorithm, Laporan
3	18222125	Sekar Anindita Nurjadini	Algoritma Simulated Annealing, Laporan
4	18222138	Khayla Belva Annandira	Algoritma Hill Climbing, Laporan

8. Lampiran

Repository tempat penyimpanan program dapat diakses melalui **link** berikut.

Referensi

- Features of the magic cube - Magisch vierkant
- Perfect Magic Cubes (trump.de)
- Magic cube - Wikipedia
- Rubiks Professors Cube 5 x 5 x 5 (grubiks.com)
- Beyond Classical Search - Local Search (Edunex)
- Beyond Classical Search - Hill Climbing (Edunex)
- Beyong Classical Search - Simulated Annealing (Edunex)
- Beyond Classical Search - Genetic Algorithm (Edunex)