

LAPORAN TUGAS BESAR I

IF3170 Intelegensi Artifisial

Pencarian Solusi Diagonal Magic Cube dengan Local Search



Disusun untuk memenuhi tugas mata kuliah Intelegensi Artifisial
pada Semester I Tahun Akademik 2024/2025.

Oleh Kelompok 23:

Thea Josephine Halim	13522012
Debrina Veisha Rashika W	13522025
Melati Anggraini	13522035
Raffael Boymian Siahaan	13522046

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

2024

DAFTAR ISI

DAFTAR ISI	1
BAB I	
DESKRIPSI MASALAH	3
BAB II	
DASAR TEORI	5
2.1 Local Search	5
2.1.1 Algoritma Steepest Ascent Hill-climbing	6
2.1.2 Algoritma Hill-climbing with Sideways Move	6
2.1.3 Algoritma Random Restart Hill-climbing	7
2.1.4 Algoritma Stochastic Hill-climbing	7
2.1.5 Algoritma Simulated Annealing	8
2.1.6 Genetic Algorithm	9
BAB III	
IMPLEMENTASI DAN PENGUJIAN	10
3.1. Pemilihan Objective Function	10
3.2. Heuristic Function	12
3.3. Implementasi Program	14
3.3.1. Steepest Ascent Hill-Climbing	14
3.3.2. SideWaysMove Hill-Climbing	16
3.3.3. Stochastic Hill-Climbing	19
3.3.4. Random Restart Hill-Climbing	22
3.3.5. Simulated Annealing	25
3.3.6. Genetic Algorithm	30
3.3.7. Lain-lain	36
3.3.7.1. Utility	36
3.4. Tampilan Antarmuka	42
3.5. Hasil Pengujian	45
3.5.1. Pengujian Steepest Ascent Hill-Climbing	45
3.5.2. Pengujian Sideways Move Hill-Climbing	48
3.5.3. Pengujian Stochastic Hill-Climbing	53
3.5.4. Pengujian Restart Hill-Climbing	55
BAB IV	77
Stochastic Hill-Climbing	77
Restart Hill-Climbing	77
KESIMPULAN DAN SARAN	82
5.1. Kesimpulan	82

5.2	Saran	82
LAMPIRAN		83
Repository		83
DAFTAR PUSTAKA		83

BAB I

DESKRIPSI MASALAH

Diagonal magic cube merupakan kubus yang tersusun dari angka 1 hingga n^3 tanpa pengulangan dengan n adalah panjang sisi pada kubus tersebut. Angka-angka pada tersusun sedemikian rupa sehingga properti-properti berikut terpenuhi:

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

67	18	119	106	5
116	17	14	73	95
40	50	81	65	79
56	120	55	49	35
36	110	46	22	101

Berdasarkan tingkat simetri dari sebuah *magic cube*, kita bisa membaginya dalam beberapa jenis:

1. *Perfect Magic Cube*

Magic cube yang sempurna, yaitu semua baris, kolom, tiang, dan diagonal baik di sepanjang ruang maupun di dalam setiap bidang menghasilkan jumlah yang sama dengan *magic number*.

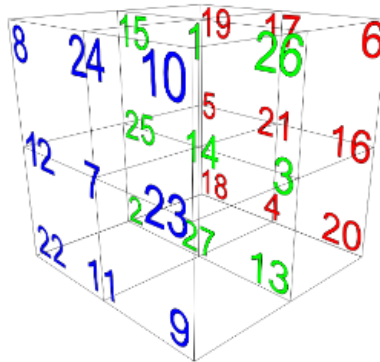
2. *Semiperfect Magic Cube*

Magic cube yang memiliki jumlah angka sama pada semua baris, kolom, dan tiang, tetapi tidak pada semua diagonal di ruang atau bidang.

3. *Simple Magic Cube*

Jenis *magic cube* yang hanya memenuhi syarat untuk baris, kolom, dan tiang, tetapi tidak mempertimbangkan diagonal baik dalam ruang atau bidang. Merupakan bentuk yang paling sederhana dari *magic cube*.

Berikut ilustrasi dari potongan bidang yang ada pada suatu kubus berukuran 3:



Terdapat 9 potongan bidang, yaitu:

8	24	10		15	1	26		19	17	6
12	7	23		25	14	3		5	21	16
22	11	9		2	27	13		18	4	20
19	17	6		5	21	16		18	4	20
15	1	26		25	14	3		2	27	13
8	24	10		12	7	23		22	11	9
8	15	19		12	25	5		22	2	18
24	1	17		7	14	21		11	27	4
10	26	6		23	3	16		9	13	20

Pada kasus *magic cube*, terdapat C_2^n kemungkinan pertukaran dua elemen dengan n ukuran *magic cube*, maka pada ukuran 3 blok *magic cube*, kita akan mendapatkan 351 kemungkinan pertukaran.

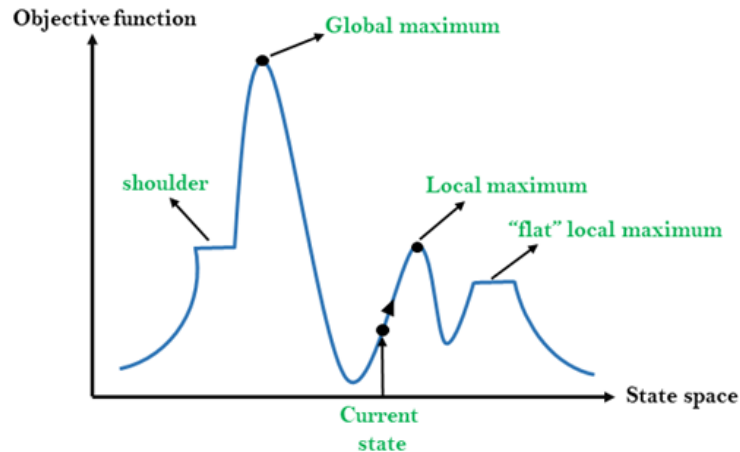
BAB II DASAR TEORI

2.1 *Local Search*

Algoritma *local search* adalah algoritma yang mencari solusi paling optimal dengan mengubah *current state*. *Local search* tidak mempedulikan urutan langkah pembuatan solusi, melainkan hasil akhir solusilah yang menjadi fokus utamanya. Berbeda dengan *classical search* yang memulai dari konfigurasi kosong, *local search* diawali dengan penetapan *initial state* yang *random*.

Pada persoalan *magic cube*, *cube* direpresentasikan sebagai sebuah *state* dengan konfigurasi letak angka yang berbeda-beda di setiap *state*. Setiap iterasinya akan dilakukan perubahan yang diharapkan akan meningkatkan nilai solusi. State yang diubah ini disebut dengan *neighbor state*, state tetangga hasil perubahan lokal dari *current state*. Sebuah fungsi heuristik akan digunakan untuk mengukur keoptimalan solusi. Mengikuti pola *greedy*, apabila nilai heuristik state tetangga lebih baik daripada *current state*, *current state* akan digantikan oleh state tetangga tersebut dan pencarian berlanjut (minimisasi *cost*, maksimisasi *profit*). Algoritma *local search* akan terus-menerus mengeksplor state tetangga lainnya hingga solusi teroptimal ditemukan atau suatu kondisi terpenuhi (*time limit*, banyak iterasi). Walaupun *local search* bertujuan untuk mencari *global optimum*, masih ada kemungkinan terjebak dalam *local optimum*.

Algoritma *Hill Climbing*, *simulated annealing*, dan *genetic algorithm* adalah contoh dari algoritma *local search*. Algoritma *Hill Climbing* dimulai dari suatu state random, dan akan iterasi terus-menerus menuju solusi dengan nilai heuristik yang bertambah. Algoritma akan berhenti (*terminate*) ketika sudah dicapai “peak”, yaitu puncak global maksimum di mana tidak ada lagi state tetangga yang lebih baik. Nantinya algoritma *Hill Climbing* akan dibagi menjadi 4 algoritma terpisah: *Steepest Ascent*, *Sideways Move*, *Random Restart*, dan *Stochastic*. Berbeda dengan algoritma *Hill Climbing* yang tidak memperbolehkan pengambilan solusi *decreasing*, algoritma *Simulated Annealing* memperbolehkan beberapa solusi yang *decreasing* dengan syarat tertentu. Hal ini tentunya membantu untuk melepaskan diri dari kemungkinan terjebak dalam *local optimum*, walaupun dengan *trade off* berkurangnya efisiensi.



Gambar 2.1.1 Grafik Hill Climbing

Source: JavaTpoint

2.1.1 Algoritma *Steepest Ascent Hill-climbing*

Pada algoritma *steepest ascent hill-climbing*, proses pencarian dimulai dari *generate initial state* random yang akan dijadikan *current state*. Berdasarkan *current state* tersebut, dibentuk semua kemungkinan *neighbor state* dan akan diambil yang nilai heuristiknya paling tinggi. Proses iterasi akan berlangsung terus-menerus hingga tercapai “peak” atau tidak ada lagi *neighbor state* yang nilai heuristiknya lebih tinggi.

Pada dasarnya, algoritma *steepest ascent* adalah yang tercepat dalam mencari solusi karena karakteristiknya yang berhenti ketika bertemu lokal optima, ketika tidak ada lagi nilai state yang lebih baik lagi dari *current state*. Sayangnya, ini juga berarti algoritma ini rentan terjebak dalam lokal optima. Setiap perubahan mungkin tidak meningkatkan solusi, menyebabkan algoritma *steepest ascent* berhenti pada titik tersebut karena tidak ada pergerakan yang terlihat lebih baik (tidak ada gradien positif). Sifat eksplorasinya yang terbatas (hanya dengan pertukaran angka) menyebabkan algoritma ini kurang menjanjikan dalam penyelesaian *magic cube*, tetapi masih mungkin mendekatkan kita pada solusi global.

2.1.2 Algoritma *Hill-climbing with Sideways Move*

Proses yang dilakukan pada algoritma *hill-climbing with sideways move* hampir sama dengan algoritma *steepest ascent hill-climbing*. Namun, perbedaan utama antara

keduanya adalah nilai heuristik yang diambil boleh sama. Hal ini bertujuan agar pencarian solusi tidak mudah terjebak pada *local optimum*. Selain itu, ketika *current state* berada pada *shoulder*, state tersebut masih memiliki peluang untuk mencapai global optimum, sehingga algoritma ini memungkinkan sejumlah percobaan dengan limit tertentu (misal 100 kali), dengan harapan bahwa state saat ini dapat bergerak menuju *global optimum*.

2.1.3 Algoritma *Random Restart Hill-climbing*

Pada algoritma *Random Restart Hill-climbing*, proses pencarian dimulai dari *generate initial state* random yang akan dijadikan *current state*. Berdasarkan *current state* tersebut akan dibentuk semua kemungkinan *neighbor state* dan akan diambil yang nilai heuristiknya paling tinggi. Dalam algoritma ini pencarian akan diulang lagi dari awal dengan *current state* random apabila tidak ada lagi *neighbor state* yang nilai heuristiknya lebih tinggi (lokal optimum), dan tidak langsung *terminate* seperti *Steepest Ascent*. Proses akan terus diulang hingga batas iterasi yang ditetapkan sudah tercapai. Hal ini mengurangi kemungkinan terjebak dalam *local optimum*, tetapi akan membutuhkan waktu yang lebih lama tergantung pada penetapan batas iterasi.

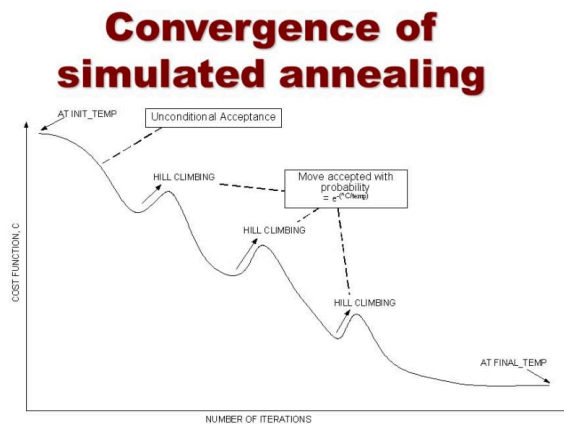
2.1.4 Algoritma *Stochastic Hill-climbing*

Cara kerja algoritma *stochastic hill-climbing* cukup berbeda dari algoritma *hill-climbing* lainnya, seperti *steepest ascent*, *sideways move*, dan *random restart*. Hal dasar yang membedakan *stochastic hill-climbing* dari ketiga variasi tersebut adalah hanya satu *neighbor state* yang akan dibangkitkan secara acak, tidak seperti variasi lainnya yang membangkitkan serta mengevaluasi semua *neighbor state*.

Pengambilan acak ini membantu mencegah terjebak dalam local optimum dan mencoba eksplorasi yang luas. Selanjutnya, *neighbor state* akan dibangkitkan secara acak. Apabila *neighbor value* tersebut lebih baik dari *current value*, *neighbor state* akan menjadi *current state*. Jika tidak, maka akan melanjutkan iterasi hingga n kali (nilai n maksimum ditentukan berdasarkan kebutuhan).

2.1.5 Algoritma *Simulated Annealing*

Berbeda dengan algoritma *hill-climbing* yang tidak memperbolehkan pengambilan solusi *decreasing*, algoritma *Simulated Annealing* memperbolehkan beberapa solusi yang *decreasing* dengan syarat tertentu. Hal ini tentunya membantu untuk melepaskan diri dari kemungkinan terjebak dalam *local optimum*, walaupun dengan *trade off* berkurangnya efisiensi. Beradaptasi dari algoritma *Stochastic Hill-climbing*, algoritma ini akan terus-menerus *looping*, apabila ditemukan *neighbor state* yang lebih baik dari *current state*, *current state* akan digantikan dengan *neighbor state*. Akan tetapi, apabila ternyata nilai *neighbor state* lebih buruk, akan dilakukan perhitungan probabilitas pengambilan.



Gambar 2.1.5.1 Grafik Simulated Annealing

Source: dev.to

Dengan T sebagai nilai temperatur dan e adalah bilangan Euler, kemungkinan diambilnya solusi yang lebih buruk bergantung pada rumus berikut:

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

$$\Delta E = \text{neighbor.value} - \text{current.value}$$

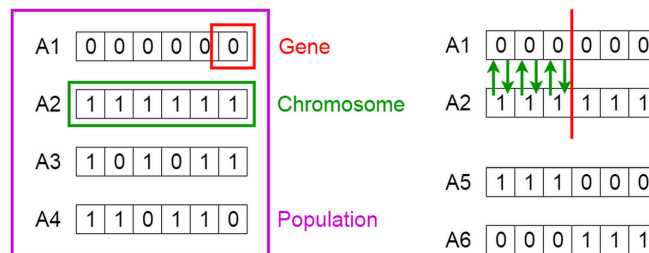
Berdasarkan rumus tersebut, nilai temperatur yang tinggi meningkatkan probabilitas diterimanya state buruk, menambah kemungkinan eksplorasi *states*, sedangkan pada temperatur yang rendah algoritma akan semakin selektif dan memperkecil kemungkinan diambilnya state buruk. Nilai T yang mencapai nol akan menghentikan algoritma *simulated annealing*.

Terdapat 2 pendekatan pengambilan keputusan berdasarkan nilai probabilitas yang diperoleh:

- *Compare to static value*: State buruk akan diambil apabila nilai probabilitas > nilai batasan (*threshold*)
- *Compare to random value*: State buruk akan diambil apabila nilai probabilitas > nilai $\text{random}(0,1)$

2.1.6 Genetic Algorithm

Genetic Algorithms



Gambar 2.1.6.1 Ilustrasi Genetic Algorithm

Source: [towardsdatascience](https://towardsdatascience.com/genetic-algorithms-a-simple-guide-to-evolutionary-computing-1a1e1e1e1e1e)

Genetic algorithm merupakan algoritma yang terinspirasi dari proses seleksi alam dan teori evolusi, yang mengadopsi konsep genetika pada ilmu biologi untuk menemukan solusi terbaik dari permasalahan yang kompleks. Pada *genetic algorithm*, terdapat banyak kombinasi *initial state* yang berjalan secara paralel, dimana satu individu merupakan kumpulan variabel yang sudah di-assign nilai. nilai pada state tersebut dinamakan fitness function, yang semakin tinggi nilainya, semakin baik.

BAB III

IMPLEMENTASI DAN PENGUJIAN

3.1. Pemilihan Objective Function

Langkah-langkah untuk mendapatkan Objective function :

1. Sebelum menentukan Objective function dari magic cube berukuran 5. Terlebih dahulu kami tentukan nilai *number of magic square*-nya dengan menggunakan rumus :

$$m \times m \times m = m^3 (\text{volume kubik}) \dots (1)$$

$$\text{sum all nums} = \frac{((m^3+1) \times m^3)}{2} \dots (2)$$

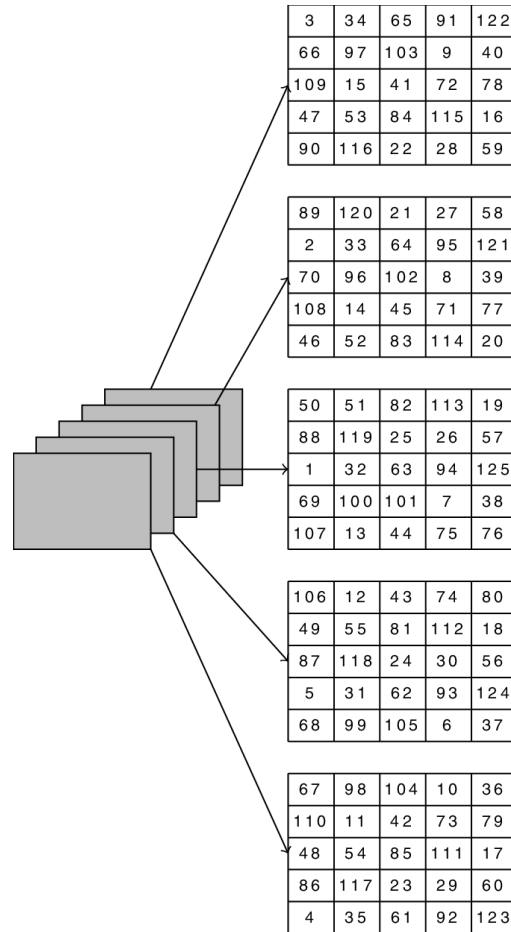
$$\text{magic nums} = \frac{((m^3+1) \times m^3)}{2m^2} \dots (3)$$

$$\text{Final Magic num} = \frac{((m^3+1) \times m)}{2}$$

Dimana n adalah ukuran dari sisi kubus (dalam hal ini, $n = 5$). Substitusi $n = 5$ kedalam rumus :

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

2. Pembagian kubus, kami akan membagi kubik tersebut menjadi 5 lapisan seperti berikut



Gambar 3.1. Pembagian Lapisan(Magic Square) Pada Kubus

Tujuannya adalah agar setiap lapisan tersebut memenuhi aturan magic square, yaitu setiap baris, kolom, dan diagonalnya memiliki jumlah yang sama dengan magic number.

3. Iterasi untuk Setiap Baris, Kolom, dan Diagonal

Secara umum, setiap lapisan akan diiterasikan hingga menemukan solusi yang bisa menghasilkan solusi magic square. Untuk mendapatkan nilai magic square, kami tentukan terlebih dahulu menentukan nilai dari objective functionnya menggunakan cara berikut:

1. Untuk setiap baris, kolom, tiang, dan diagonal, hitung nilai N sebagai *objective value* yang didapat dari selisih absolut dari M adalah *magic square*

number-nya dan *S* nilai konstan dari penjumlahan elemen-elemen dalam setiap baris, kolom, dan diagonal dari *current state*.

$$N = |M - S|$$

Pertimbangan perhitungan fungsi objektif dengan menggunakan perbedaan absolut antara hasil penjumlahan setiap baris, kolom, dan diagonal karena semakin kecil jumlah nilai dari semua perbedaan perbedaan tersebut(mendekati nol), semakin dekat state tersebut menjadi *magic square* yang valid. Sehingga dengan menggunakan *objective* ini bisa mengarahkan ke tujuan yang lebih dekat. Jika ditemukan selisih bernilai 0, maka kami telah menemukan *magic square* yang sempurna.

2. Jumlahkan semua nilai *N* untuk mendapatkan nilai Objective Function

$$F = \sum N$$

Selanjutnya nilai objektif ini akan digunakan untuk melakukan perbandingan dengan *state* lainnya. Goal yang ingin dicapai adalah meminumkan nilai objektif, sehingga semakin kecil nilai objektif sebuah *state* maka semakin baik.

3.2. Heuristic Function

Terdapat fungsi heuristic yang turut diperhitungkan dalam melakukan pencarian menggunakan algoritma local search yang diterapkan, terutama pada pencarian algoritma Hill-Climbing. Hal ini dilakukan agar hasil yang didapatkan bisa mendekati solusi global. Heuristic yang digunakan sebagai berikut:

1. Perhitungan Best Successor dari tetangga yang dipilih saat ini

Saat hendak berpindah *state* ke *processor* terbaik, bisa saja itu justru membuatnya berpindah ke *state* yang lebih buruk karena belum tentu dengan berpindah ke suatu *state* membuat kondisi state selanjutnya lebih baik dari kondisi saat ini. Oleh karena itu, dengan menggunakan metode *two step ahead* yang mana juga dilakukan pertimbangan pada nilai objektif dari *best successor* dari *successor*

yang dipilih terhadap nilai objektif saat ini diharapkan dapat bisa mendekatkan kondisi saat ini dengan kondisi global.

2. Pengecekan Nilai Tengah Kubus

Ketika dilakukan pembangkitan tetangga akan dilakukan pengecekan apakah nilai tengah kubus tersebut sudah merupakan nilai tengah yang tepat karena pada kasus Diagonal Magic Cube nilai tengah kubus selalu bernilai sama yakni, 63. Hal ini bisa dilihat pada riset [berikut](#). Jika nilai tengah kubus tersebut sudah tepat, maka pembangkitan *successor* akan difokuskan pada *state* yang memiliki nilai tengah yang benar saja. Hal ini dilakukan untuk mengurangi waktu analisis setiap state karena jumlah state yang sangat banyak dan mendekatkan solusi ke *global optima*.

3. Pengecekan Angka Pada Posisi Simetris Sentra

Untuk membuat pencarian lebih efektif dibuat sebuah *heuristic function* yang bertujuan mengurangi aspek acak pada saat pembangkitan *random successor* menjadi lebih terarah dan mendekati goal yang dicapai. Salah satu *heuristic function* yang dipertimbangkan adalah menghitung jumlah angka yang posisinya simetris sentral dan berjarak dua blok terhadap titik tengah kubus sebesar 126. Hal ini, didasari pada sebuah [riset](#) yang menyatakan angka pada posisi tersebut selalu bernilai 126. Berikut ilustrasi dari angka yang posisinya simetri sentral, gambar berikut merupakan lapisan persegi yang berada tepat di tengah kubus.

50	51	82	113	19
88	119	25	26	57
1	32	63	94	125
69	100	101	7	38
107	13	44	75	76

$1 + 125 = 126$
 $82 + 44 = 126$
 $50 + 76 = 126$
 $19 + 107 = 126$

Gambar 3.2. Jumlah Angka Pada Posisi Simetris Sentral

Dengan informasi lokasi simetris sentral dan jumlah angka pada posisi tersebut, maka pencarian (jika nilai sudah memenuhi fungsi heuristik) akan dilakukan terhadap sisa posisi sel yang bukan terletak di posisi simetris sentral sehingga mengurangi ruang pencarian. Oleh karena itu, penggunaan fungsi heuristik seperti ini memungkinkan algoritma untuk lebih efisien dalam mencari solusi optimal dengan mengurangi ruang pencarian yang perlu dieksplorasi dan memfokuskan upaya pencarian pada posisi-posisi yang lebih penting dalam struktur kubus.

4. Pertimbangan nilai variansi, mean dan beda komponen dengan nilai *magic number*

Untuk membuat pencarian efektif, kita menambahkan fungsi heuristik untuk mengurangi aspek *randomness* pada saat melakukan pembangkitan *successor* secara acak terutama pada algoritma stochastic hill climbing dan simulated annealing. Kami menghitung nilai variansi, mean dan perbedaan total value tiap komponen dan memastikan nilai-nilai tersebut lebih kecil dari dari value state saat ini. Jika nilai-nilai tersebut lebih kecil maka state akan dipilih. Dengan menggunakan tambahan heuristic ini diharapkan bisa mendekatkan state ke global optima.

3.3. Implementasi Program

Program dibagi menjadi beberapa class berdasarkan nama algoritma, dengan file utama main.py untuk dijalankan.

3.3.1. Steepest Ascent Hill-Climbing

3.3.1.1. Langkah - Langkah

Secara keseluruhan, berikut adalah langkah-langkah pemecahan masalah dengan algoritma steepest ascent:

1. Mulai dengan nilai initial state yang acak(random).
2. Dalam setiap iterasi, algoritma akan memilih tetangga(successor) dengan nilai objektif diantara semua tetangga.

3. Jika nilai tetangga lebih kecil atau sama dengan keadaan saat ini, algoritma akan berhenti. Jika nilainya lebih tinggi current state akan diganti dengan nilai tetangga.
4. Algoritma akan berhenti ketika mencapai lokal optima dimana tidak ada tetangga yang lebih tinggi

3.3.1.2. Kode Program

a. Fungsi init

Deskripsi	Fungsi untuk melakukan inisiasi state awal dan menyimpan <i>history</i> tiap iterasinya.
<pre>def __init__(self): self.Node = Node(cube_size=5) self.history = [] self.initial_state = self.Node self.history.append({ "frame": 1, "cube": copy.deepcopy(self.Node.cube), "objective_value": self.Node.current_value })</pre>	

b. Fungsi solveCube

Deskripsi	Fungsi untuk melakukan pencarian dengan Steepest Ascent Algorithm
<pre>def solveCube(self): print("Starting search process (Steepest)") print(f"Initial Node(Objektif Value) value: {self.Node.current_value}") i = 2 while True: neighbour = self.Node.getHighestSuccessor()</pre>	


```
        if neighbour.current_value == 0:
            self.Node = neighbour
            self.history.append({"frame": i, "cube":
copy.deepcopy(self.Node.cube), "objective_value":
self.Node.current_value})
            break

        if neighbour.current_value >
self.Node.current_value:
            print("Local maximum reached.")
            break

        self.Node = neighbour
        print(f"Updated Node (Objektif Value) to new
value: {self.Node.current_value}")
        self.history.append({"frame": i, "cube":
copy.deepcopy(self.Node.cube), "objective_value":
self.Node.current_value})
        i += 1

    print(f"Initial state of the cube:")
    self.initial_state.showCube
    print(f"Final state of the cube:")
    self.Node.showCube()
    print(f"Final objective function value achieved:
{self.Node.current_value}")
    print(f"Total iterations until search stopped:
{i-1}")

    return self.Node
```

3.3.2. SideWaysMove Hill-Climbing

3.3.2.1. Langkah - Langkah

Secara keseluruhan, berikut adalah langkah-langkah pemecahan masalah dengan algoritma sideways move:

1. Memulai dengan nilai initial state yang acak(random).
2. Dalam setiap iterasi, algoritma akan memilih tetangga(successor) dengan nilai objektif diantara semua tetangga.
3. Jika nilai tetangga lebih kecil dengan keadaan saat ini, algoritma akan berhenti. Jika nilainya lebih tinggi current state akan diganti dengan nilai tetangga. Ini memungkinkan gerakan ke samping(sideways) hingga batas waktu tertentu untuk menangani kondisi flat agar tidak terjebak di lokal optima.
4. Algoritma akan berhenti ketika menemukan tetangga dengan nilai objektif 0 atau saat mencapai maksimal iterasi.

3.3.2.2. Kode Program

a. Fungsi init

Deskripsi	Fungsi untuk melakukan inisiasi state awal dan menyimpan <i>history</i> tiap iterasinya.
<pre>def __init__(self): self.Node = Node(cube_size=5) self.history = [] self.initial_state = self.Node self.history.append({ "frame": 1, "cube": copy.deepcopy(self.Node.cube), "objective_value": self.Node.current_value })</pre>	

b. Fungsi solveCube

Deskripsi	Fungsi untuk melakukan pencarian dengan Sideways Move Algorithm
<pre> def solveCube(self, max_sideways_moves): print("Starting search process (Sideways Move Hill-Climbing)") i = 2 sideways_moves = 0 print(f"Initial Node(Objektif Value) value: {self.Node.current_value}") while True: neighbour = self.Node.getHighestSuccessor() print(f"neighbor value : {neighbour.current_value} current value : {self.Node.current_value} ") if neighbour.current_value == 0: self.Node = neighbour self.history.append({"frame": i, "cube": copy.deepcopy(self.Node.cube), "objective_value": self.Node.current_value}) break if neighbour.current_value > self.Node.current_value: print("Local maximum reached.") break if neighbour.current_value < self.Node.current_value: self.Node = neighbour elif neighbour.current_value == </pre>	

```
self.Node.current_value:
    if sideways_moves < max_sideways_moves:
        sideways_moves += 1
        self.Node = neighbour
    else:
        print("Max sideways moves reached.
Stopping search.")
        break

    print(f"Updated Node (Objektif Function) to
new value: {self.Node.current_value}")
    self.history.append({"frame": i, "cube":
copy.deepcopy(self.Node.cube), "objective_value":
self.Node.current_value})
    i += 1

    print(f"Initial state of the cube:")
    self.initial_state.showCube
    print(f"Final state of the cube:")
    self.Node.showCube()
    print(f"Final objective function value achieved:
{self.Node.current_value}")
    print(f"Total iterations until search stopped:
{i-1}")
    print(f"Total sideways move: {sideways_moves}")

    return self.Node
```

3.3.3. Stochastic Hill-Climbing

3.3.3.1. Langkah - Langkah

Berikut merupakan langkah langkah yang dilakukan dengan menggunakan algoritma stochastic hill climbing pada aplikasi tugas ini:

1. Langkah pertama adalah membentuk initial state, yaitu membuat sebuah kubus yang diisi dengan angka-angka secara acak.
2. Algoritma kemudian memilih sebuah tetangga (neighbor) secara acak dari initial state.
3. Setelah tetangga dipilih, algoritma akan membandingkan nilai objektif (objective value) dari current state dan tetangga baru yang dibangkitkan secara acak dipilih. Nilai objektif ini mengukur kualitas solusi, seperti seberapa baik susunan angka di kubus sesuai dengan tujuan masalah.
4. Jika nilai objektif dari tetangga lebih baik daripada current state, maka algoritma akan memperbarui state dengan berpindah ke tetangga yang baru.
5. Jika nilai objektif tetangga lebih buruk, algoritma tidak berpindah dan tetap berada di current state. Ini memungkinkan algoritma untuk fokus pada solusi yang lebih baik di iterasi berikutnya.
6. Langkah 2 hingga 5 akan terus diulang hingga mencapai jumlah iterasi maksimum yang telah ditentukan sebelumnya. Pada akhir iterasi, algoritma akan menghasilkan solusi terbaik yang ditemukan selama proses berlangsung.

3.3.3.2. Kode Program

a. Fungsi init

Deskripsi	Fungsi untuk melakukan inisiasi state awal dan menyimpan <i>history</i> tiap iterasinya.
<pre>def __init__(self): self.Node = Node(cube_size=5) self.history = [] self.initial_state = self.Node self.history.append({ "frame": 1, "cube": copy.deepcopy(self.Node.cube),</pre>	

```
"objective_value": self.Node.current_value
}))
```

b. Fungsi solveCube

Deskripsi	Fungsi untuk melakukan pencarian dengan Stochastic Algorithm
	<pre>def solveCube(self, maxIteration): print("Starting search process (Stochastic Hill-Climbing)") i = 0 while i < maxIteration: neighbour = self.Node.getRandomSuccessor() if neighbour.current_value == 0: self.Node = neighbour self.history.append({"frame": i, "cube": copy.deepcopy(self.Node.cube), "objective_value": self.Node.current_value}) break if neighbour.current_value <= self.Node.current_value: neighbour2 = neighbour.getHighestSuccessor() if neighbour2.current_value <= self.Node.current_value: self.Node = neighbour2 else: self.Node = neighbour print(f"Updated Node to new value:</pre>

```
{self.Node.current_value}")

        self.history.append({"frame": i, "cube":
copy.deepcopy(self.Node.cube), "objective_value":
self.Node.current_value})
        i += 1

    print(f"Initial state of the cube:")
    self.initial_state.showCube
    print(f"Final state of the cube:")
    self.Node.showCube()
    print(f"Final objective function value achieved:
{self.Node.current_value}")
    print(f"Total iterations until search stopped: {i}")

    return self.Node
```

3.3.4. Random Restart Hill-Climbing

3.3.4.1. Langkah - Langkah

Berikut merupakan langkah langkah yang dilakukan dengan menggunakan algoritma random restart hill climbing pada aplikasi tugas ini:

1. Menentukan initial state secara acak untuk membangun kubus pada setiap *restart*.
2. Algoritma akan mengecek keseluruhan tetangga dan menghitung nilai objektifnya.
3. Algoritma akan memilih tetangga yang terbaik yang nantinya akan dilakukan pemindahan state dari current menuju tetangga jika nilainya lebih baik (dalam kasus ini yang memiliki nilai objektif lebih kecil dari current state).
4. Algoritma terus mengevaluasi dan bergerak menuju tetangga terbaik sampai tidak ada gerakan yang lebih baik lagi. Hal ini bisa terjadi saat

kubus hampir terpecahkan atau ketika algoritma terjebak di maksimum lokal, di mana semua gerakan yang mungkin justru memperburuk keadaan.

5. Jika solusi optimal global belum ditemukan, ulangi proses dengan initial state baru (*restart*) secara acak.
6. Algoritma berhenti pada setiap iterasi ketika mencapai keadaan di mana tidak ada tetangga yang memiliki nilai lebih baik dari keadaan saat ini. Jika state menunjukkan nilai objektifnya nol maka kubus telah terpecahkan,.
7. Algoritma berhenti setelah mencapai kondisi *goal state* yaitu objektif functionnya nol atau telah mencapai batas pengulangan yang telah ditentukan sebelumnya.

3.3.4.2. Kode Program

a. Fungsi init

Deskripsi	Fungsi untuk melakukan inisiasi state awal dan menyimpan <i>history</i> tiap iterasinya.
<pre>def __init__(self): self.history = [] self.initial_state = None</pre>	

b. Fungsi SolveCube

Deskripsi	Fungsi untuk melakukan pencarian dengan Random Restart Hill-climbing
<pre>def solveCube(self, maxRestart): print("Starting search process (Random Restart Hill-Climbing)") print(f"Maximum restarts allowed: {maxRestart}")</pre>	


```
currvalue = float('inf')
currcube = None
self.initial_state = None

for i in range(maxRestart):
    RandomNode = Node(cube_size=5)
    if i == 0:
        self.initial_state = RandomNode

    iterations = 0

    while iterations < 1000:
        neighbour = RandomNode.getHighestSuccessor()
        iterations += 1

        if neighbour.current_value <
RandomNode.current_value:
            neighbour2 =
neighbour.getHighestSuccessor()

            if neighbour2.current_value <
RandomNode.current_value:
                RandomNode = neighbour2
            else:
                RandomNode = neighbour
        else:
            break

    if RandomNode.current_value < currvalue:
        currvalue = RandomNode.current_value
        currcube = RandomNode
        print(f"New best value found: {currvalue}
(Restart {i + 1}, Iterations: {iterations})")
```

```
        self.history.append({"frame": i + 1, "cube":  
currcube.cube, "iterations": iterations,  
"objective_value": currvalue})  
  
        if i + 1 == maxRestart:  
            print("Maximum restarts reached, stopping  
search.")  
            break  
  
        print(f"Initial state of the cube: ")  
        self.initial_state.showCube()  
        print(f"Final state of the cube: ")  
        currcube.showCube()  
        print(f"Final objective function value achieved:  
{currvalue}")  
        print(f"Total restarts: {i + 1}")  
  
        return currcube
```

3.3.5. Simulated Annealing

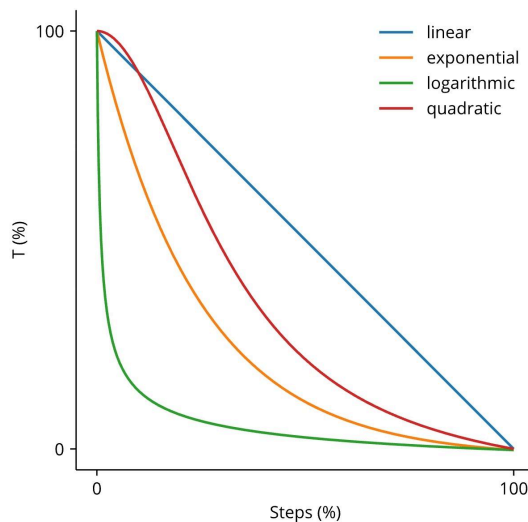
3.3.5.1. Langkah - Langkah

Secara keseluruhan, berikut adalah langkah-langkah pemecahan masalah dengan algoritma simulated annealing:

1. Memulai dengan nilai initial state yang acak (random).
2. Untuk setiap proses iterasi, akan dilakukan validasi apakah temperatur sudah mencapai 0 (kondisi ideal). Jika iya, langsung me-*return current state*. Jika tidak, akan dibangkitkan 1 (satu) *neighbor state* secara acak.
3. Selanjutnya, akan diperiksa nilai delta E. jika $\Delta E > 0$, maka *neighbor state* akan menjadi *current state*, karena *neighbor state* dianggap memiliki *value* yang lebih baik ketimbang *current state*. Jika tidak, akan diperiksa nilai probabilitas dengan menggunakan rumus eksponen $P = e^{\frac{\Delta E}{T}}$.

4. Jika nilai *random* antara 0 - 1 lebih kecil dari probabilitas, *worse neighbor* tersebut akan tetap diterima karena masih memiliki probabilitas kuat untuk menuju ke *goal state*.
5. Sebelum melanjutkan iterasi, temperatur akan berkurang berdasarkan *cooling rate* atau dengan variasi lain (misal pengurangan nilai temperatur sebesar -1 untuk satu kali iterasi).

Dalam implementasi program Simulated Annealing kami memberikan kebebasan kepada user untuk memilih jenis cooling rate: linear, eksponensial, logaritmik, dan kuadratik.



Gambar 3.3.5.1 Grafik Perbandingan Jenis Schedule

Linear cooling schedule dirumuskan dalam:

$$Temp = T - cooling_rate * iteration$$

Eksponensial cooling schedule dirumuskan dalam:

$$Temp = T * (cooling_rate^{iteration})$$

Logaritmik cooling schedule dirumuskan dalam:

$$Temp = T / (1 + cooling_rate * \log(k + iteration))$$

Kuadratik cooling schedule dirumuskan dalam:

$$Temp = T / (1 + cooling_rate * k)$$

Berdasarkan analisis yang dilakukan Nathan Rooy, ada beberapa tips untuk pemilihan schedule annealing pada cooling schedule. Apabila kita sudah yakin

bahwa objective function kita sudah jelas, lebih baik untuk tidak melakukan eksplorasi luas dan fokus pada penurunan suhu yang lebih cepat (cooling rate tinggi). Penurunan suhu yang cepat cocok untuk schedule logaritmik atau eksponensial. Sedangkan untuk masalah yang memiliki banyak lokal maksima, kita bisa menggunakan algoritma dengan penurunan suhu lebih lambat, membuat kita bisa melakukan eksplorasi lebih luas dan menghindari terjebak pada maksimum lokal.

Kasus kompleks magic cube membutuhkan banyak konfigurasi dan banyaknya kemungkinan terjebak pada lokal maksima (atau minima jika menghitung nilai yang lebih rendah yang lebih baik), sehingga scheduling linier atau kuadratik lebih cocok. Pemilihan scheduling dengan penurunan suhu lambat membantu kita bisa mengurangi kemungkinan resiko stuck di local optima.

3.3.5.2. Kode Program

a. Fungsi init

Deskripsi	Fungsi untuk melakukan inisiasi state awal dan menyimpan <i>history</i> tiap iterasinya.
<pre>def __init__(self, initial_temp, cooling_rate, schedule_type): self.Node = Node(cube_size=5) self.history = [] self.initial_temp = initial_temp self.cooling_rate = cooling_rate self.schedule_type = schedule_type self.stuck = 0 self.history.append({ "frame": 1, "cube": copy.deepcopy(self.Node.cube), "objective_value": self.Node.current_value })</pre>	

b. Fungsi linear_cooling

Deskripsi	Fungsi untuk melakukan penghitungan temperatur baru dengan jenis schedule linear
<pre>def linear_cooling(self, temp, iteration): return temp - self.cooling_rate * iteration</pre>	

c. Fungsi exponential_cooling

Deskripsi	Fungsi untuk melakukan penghitungan temperatur baru dengan jenis schedule eksponensial
<pre>def exponential_cooling(self, temp, iteration): return temp * (self.cooling_rate ** iteration)</pre>	

d. Fungsi logarithmic_cooling

Deskripsi	Fungsi untuk melakukan penghitungan temperatur baru dengan jenis logaritmik
<pre>def logarithmic_cooling(self, temp, iteration): return temp / (1 + self.cooling_rate * math.log(1 + iteration))</pre>	

e. Fungsi quadratic_cooling

Deskripsi	Fungsi untuk melakukan penghitungan temperatur baru dengan jenis schedule kuadratik
<pre>def quadratic_cooling(self, temp, iteration): return temp / (1 + self.cooling_rate * iteration * iteration)</pre>	

f. Fungsi simulatedAnnealing

Deskripsi	Fungsi untuk melakukan pencarian dengan Simulated Annealing
	<pre> def simulatedAnnealing(self): current_heuristic = self.Node.calculateHeuristic() initial_heuristic = current_heuristic temp = self.initial_temp i = 2 while True: if temp <= 0: break neighbor = self.Node.getRandomSuccessor() neighbor_heuristic = neighbor.calculateHeuristic() deltaE = neighbor_heuristic - current_heuristic if neighbor.current_value == 0: self.Node = neighbor self.history.append({"frame": i, "cube": copy.deepcopy(self.Node.cube), "objective_value": self.Node.current_value}) break if deltaE < 0: self.Node = neighbor current_heuristic = neighbor_heuristic self.history.append({"frame": i, "cube": copy.deepcopy(self.Node.cube), "objective_value": </pre>

```
self.Node.current_value})

        else:
            self.stuck+=1
            rand = random.random()
            val = math.exp(-deltaE / temp)
            if val > rand:
                self.Node = neighbor
                current_heuristic =
neighbor_heuristic
                self.history.append({"frame": i,
"cube": copy.deepcopy(self.Node.cube),
"objective_value": self.Node.current_value})

            else:
                if self.schedule_type == "linear":
                    temp = self.linear_cooling(temp, i)
                elif self.schedule_type == "exponential":
                    temp = self.exponential_cooling(temp, i)
                elif self.schedule_type == "logarithmic":
                    temp = self.logarithmic_cooling(temp, i)
                elif self.schedule_type == "quadratic":
                    temp = self.quadratic_cooling(temp, i)
                i+=1

        print("Final state of the cube:")
        self.Node.showCube()
        print(f"Initial objective function value:
{initial_heuristic}")
        print(f"Final objective function value:
{current_heuristic}")
        print(f"Total iterations: {i}")
        print(f"Stuck in local optima: {self.stuck}")
        return self.Node
```

3.3.6. Genetic Algorithm

3.3.6.1. Langkah - Langkah

Secara keseluruhan, berikut adalah langkah-langkah pemecahan masalah dengan algoritma Genetic Algorithm:

1. Sebuah populasi akan diinisialisasi secara acak. Satu populasi berisikan n individu yang masing-masing sudah memiliki konfigurasi lengkap *magic cube*, yaitu kubus yang sudah tersusun atas nomor-nomor untuk semua *block*.
2. Selanjutnya, dilakukan *fitness evaluation* dari setiap individu berdasarkan *fitness function*. *Fitness function* disesuaikan dengan kasus tertentu, tetapi untuk kasus *magic cube*, dapat menggunakan *fitness function* berikut :
$$fitness\ function = \max(array\ of\ heuristic\ value) - heuristic\ value$$
Rumus tersebut menyatakan bahwa semakin besar *fitness value* suatu *node* (semakin dekat dengan *magic number*), semakin besar pula kemungkinan dia terpilih sebagai *parent* untuk *crossover*.
3. Setelah menentukan *fitness value*, dilakukan proses seleksi menggunakan *roulette wheel* atau pita dari angka 0 - 100 (*range* angka diperoleh dari *fitness function*, yaitu $fitness\ value\ individu\ ke-i / total\ fitness\ value\ seluruh\ individu$). Tujuan dari proses ini adalah memilih kandidat *parent* untuk diproses di tahap selanjutnya.
4. Setelah dilakukan seleksi, akan dilakukan *crossover* (penyilangan) antar individu terpilih untuk menghasilkan individu baru. Metode *crossover* dapat disesuaikan dengan kebutuhan.
5. Dalam probabilitas yang sangat kecil, dapat dilakukan mutasi, misalkan dengan menukar dua elemen secara acak dalam suatu individu.
6. Setelah melalui *crossover* (bahkan mutasi), *child* yang diperoleh dari ‘perkawinan’ kedua *parent* akan menjadi populasi baru yang akan

mengulangi proses yang dijalani pendahulunya hingga mencapai solusi yang diinginkan.

7. Proses ini akan terus berulang hingga solusi ditemukan atau mencapai jumlah maksimal generasi yang diinginkan.

3.3.6.2. Kode Program

a. Fungsi Init

Deskripsi	Fungsi untuk melakukan inisiasi state awal populasi, ukuran populasi, tingkat mutasi, jumlah iterasi maksimum, dan penyimpanan riwayat untuk analisis.
<pre>def __init__(self, cube_size, population_size, max_iterations): self.population = [Node(cube = None, cube_size = cube_size) for i in range (population_size)] self.population_size = population_size self.mutation_rate = 0.3 self.max_iterations = max_iterations self.history = []</pre>	

b. Fungsi calculatePopulationFitness

Deskripsi	Fungsi untuk Menghitung fitness setiap individu dalam populasi.
<pre>def calculatePopulationFitness(self): result = [] max_nodes = max(self.population).current_value for i in range (len(self.population)): result.append(max_nodes - self.population[i].calculateHeuristic()) total = sum(result) if total == 0:</pre>	

```

        return [1 / self.population_size] *
self.population_size
        for i in range (len(self.population)):
            temp = result[i]
            result[i] = temp / total
        return result

```

c. Fungsi CreateInterval

Deskripsi	Fungsi untuk Membuat interval kumulatif berdasarkan fitness populasi.
<pre> def calculatePopulationFitness(self): result = [] max_nodes = max(self.population).current_value for i in range (len(self.population)): result.append(max_nodes - self.population[i].calculateHeuristic()) total = sum(result) if total == 0: return [1 / self.population_size] * self.population_size for i in range (len(self.population)): temp = result[i] result[i] = temp / total return result </pre>	

d. Fungsi selection

Deskripsi	Fungsi untuk Membuat interval kumulatif berdasarkan fitness populasi.
<pre> def selection(self): intervals = self.createInterval() random_number = random.random() </pre>	

```
for i, (start, end) in enumerate(intervals):
    if start <= random_number < end:
        return self.population[i].cube
```

e. Fungsi crossover

Deskripsi

Fungsi untuk melakukan persilangan (crossover) antara dua individu (parent) untuk menghasilkan dua anak (offspring).

```
def crossover(self, parent1, parent2):
    cutting_point_x = random.randint(0, 4)
    cutting_point_y = random.randint(0, 4)
    cutting_point_z = random.randint(0, 4)

    offspring1 = [[[0 for k in range (5)] for j in
range (5)] for i in range (5)]
    offspring2 = [[[0 for k in range (5)] for j in
range (5)] for i in range (5)]

    for i in range (5):
        for j in range (5):
            for k in range (5):
                if (i < cutting_point_x) or (i ==
cutting_point_x and j < cutting_point_y) or (i ==
cutting_point_x and j == cutting_point_y and k <=
cutting_point_z):
                    offspring1[i][j][k] =
parent1[i][j][k]
                    offspring2[i][j][k] =
parent2[i][j][k]
                else:
                    offspring1[i][j][k] =
parent2[i][j][k]
```

```

                                offspring2[i][j][k] =
parent1[i][j][k]
                                return offspring1, offspring2

```

f. Fungsi mutation

Deskripsi	Fungsi untuk melakukan mutasi pada individu offspring dengan cara menukar elemen-elemen pada kubus untuk meningkatkan variasi dan menghindari local optima
<pre> def mutation(self, offspring): Utility.swapElement(offspring) </pre>	

g. Fungsi createChild

Deskripsi	Fungsi untuk membuat dua individu offspring dengan memilih dua parent menggunakan metode seleksi dan menerapkan crossover serta mutasi pada offspring
<pre> def createChild(self, fitness_scores): parent1 = self.selection() parent2 = self.selection() while (not parent1) or (not parent2): parent1 = self.selection() parent2 = self.selection() offspring1, offspring2 = self.crossover(parent1, parent2) if random.random() < self.mutation_rate: self.mutation(offspring1) self.mutation(offspring2) return offspring1, offspring2 </pre>	

h. Fungsi solveGeneticAlgorithm

Deskripsi	Fungsi utama untuk menjalankan pencarian solusi menggunakan Genetic Algorithm.
------------------	--

```
def solveGeneticAlgorithm(self):
    optimum_value = float('inf')
    for i in range (self.max_iterations):
        new_generation = []
        fitness_scores =
self.calculatePopulationFitness()

        with ThreadPoolExecutor(max_workers=20) as
executor:
            for _ in range(self.population_size):
                results = list(executor.map(lambda _:
self.createChild(fitness_scores),
range(self.population_size // 2)))

            for offspring1, offspring2 in results:
                new_generation.append(Node(offspring1))
                new_generation.append(Node(offspring2))
            best_individual = min(new_generation,
key=lambda node: node.calculateHeuristic())
            best_fitness =
best_individual.calculateHeuristic()
            print(f"Generasi {i + 1}: Fitness terbaik =
{best_fitness}")
            self.history.append({"frame": i + 1, "cube":
best_individual.cube, "objective_value": best_fitness})
            if best_fitness == 0:
                print("Solusi optimal ditemukan!")
                return best_individual
            self.population = new_generation
            if best_fitness < optimum_value:
                optimum_value = best_fitness
            print("Jumlah iterasi maksimum tercapai.")
            print("Maximum value: ", optimum_value)
            return best_individual
```

3.3.7. Lain-lain

Untuk membantu pengembangan tiap algoritma dibuat dua kelas yaitu kelas utility dan kelas Node yang menyimpan kondisi state kubus saat ini sebagai berikut:

3.3.7.1. Utility

a. Fungsi magicNumber

Deskripsi	Fungsi untuk menghitung magic number dari kubus
<pre>def magicNumber(cube_size): return (cube_size * (cube_size**3 + 1)) / 2</pre>	

b. Fungsi objectiveFunction

Deskripsi	Fungsi untuk menghitung fungsi objektif dari suatu state
<pre>def objectiveFunction(cube, magic_number): obejective_cost = 0 n = 5 main_diagonal_1 = 0 main_diagonal_2 = 0 diagonal_ltr = 0 diagonal_rtl = 0 diagonal_ttb = 0 diagonal_btt = 0 for i in range(n): for j in range(n): row_sum = 0</pre>	

```
        col_sum = 0
        pillar_sum = 0
        for k in range(n):
            row_sum += cube[i][j][k]
            col_sum += cube[i][k][j]
            pillar_sum += cube[k][i][j]

        obejective_cost += abs(row_sum -
magic_number)
        obejective_cost += abs(col_sum -
magic_number)
        obejective_cost += abs(pillar_sum -
magic_number)

        main_diagonal_1 += cube[i][i][i]
        main_diagonal_2 += cube[i][i][n - i - 1]
        diagonal_ltr += cube[i][n - i - 1][i]
        diagonal_rtl += cube[n - i - 1][i][i]
        diagonal_ttb += cube[i][i][i]
        diagonal_btt += cube[n - i - 1][i][i]

        obejective_cost += abs(main_diagonal_1 -
magic_number)
        obejective_cost += abs(main_diagonal_2 -
magic_number)
        obejective_cost += abs(diagonal_ltr - magic_number)
        obejective_cost += abs(diagonal_rtl - magic_number)
        obejective_cost += abs(diagonal_ttb - magic_number)
        obejective_cost += abs(diagonal_btt - magic_number)

    return obejective_cost
```

c. Fungsi RandomCube

Deskripsi	Fungsi untuk membangkitkan state kubus secara random
<pre> def RandomCube(cube_size, magic_number): cube = [[[0 for k in range (cube_size)] for j in range (cube_size)] for i in range (cube_size)] num = [i for i in range(1,126)] num.remove(63) # hapus 63 karena nanti gak masuk random for i in range (cube_size): for j in range (cube_size): for k in range (cube_size): if i==2 and j==2 and k==2: cube[i][j][k] = 63 else: randomidx= random.randint(0, len(num)-1) randomnum = num[randomidx] num.pop(randomidx) cube[i][j][k] = randomnum return cube </pre>	

d. Fungsi SwapCubeValue

Deskripsi	Fungsi untuk menukar 2 elemen pada kubus yang digunakan untuk membangkitkan state baru
<pre> def swapCubeValue(cube, pos1, pos2): cubetemp = cube i1, j1, k1 = Utility.postoiijk(pos1) </pre>	


```

i2, j2, k2 = Utility.postoijk(pos2)

while (i1==2 and j1==2 and k1==2) or (i1==i2 and
j1==j2 and k1==k2):
    i1, j1, k1 = Utility.postoijk(pos1)
    while (i2==2 and j2==2 and k2==2) :
        i1, j1, k1 = Utility.postoijk(pos2)

# tukar posisi angka
temp = cubetemp[i1][j1][k1]
cubetemp[i1][j1][k1] = cubetemp[i2][j2][k2]
cubetemp[i2][j2][k2] = temp

return cubetemp

```

e. Fungsi eliminateRandomState

Deskripsi	Fungsi heuristik untuk mengurangi banyak state pencarian dengan menggunakan angka pada posisi simetris sentral
<pre> def eliminateRandomState(cube): if cube[2][2][0] + cube[2][4][4] == 126 or cube[2][2][0] + cube[2][2][4] == 126 or cube[2][0][2] + cube[2][4][2] == 126 or cube[2][0][4] + cube[2][4][0] == 126 : return True return False </pre>	

3.3.7.2. Node

a. Fungsi init

Deskripsi	Fungsi untuk menyimpan kondisi state
<pre>def __init__(self, cube=None, cube_size=5): self.magic_number = Utility.magicNumber(cube_size) if cube is not None: self.cube = cube else: self.cube = Utility.generateRandomCube(cube_size, self.magic_number // 2) self.current_value = Utility.objectiveFunction(self.cube, self.magic_number)</pre>	

b. Fungsi showCube

Deskripsi	Fungsi untuk menampilkan state
<pre>def showCube(self): for i in range (len(self.cube)): for j in range (len(self.cube)): for k in range (len(self.cube)): print(self.cube[i][j][k], end=' ') print() print() print()</pre>	

c. Fungsi getHighestSuccessor

Deskripsi	Fungsi untuk mendapatkan state tetangga yang memiliki nilai objektif yang paling tinggi
<pre>def getHighestSuccessor(self):</pre>	

```
bestcube = self.cube
value = self.current_value

for i in range (125):
    for j in range (125):
        if i==j or i==62 or j==62:
            continue
        newcube = Utility.swapCubeValue(self.cube, i,
j)
        newval = Utility.objectiveFunction(self.cube,
315)

        if newval < value:
            value = newval
            bestcube = newcube

newNode = Node(bestcube)
return newNode
```

d. Fungsi getRandomSuccessor

Deskripsi	Fungsi untuk mendapatkan state tetangga secara acak, digunakan untuk algoritma stochastic dan simulated annealing.
<pre>def getRandomSuccessor(self): random1 = random.randint(0, 124) random2 = random.randint(0, 124) while random1==random2 or random1==62 or random2==62: random1 = random.randint(0, 124) random2 = random.randint(0, 124)</pre>	

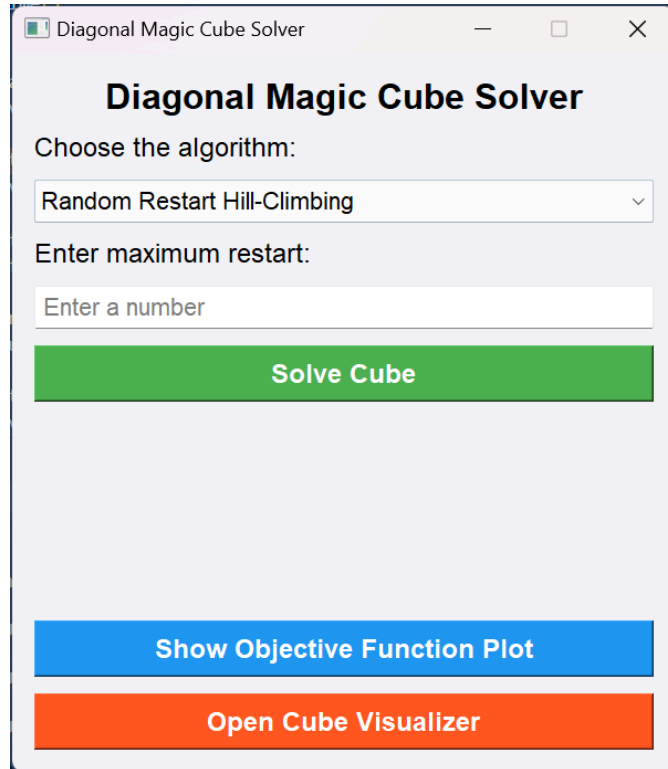
```
newcube = Utility.swapCubeValue(self.cube, random1,  
random2)  
  
newNode = Node(newcube)  
return newNode
```

3.4. Tampilan Antarmuka

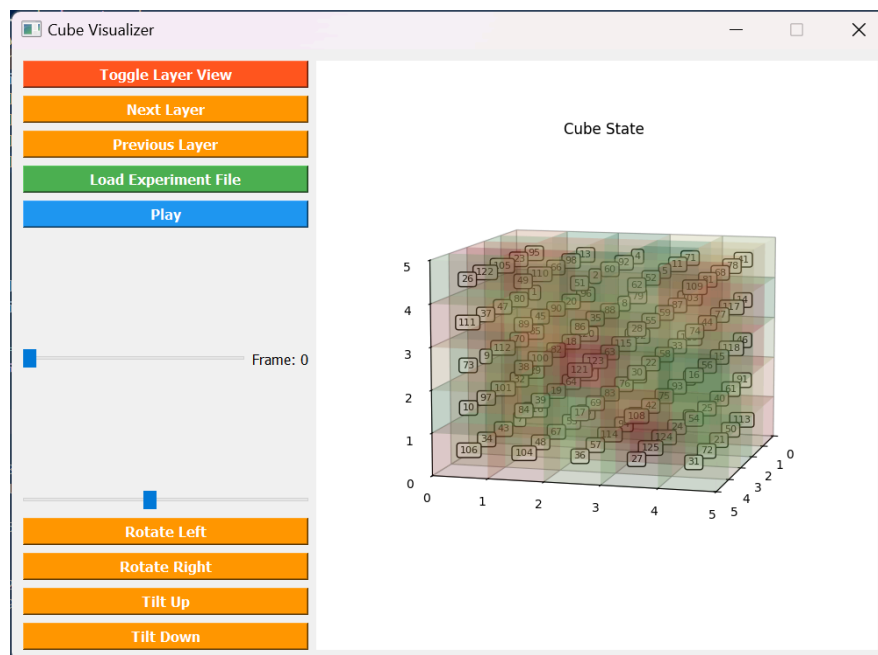
Untuk tampilan antarmuka, pembuatan GUI menggunakan PyQt dan matplotlib untuk memvisualisasikan objektif plot pada setiap statenya. Untuk video player yang menampilkan kondisi setiap state juga menggunakan PyQt dan matplotlib dalam pembuatannya dan video player akan menerima file input dengan format json. Tampilan antarmuka disimpan pada dua kelas yakni, CubeSolverApp sebagai main program dan CubeVisualizer sebagai kelas untuk menampilkan video player.

Saat menjalankan program, pengguna dapat memilih algoritma yang akan digunakan untuk melakukan proses pencarian beserta jumlah iterasi yang diinginkan. Setelah selesai melakukan pencarian program akan menampilkan waktu yang dibutuhkan untuk melakukan pencarian. Pengguna juga bisa melihat grafik plot fungsi objektif dengan menekan tombol “Show Objective Function Plot” dan menampilkan visualisasi perubahan antara state dengan menekan tombol “Open Cube Visualizer”.

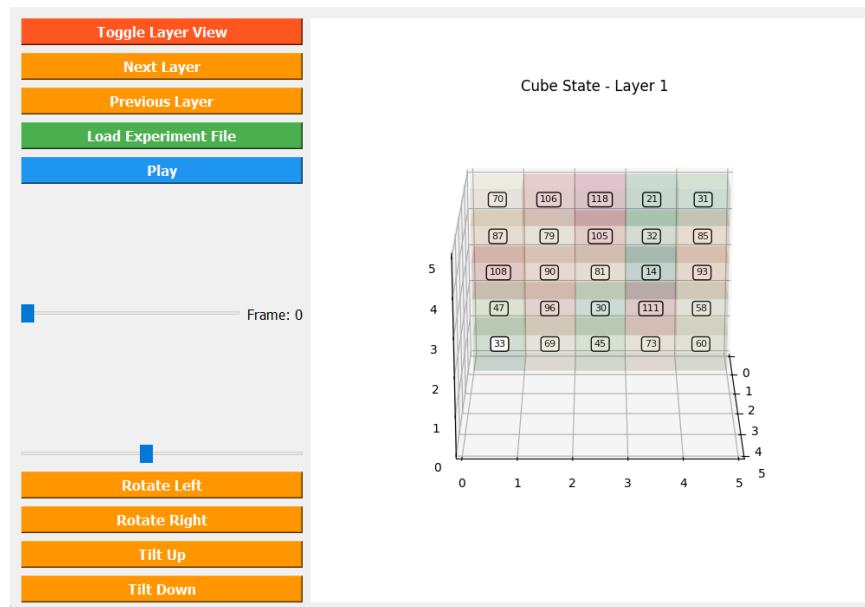
Pada *video player*, pengguna dapat memasukkan input file json dari hasil pencarian yang baru saja dilakukan. File json tersebut otomatis akan tersimpan saat program selesai melakukan pencarian. Video player akan menampilkan perubahan pada setiap statenya. Pengguna dapat mengatur kecepatan video dan berpindah frame dengan menggeser toggle frame yang ada. Pengguna juga bisa memutar tampilan kubus dengan menekan tombol yang ada atau langsung *men-drag* kubus yang ada pada tampilan.



Gambar 3.4.1 Tampilan Antarmuka Menu



Gambar 3.4.2 Tampilan Antarmuka Video Player

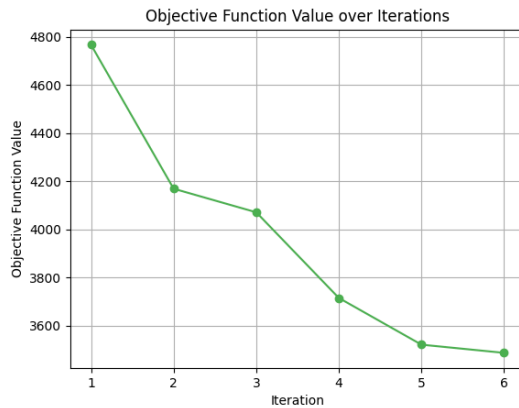


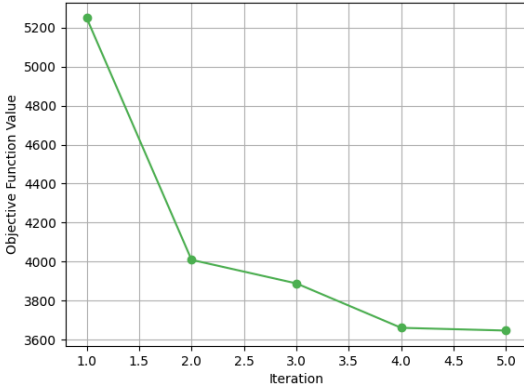
Gambar 3.4.3 Tampilan Antarmuka Layer Cube pada Video Player

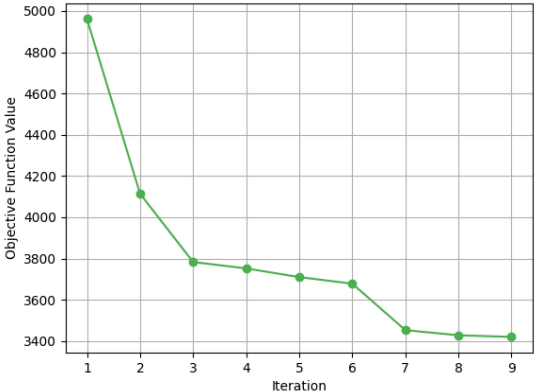
3.5. Hasil Pengujian

3.5.1. Pengujian Steepest Ascent Hill-Climbing

Tabel 3.5.1.1 Hasil Pengujian Steepest Ascent Hill-Climbing

State Awal	State Akhir	Nilai Objektif	Plot nilai Objektif	Durasi	Banyak Iterasi														
<p>Initial state of the cube:</p> <p>9 70 20 5 49 25 103 15 61 90 78 120 82 86 14 3 108 69 50 7 79 111 21 27 72</p> <p>37 91 92 32 13 34 106 100 28 45 57 123 29 1 96 31 80 47 22 65 119 62 104 112 40</p> <p>83 54 23 24 117 18 11 122 124 41 59 75 63 26 116 39 73 81 44 88 114 84 98 33 67</p>	<p>Final state of the cube:</p> <p>70 61 78 20 5 49 25 103 15 90 120 82 86 14 3 69 50 7 111 21 27 72 37 91 92</p> <p>32 13 34 106 100 28 45 57 123 29 1 96 31 80 47 22 65 119 62 104 112 40 83 54 23</p> <p>24 117 18 11 122 124 41 59 75 26 116 39 63 73 81 44 88 114 84 98 67 33 108 125 101</p>	3712	 <table><caption>Objective Function Value over Iterations</caption><thead><tr><th>Iteration</th><th>Objective Function Value</th></tr></thead><tbody><tr><td>1</td><td>4750</td></tr><tr><td>2</td><td>4150</td></tr><tr><td>3</td><td>4050</td></tr><tr><td>4</td><td>3700</td></tr><tr><td>5</td><td>3550</td></tr><tr><td>6</td><td>3500</td></tr></tbody></table>	Iteration	Objective Function Value	1	4750	2	4150	3	4050	4	3700	5	3550	6	3500	5.58 seconds	6
Iteration	Objective Function Value																		
1	4750																		
2	4150																		
3	4050																		
4	3700																		
5	3550																		
6	3500																		

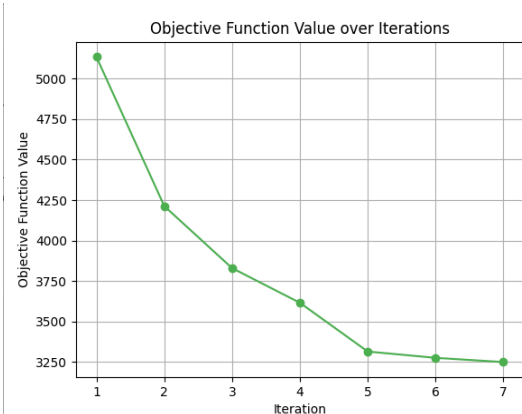
<div><div>125 101 89 17 38</div><div>115 118 99 58 30</div><div>109 2 4 76 74</div><div>46 97 60 52 93</div><div>51 66 35 77 36</div><div>53 48 55 121 105</div><div>94 16 113 68 56</div><div>12 85 87 43 107</div><div>71 8 110 42 6</div><div>19 10 64 102 95</div></div>	<div><div>89 17 38 115 118</div><div>99 58 30 109 2</div><div>4 76 74 46 97</div><div>60 52 93 51 66</div><div>35 77 36 53 48</div><div>55 121 105 94 16</div><div>113 68 56 12 85</div><div>87 43 107 71 8</div><div>110 42 6 19 10</div><div>64 102 95 9 79</div></div>																
<div><div>Initial state of the cube:</div><div>107 25 80 94 86</div><div>35 76 70 26 24</div><div>14 98 105 2 32</div><div>16 19 84 1 12</div><div>72 67 31 106 111</div><div>96 79 124 7 99</div><div>17 57 104 62 114</div><div>45 36 119 65 61</div><div>50 47 38 69 112</div><div>53 44 5 56 90</div><div>103 87 27 28 93</div></div>	<div><div>Final state of the cube:</div><div>38 47 69 112 53</div><div>44 5 56 90 103</div><div>87 27 28 21 17</div><div>50 117 39 78 29</div><div>116 15 97 20 122</div><div>85 49 100 8 77</div><div>121 40 10 120 11</div><div>3 64 37 118 123</div><div>58 71 42 51 113</div><div>33 115 59 68 88</div><div>74 125 18 60 95</div></div>	3436	<div><div>Objective Function Value over Iterations</div><table><thead><tr><th>Iteration</th><th>Objective Function Value</th></tr></thead><tbody><tr><td>1.0</td><td>5200</td></tr><tr><td>2.0</td><td>4000</td></tr><tr><td>3.0</td><td>3880</td></tr><tr><td>4.0</td><td>3680</td></tr><tr><td>5.0</td><td>3650</td></tr></tbody></table></div>	Iteration	Objective Function Value	1.0	5200	2.0	4000	3.0	3880	4.0	3680	5.0	3650	3.96 seconds	5
Iteration	Objective Function Value																
1.0	5200																
2.0	4000																
3.0	3880																
4.0	3680																
5.0	3650																

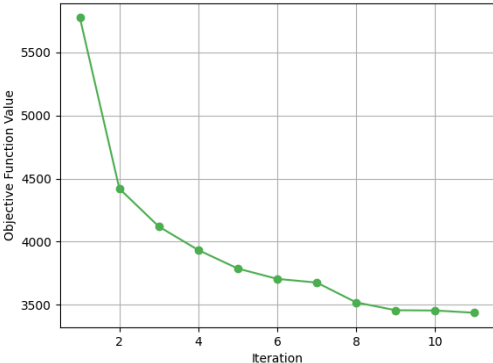
<div>21 117 39 78 29 116 15 63 97 20 122 85 49 41 100 8 77 121 40 10 120 11 3 64 37 118 123 58 71 42 51 113 33 115 59 68 88 74 125 18 60 95 54 102 43 6 66 75 73 48 81 92 30 82 101 4 46 89 108 52 34 91 55 23 22 110 13 9 109 83</div>	<div>54 102 43 6 66 75 73 63 48 81 92 30 82 101 4 46 89 108 52 34 91 55 23 22 110 13 9 109 83 107 93 41 25 80 94 86 35 76 70 26 24 14 98 105 2 32 16 19 84 1 12 72 67 31 106 111 96 79 124 7 99 57 104 62 114 45 36 119 65 61</div>																								
<div>Initial state of the cube: 33 47 108 87 70 69 96 90 79 106 45 30 81 105 118 73 111 14 32 21 60 58 93 85 31 37 52 17 61 97 103 123 15 29 46 34 65 120 125 24</div>	<div>Final state of the cube: 26 30 69 96 79 106 45 81 105 118 73 111 14 32 21 60 58 93 85 31 37 52 17 61 97 123 15 29 46 34 65 120 125 24 119 83 1 110 74 66</div>	3420	<div>Objective Function Value over Iterations <table border="1"><thead><tr><th>Iteration</th><th>Objective Function Value</th></tr></thead><tbody><tr><td>1</td><td>4950</td></tr><tr><td>2</td><td>4100</td></tr><tr><td>3</td><td>3780</td></tr><tr><td>4</td><td>3750</td></tr><tr><td>5</td><td>3720</td></tr><tr><td>6</td><td>3680</td></tr><tr><td>7</td><td>3450</td></tr><tr><td>8</td><td>3430</td></tr><tr><td>9</td><td>3420</td></tr></tbody></table></div>	Iteration	Objective Function Value	1	4950	2	4100	3	3780	4	3750	5	3720	6	3680	7	3450	8	3430	9	3420	8.39 seconds	9
Iteration	Objective Function Value																								
1	4950																								
2	4100																								
3	3780																								
4	3750																								
5	3720																								
6	3680																								
7	3450																								
8	3430																								
9	3420																								

119 83 1 110 74 66 22 121 10 67	22 121 10 67 82 39 54 117 84 50				
82 39 54 117 84 50 68 102 20 122 51 13 63 19 75 115 41 88 16 44 124 5 109 23 2	68 102 20 122 51 13 19 75 115 41 88 16 63 44 124 5 109 23 2 91 48 98 11 94 38				
91 48 98 11 94 38 4 100 71 78 53 80 25 104 6 99 27 95 18 55 86 77 114 3 42	4 100 71 78 53 80 25 6 99 27 95 18 55 86 77 114 3 42 40 92 9 59 101 57 64				
40 92 9 59 101 57 64 36 62 113 49 35 12 43 28 8 112 56 107 89 76 72 116 26 7	36 62 113 49 35 12 43 28 8 112 56 107 89 76 72 116 7 87 104 47 108 103 90 33 70				

3.5.2. Pengujian Sideways Move Hill-Climbing

Tabel 3.5.2.1 Hasil Pengujian Sideways Move Hill-Climbing

State Awal	State Akhir	Nilai Objektif	Plot nilai Objektif	Durasi	Banyak Iterasi	Maksimal Sideways move																
<div>Initial state of the cube:</div> <div>40 48 61 93 75 19 12 21 109 89 125 88 91 28 53 29 66 33 77 95 114 84 83 69 51</div> <div>80 8 107 112 74 94 104 56 27 32 44 24 10 38 121 45 55 43 87 70 18 116 96 49 15</div> <div>20 59 22 17 31 65 102 82 79 37 35 122 63 78 7 36 9 25 97 105 106 23 113 111 123</div> <div>92 13 16 4 117 42 52 108 110 90</div>	<div>Final state of the cube:</div> <div>116 107 30 43 18 57 65 82 68 27 91 99 66 85 87 5 119 83 19 98 40 38 78 110 73</div> <div>10 26 53 45 61 13 92 67 104 35 95 50 77 84 113 46 3 9 81 89 122 55 100 20 23</div> <div>25 94 108 103 101 117 22 72 106 96 90 118 63 14 36 71 29 109 6 114 88 54 4 121 33</div> <div>17 8 39 7 125 64 69 58 15 76 11 16 105 56 49 51 75 111 74 12</div>	3250	<div>Objective Function Value over Iterations</div>  <table border="1"><thead><tr><th>Iteration</th><th>Objective Function Value</th></tr></thead><tbody><tr><td>1</td><td>5000</td></tr><tr><td>2</td><td>4200</td></tr><tr><td>3</td><td>3850</td></tr><tr><td>4</td><td>3650</td></tr><tr><td>5</td><td>3300</td></tr><tr><td>6</td><td>3280</td></tr><tr><td>7</td><td>3250</td></tr></tbody></table>	Iteration	Objective Function Value	1	5000	2	4200	3	3850	4	3650	5	3300	6	3280	7	3250	9.28 seconds	7	100
Iteration	Objective Function Value																					
1	5000																					
2	4200																					
3	3850																					
4	3650																					
5	3300																					
6	3280																					
7	3250																					

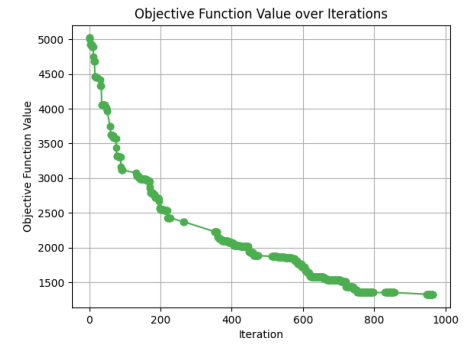
73 67 68 76 115 99 54 101 62 50 3 64 86 6 98 58 100 103 5 41 118 57 72 11 47 60 34 30 81 124 85 71 119 26 1 14 46 2 120 39	115 79 2 42 41 60 97 32 80 86 48 1 93 62 120 52 47 28 102 37 123 70 24 34 21 31 124 44 59 112																													
Intial state of the cube: 75 69 92 2 109 57 55 35 101 125 16 18 12 22 38 47 60 98 43 112 1 111 121 113 21 83 97 124 71 78 45 67 90 114 50 26 14 51 37 79 62 40 10 108 80 118 96 85 30 13 42 100 7 58 6 110 94 23 117 3 56 15 63 68 72	Final state of the cube: 51 81 103 5 93 21 58 47 86 112 35 28 67 48 24 107 117 98 99 11 15 19 37 9 13 119 26 55 25 88 70 45 6 42 59 76 71 90 114 38 65 113 32 96 44 31 73 122 61 121 30 33 74 125 29 108 110 120 40 39	3438	<div>Objective Function Value over Iterations</div>  <table border="1"><thead><tr><th>Iteration</th><th>Objective Function Value</th></tr></thead><tbody><tr><td>1</td><td>5700</td></tr><tr><td>2</td><td>4400</td></tr><tr><td>3</td><td>4150</td></tr><tr><td>4</td><td>3950</td></tr><tr><td>5</td><td>3800</td></tr><tr><td>6</td><td>3700</td></tr><tr><td>7</td><td>3650</td></tr><tr><td>8</td><td>3550</td></tr><tr><td>9</td><td>3450</td></tr><tr><td>10</td><td>3420</td></tr><tr><td>11</td><td>3400</td></tr></tbody></table>	Iteration	Objective Function Value	1	5700	2	4400	3	4150	4	3950	5	3800	6	3700	7	3650	8	3550	9	3450	10	3420	11	3400	14.5 8 seconds	11	100000
Iteration	Objective Function Value																													
1	5700																													
2	4400																													
3	4150																													
4	3950																													
5	3800																													
6	3700																													
7	3650																													
8	3550																													
9	3450																													
10	3420																													
11	3400																													

84 89 28 119 122 120 77 39 29 33 95 4 52 32 70 48 64 19 49 91 123 9 36 99 46 27 87 54 59 103 24 116 11 74 65 105 44 61 20 31 66 34 107 106 82 102 81 17 104 93 8 25 5 41 115 53 76 88 73 86	123 10 63 36 104 23 46 53 56 118 109 57 52 87 4 95 2 69 106 84 92 111 85 14 8 41 7 83 80 100 22 49 1 72 116 78 62 89 66 91 17 60 43 54 94 18 20 16 77 27 102 75 79 3 101 115 97 124 82 12 68 105 34 50 64															
Initial state of the cube: 123 43 98 14 96 25 29 31 97 71 77 72 69 87 40 125 70 117 39 115 12 38 93 13 101 107 116 89 73 113	Final state of the cube: 43 105 115 23 62 92 52 55 46 68 118 64 17 97 82 42 41 101 99 33 75 60 15 74 34 4 6 72 5 90 108 100 81 85 14 73 69 27 125 117	3557	<div>Objective Function Value over Iterations</div> <table><thead><tr><th>Iteration</th><th>Objective Function Value</th></tr></thead><tbody><tr><td>1.0</td><td>4600</td></tr><tr><td>2.0</td><td>3900</td></tr><tr><td>3.0</td><td>3850</td></tr><tr><td>4.0</td><td>3557</td></tr></tbody></table>	Iteration	Objective Function Value	1.0	4600	2.0	3900	3.0	3850	4.0	3557	5.36 seconds	4	20
Iteration	Objective Function Value															
1.0	4600															
2.0	3900															
3.0	3850															
4.0	3557															

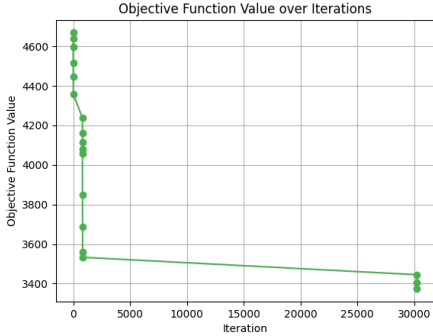
88 47 114 55 5	36 119 9 106 12					
67 50 46 37 91	3 29 111 124 89					
95 33 82 100 86						
18 90 21 6 109	56 103 107 13 44					
	2 61 80 110 65					
81 44 53 30 104	47 95 63 83 58					
10 112 54 84 20	70 45 32 7 104					
23 61 63 56 58	88 37 51 114 112					
26 1 68 49 41						
22 28 103 59	98 10 59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

3.5.3. Pengujian Stochastic Hill-Climbing

Tabel 3.5.3.1 Hasil Pengujian Stochastic

State Awal	State Akhir	Nilai Objektif Terbaik	Plot nilai Objektif	Durasi (s)	Banyak Iterasi
78 24 20 110 32 59 121 57 69 50 16 25 13 65 3 9 66 99 54 96 82 86 70 36 55 113 108 14 122 104 117 105 29 12 89 103 92 116 27 21 77 61 17 93 5 4 68 76 48 33 53 112 81 75 73 120 111 28 22 125 41 115 63 119 58 106 52 8 97 95 109 94 98 71 10 45 44 124 62 23 18 91 43 2 88 114 60 87 6 46 39 51 118 100 107 85 67 37 79 74 1 101 19 84 30 35 123 64 47 80 102 42 15 40 72 49 7 56 83 34 31 90 11 38 26	32 44 14 87 21 27 91 17 71 96 66 29 22 51 117 75 82 31 76 8 93 39 5 78 52 86 103 16 20 48 101 37 110 83 92 77 84 3 56 6 18 67 13 94 50 60 45 26 35 115 113 104 88 69 102 2 30 72 41 79 98 124 63 53 49 70 108 68 119 47 40 1 125 116 59 42 11 62 105 95 12 23 100 25 4 24 43 33 112 65 106 85 55 89 7 114 61 46 15 36 28 64 118 9 10 73 120 54 121 107 34 38 74 57 90 123 97 19 58 111 122 81 80 109 99	3339		32.83	5000

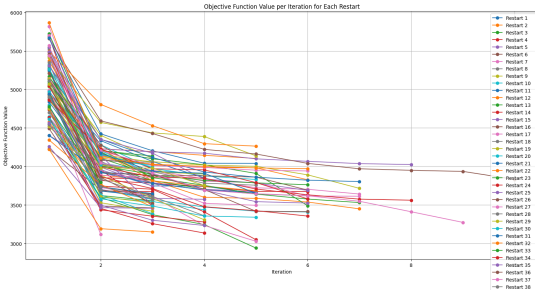
<div><div>100 38 123 99 79</div><div>85 11 4 111 34</div><div>113 28 5 10 108</div><div>93 77 106 54 41</div><div>45 14 81 80 105</div></div> <div><div>119 120 112 39 117</div><div>90 43 16 17 88</div><div>73 7 98 83 87</div><div>96 33 15 12 122</div><div>78 20 69 64 52</div></div> <div><div>60 37 104 110 116</div><div>70 72 21 44 68</div><div>36 89 63 46 9</div><div>76 13 97 30 65</div><div>57 109 22 32 48</div></div> <div><div>2 49 91 55 27</div><div>125 1 101 118 82</div><div>24 50 59 62 47</div><div>124 18 56 35 102</div><div>61 95 67 19 3</div></div> <div><div>6 84 94 86 51</div><div>58 23 29 74 31</div><div>114 40 53 25 103</div><div>71 26 66 8 115</div><div>92 42 107 75 121</div></div>	<div><div>40 96 8 74 19</div><div>88 52 69 3 102</div><div>100 82 71 95 22</div><div>39 59 58 97 91</div><div>5 104 89 49 85</div></div> <div><div>119 38 43 14 25</div><div>46 37 27 47 54</div><div>120 29 31 77 68</div><div>60 78 61 81 111</div><div>106 15 62 123 18</div></div> <div><div>4 12 112 35 73</div><div>44 108 114 24 110</div><div>32 42 63 99 79</div><div>75 80 36 17 118</div><div>93 83 10 50 125</div></div> <div><div>105 67 86 41 55</div><div>53 113 21 30 34</div><div>11 117 76 16 124</div><div>116 9 23 7 33</div><div>57 90 64 2 101</div></div> <div><div>20 51 122 107 87</div><div>6 92 84 103 115</div><div>56 98 13 26 48</div><div>109 45 1 70 66</div><div>28 121 65 94 72</div></div>	3358	<div><div>Objective Function Value over Iterations</div><table><thead><tr><th>Iteration</th><th>Objective Function Value</th></tr></thead><tbody><tr><td>0</td><td>5500</td></tr><tr><td>10</td><td>5300</td></tr><tr><td>20</td><td>5100</td></tr><tr><td>30</td><td>4900</td></tr><tr><td>40</td><td>4700</td></tr><tr><td>50</td><td>4500</td></tr><tr><td>60</td><td>4300</td></tr><tr><td>100</td><td>4300</td></tr><tr><td>200</td><td>4000</td></tr><tr><td>700</td><td>4000</td></tr><tr><td>750</td><td>3358</td></tr></tbody></table></div>	Iteration	Objective Function Value	0	5500	10	5300	20	5100	30	4900	40	4700	50	4500	60	4300	100	4300	200	4000	700	4000	750	3358	69.49	10000
Iteration	Objective Function Value																												
0	5500																												
10	5300																												
20	5100																												
30	4900																												
40	4700																												
50	4500																												
60	4300																												
100	4300																												
200	4000																												
700	4000																												
750	3358																												

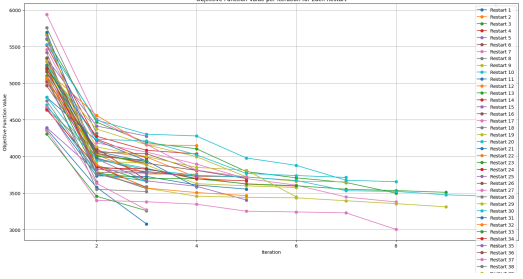
65 3 100 107 123 31 4 86 25 52 5 124 98 19 71 29 55 89 113 109 78 95 41 20 53 118 110 66 121 114 10 67 120 105 102 74 64 43 62 1 116 33 21 101 51 94 57 80 30 7 12 73 47 60 6 40 88 42 38 117 111 15 63 90 70 27 9 84 81 14 92 91 79 99 11 115 106 34 36 16 125 8 18 56 97 46 48 112 45 77 17 108 61 58 54 82 85 69 83 103 76 96 2 28 75 87 68 37 22 32 39 93 35 24 72 49 26 13 44 122 119 59 50 104 23	64 109 44 19 74 83 57 69 47 94 82 16 34 50 11 77 32 120 42 117 114 17 103 124 111 72 99 37 84 86 59 33 97 3 1 10 73 80 27 113 116 62 119 67 25 90 22 9 45 75 76 13 104 21 125 12 7 56 118 71 52 66 63 53 6 35 88 43 91 81 105 79 102 54 4 89 78 51 48 87 68 18 24 95 65 106 2 31 5 14 122 26 46 121 96 8 101 98 49 58 108 112 60 123 15 28 23 40 20 38 36 39 107 110 93 61 70 100 30 115 92 29 55 41 85	3373		129.09	50000
---	---	------	---	--------	-------

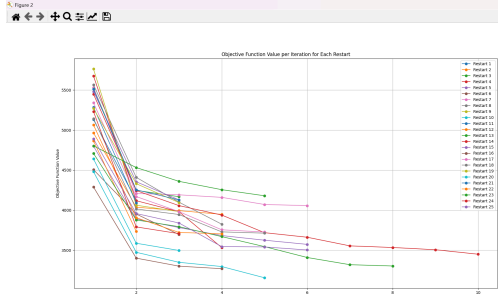
3.5.4. Pengujian Restart Hill-Climbing

Tabel 3.5.4.1 Hasil Pengujian Restart

State Awal	State Akhir	Nilai	Plot nilai Objektif	Durasi	Banyak
------------	-------------	-------	---------------------	--------	--------

		Objektif			Restart
80 61 81 5 72 124 78 111 26 118 24 83 105 94 19 67 15 95 32 117 7 35 123 119 70 92 106 11 120 87 12 42 69 112 91 88 45 97 86 90 55 25 17 54 107 74 34 58 66 14 57 44 73 125 85 100 9 10 46 108 62 82 63 110 64 71 60 99 48 51 76 20 104 122 43 2 56 75 39 22 29 109 28 79 40 68 30 49 59 113 31 33 18 89 4 50 13 23 1 101 98 47 103 41 21 114 27 3 84 65 96 6 115 36 16	26 47 113 43 88 89 56 60 53 27 6 34 54 18 8 76 93 46 119 1 69 80 25 124 96 4 91 28 87 57 112 125 42 7 106 13 37 111 83 11 66 48 82 52 85 99 9 73 97 16 105 21 2 67 120 59 121 36 86 38 50 64 63 109 30 35 71 115 58 104 62 61 117 24 41 39 74 98 95 3 84 32 116 68 110 100 72 51 17 31 33 75 5 92 107 15 70 77 19 90 122 101 123 14 65 55 12 49 108 102 40 10 22 103 118	2941		559.59	100 Best Restart: Restart 33, Iterations 5

38 77 53 121 37 102 52 8 116 93	79 29 44 20 78 81 114 94 45 23				
11 107 44 75 93 119 55 49 32 78 26 52 50 76 98 120 84 14 108 29 77 39 69 92 51 10 28 64 46 9 110 21 111 114 61 96 68 41 88 15 58 22 20 30 125 16 101 85 100 4 48 57 87 31 67 74 105 66 59 47 113 90 63 54 13 79 5 115 124 3 23 38 60 71 42 86 106 6 91 112 40 82 116 1 34 123 33 102 24 70 25 99 36 117 18 8 37 80 17 35 27 62 73 43 45 121 2 122 65 12 7 72 19 89 53	19 81 45 35 98 29 30 114 121 36 60 86 59 109 33 82 111 47 7 79 104 34 24 40 56 67 41 92 61 13 116 69 18 20 68 65 110 119 16 23 54 15 96 72 108 9 90 44 120 113 53 95 52 117 2 39 107 99 74 5 125 17 63 118 103 10 124 1 123 55 31 75 62 46 50 84 27 101 4 91 3 49 122 78 115 12 25 85 105 70 87 38 21 66 89 112 77 11 80 14	3003		695.06	50 Best Restart: Restart 27, Iterations 82

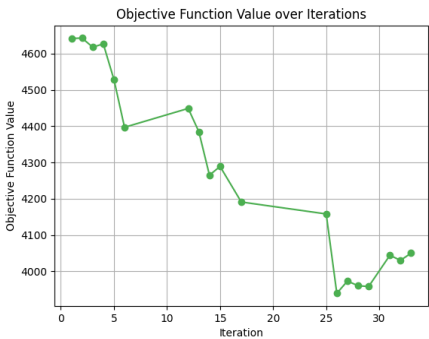
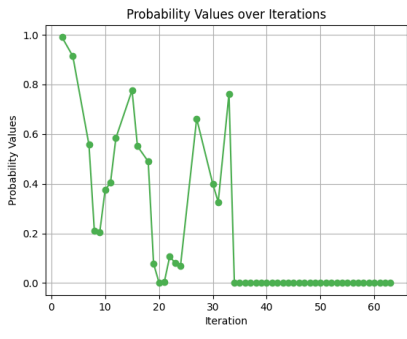
118 56 109 95 104 97 83 103 81 94	57 22 64 71 76 26 106 37 83 28 8 100 94 6 88 93 43 102 32 48 58 73 51 42 97				
94 27 26 90 85 53 12 8 101 112 32 96 25 16 68 89 86 57 30 28 24 99 48 117 13 35 58 113 2 46 103 115 114 43 19 100 118 41 84 119 21 76 55 7 121 109 75 98 93 17 80 62 66 34 10 9 51 11 67 120 47 23 63 104 20 29 95 97 49 107 40 122 60 33 91 52 92 37 73 54 4 65 82 14 18 45 22 116 77 44 50 81 39 72 110	104 90 35 48 70 61 44 81 41 111 77 64 107 54 10 39 15 27 55 78 51 113 46 102 21 106 20 69 17 110 65 43 53 112 29 36 3 58 98 103 19 119 47 40 92 75 97 24 52 115 31 96 66 118 87 74 16 11 49 89 124 50 63 42 82 26 57 83 32 6 22 85 76 33 86 100 94 84 45 13 1 108 116 9 117 114 56 8 38 109 99 88 28 18 12	3156		153.39	25 Best Restart: Restart 10, Iterations: 5

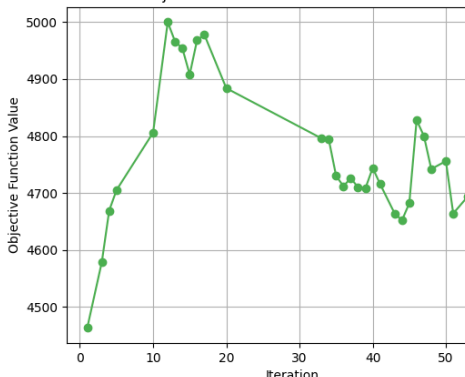
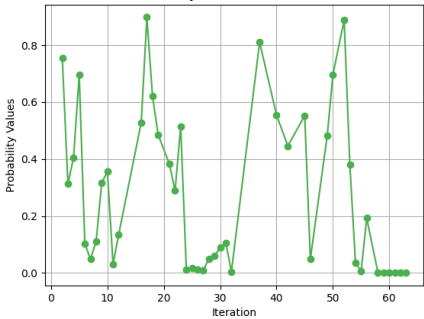
71 42 87 108 111	14 34 79 125 67				
102 38 83 6 5	120 37 59 93 23				
15 1 59 61 105	68 123 4 91 25				
36 88 56 123 106	30 95 121 62 7				
79 64 74 31 78	105 71 80 5 72				
3 70 124 69 125	122 2 60 101 73				

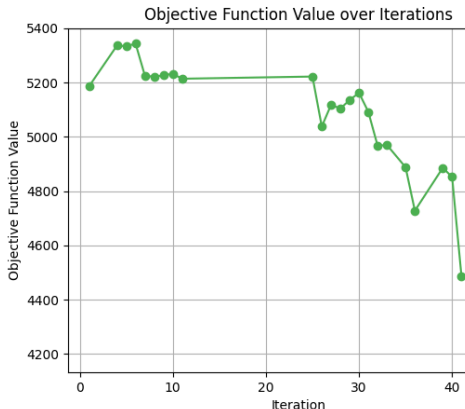
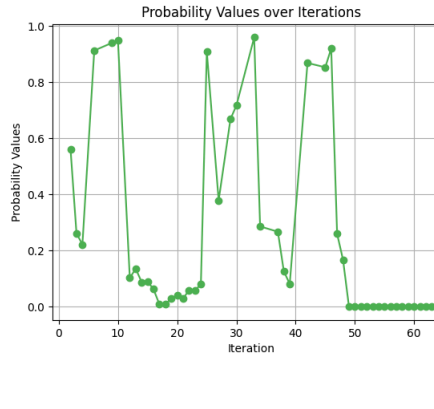
3.5.5. Pengujian Simulated Annealing

Pada simulated annealing akan dilakukan analisis pada 2 jenis scheduling saja, linear dan kuadratik dengan ukuran parameter temperatur dan cooling rate yang sama.

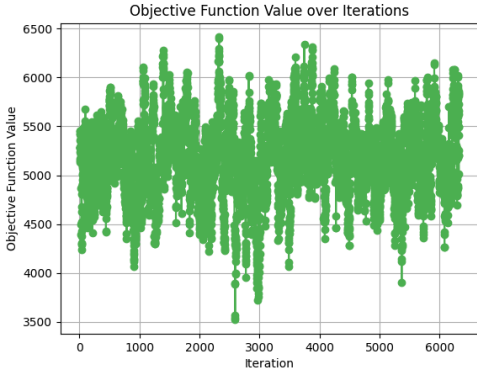
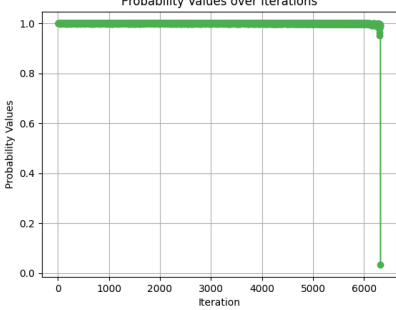
Tabel 3.5.5.1 Hasil Pengujian Simulated Annealing Linear Schedule (T=100 dan cooling rate = 0.05)

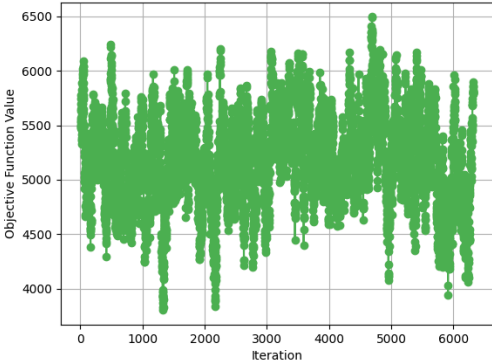
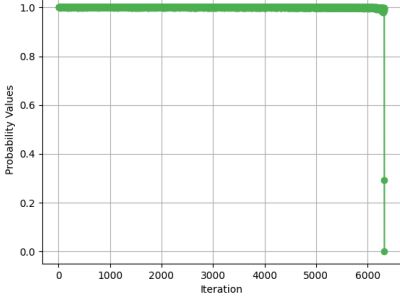
State Awal	State Akhir	Plot Objektif	Plot Probability	Durasi	Banyak Iterasi	Stuck Freq
103 37 4 124 20 34 6 12 111 125 76 120 93 25 88 92 14 108 69 56 68 73 26 82 55 100 81 106 44 59 112 121 38 19 43 66 16 61 117 27 31 32 113 75 95 115 107 97 94 53 10 24 74 45 22 50 48 49 41 29 39 96 63 46 67 47 33 119 23 118 85 21 52 60 65 98 105 40 86 116 42 54 18 8 11 84 15 2 80 90 57 91 70 83 3 1 51 102 109 110 9 78 36 13 72 122 87 28 30 64 71 99 79 7 89	101 29 54 51 105 39 66 88 77 22 19 32 89 100 120 62 45 36 75 81 93 113 83 16 3 42 85 79 6 65 74 91 34 90 107 115 86 64 28 116 38 98 58 73 110 21 96 94 104 124 61 119 2 99 67 59 87 55 97 14 114 125 63 20 1 48 26 117 82 10 112 12 106 9 123 72 25 78 41 49 50 35 4 80 31 111 40 92 71 30 76 15 37 108 68 53 18 70 109 95 44 52 47 7 13 23 17 27 84 8 60 57 46 69 118	 <p>Initial value: 4642.0 Final value: 4050.0</p>		0.122	64	51

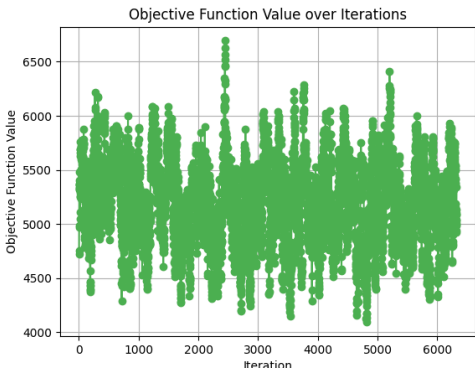
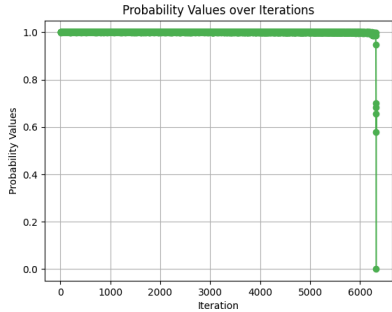
62 77 104 35 17 58 123 101 114 5	24 122 5 103 102 121 43 33 56 11					
94 84 100 50 67 17 73 33 46 86 105 45 55 59 107 111 2 22 52 31 9 18 108 121 56 36 76 82 44 81 116 109 25 49 96 7 51 87 70 19 118 110 102 113 64 95 4 57 39 125 103 54 66 53 38 122 72 98 65 71 27 68 63 85 47 93 42 40 112 61 104 30 60 89 74 90 1 28 97 79 58 6 24 120 123 26 101 21 88 48 80 62 5 13 41 14 8 99 15 119 75 10 23 117 43 34 20 78 37 124 114 77 83 16 12 29 115 3 32 11 35 69 106 91 92	94 51 105 50 67 17 59 33 24 35 61 90 55 77 22 111 2 57 52 36 9 82 73 56 121 31 32 6 44 39 116 69 25 74 103 85 18 8 19 68 115 110 60 92 64 95 58 87 99 5 46 15 66 76 4 122 72 98 102 27 71 28 63 100 53 93 89 40 112 79 81 108 65 42 41 45 124 101 97 104 48 3 80 114 49 26 107 21 88 20 70 123 125 13 62 75 96 7 78 119 14 84 23 117 43 34 38 109 37 11 10 30 83 16 12 29 118 1 47 120 86 54 106 91 113	<p>Objective Function Value over Iterations</p>  <p>Initial value: 4463.0 Final value: 4678.0</p>	<p>Probability Values over Iterations</p> 	0.162 3	64	47

67 42 124 48 2 75 82 4 100 58 41 109 6 95 7 69 38 78 46 15 81 26 88 87 66 117 89 3 24 80 28 1 54 21 18 106 122 73 96 56 105 91 14 59 102 123 33 61 111 119 57 32 113 22 64 51 40 65 101 110 31 12 63 83 47 39 74 30 108 37 43 45 13 90 85 29 16 68 79 77 44 10 50 93 107 8 120 112 27 62 115 17 97 20 86 99 5 125 104 94 55 71 49 103 84 70 72 36 53 9 116 76 92 60 25 11 98 118 19 34 23 114 35 121 52	86 37 114 48 2 35 1 101 67 71 10 87 11 57 40 77 82 8 14 93 69 26 88 109 18 56 53 104 27 118 31 117 55 19 92 110 59 125 96 16 34 74 50 30 24 111 15 78 41 61 122 42 123 22 64 20 43 29 47 13 115 45 63 70 89 94 52 33 79 39 91 28 100 81 113 62 60 106 112 44 75 119 9 85 107 108 90 54 84 36 17 46 120 68 21 124 105 6 66 5 116 73 49 102 25 38 72 98 83 3 51 76 23 103 99 65 32 12 4 80 58 97 95 121 7	 <p>Objective Function Value over Iterations</p> <p>Initial value: 5186.0 Final value: 4202.0</p>	 <p>Probability Values over Iterations</p>	0.170	64	
Rata-rata	Selisih avg = 509			0.222	64	

Tabel 3.5.5.2 Hasil Pengujian Simulated Annealing Linear Schedule (T=1000000 dan cooling rate = 0.05)

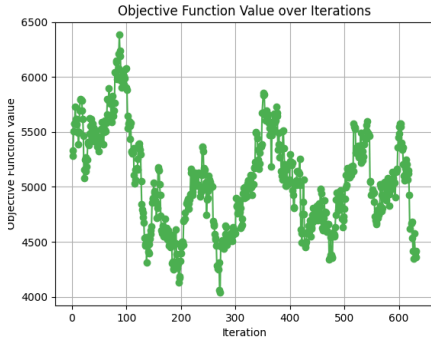
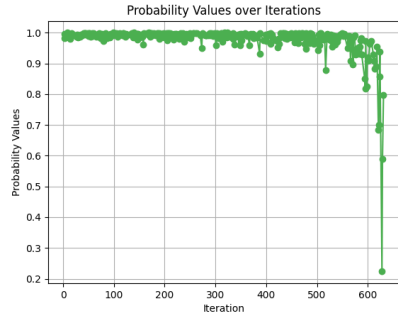
State Awal	State Akhir	Plot Objektif	Plot Probability	Durasi	Banyak Iterasi	Stuck Freq
123 55 27 109 66 2 73 121 85 72 57 100 22 71 124 30 10 103 39 45 119 7 61 54 64 106 122 23 91 53 117 86 47 104 50 113 101 16 69 93 48 17 1 107 19 88 44 108 102 6 111 12 37 35 20 68 81 11 25 14 21 87 63 31 105 15 62 74 82 8 83 33 5 28 46 3 76 52 41 95 90 34 70 43 26 80 67 116 40 110 96 49 29 125 24 36 89 65 79 78 38 4 13 92 51 112 84 97 56 118 59 120 75 60 18	123 55 27 109 66 2 73 121 85 72 57 100 22 71 124 30 10 103 39 45 119 7 61 54 64 106 122 23 91 53 117 86 47 104 50 113 101 16 69 93 48 17 1 107 19 88 44 108 102 6 111 12 37 35 20 68 81 11 25 14 21 87 63 31 105 15 62 74 82 8 83 33 5 28 46 3 76 52 41 95 90 34 70 43 26 80 67 116 40 110 96 49 29 125 24 36 89 65 79 78 38 4 13 92 51 112 84 97 56 118 59 120 75 60 18	 <p>Initial value: 5416.0 Final value: 5014.0</p>		9.97	6326	3169

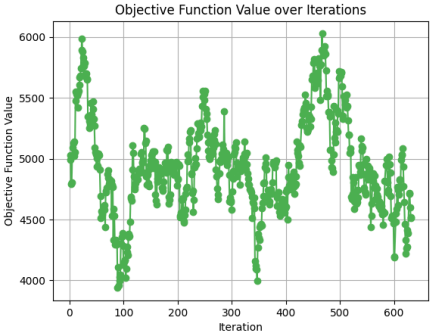
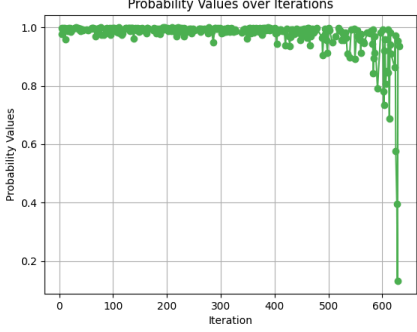
99 32 42 9 58 115 94 77 98 114	99 32 42 9 58 115 94 77 98 114					
27 78 104 59 90 45 89 5 52 30 120 101 37 4 73 58 111 31 77 114 105 25 18 50 28 55 20 41 16 107 91 67 35 115 62 118 108 7 49 19 10 40 92 64 72 34 14 42 110 80 79 29 32 48 95 81 109 57 102 15 23 24 63 33 66 112 13 106 85 121 43 39 46 68 103 12 94 86 84 74 9 70 75 69 93 116 53 76 88 56 8 17 36 44 96 124 38 122 87 113 71 22 61 26 125 6 1 100 83 65 21 51 119 117 97 123 60 2 47 3	27 78 104 59 90 45 89 5 52 30 120 101 37 4 73 58 111 31 77 114 105 25 18 50 28 55 20 41 16 107 91 67 35 115 62 118 108 7 49 19 10 40 92 64 72 34 14 42 110 80 79 29 32 48 95 81 109 57 102 15 23 24 63 33 66 112 13 106 85 121 43 39 46 68 103 12 94 86 84 74 9 70 75 69 93 116 53 76 88 56 8 17 36 44 96 124 38 122 87 113 71 22 61 26 125 6 1 100 83 65 21 51 119 117 97 123 60 2 47 3	<p>Objective Function Value over Iterations</p>  <p>Initial value: 5092.0 Final value: 4653.0</p>	<p>Probability Values over Iterations</p> 	9.98	6326	3169

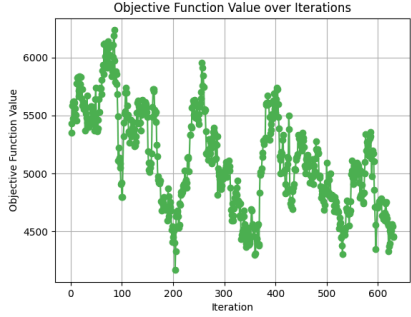
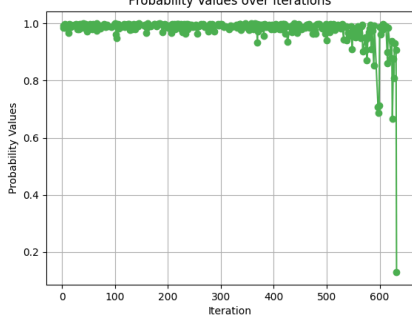
11 99 82 98 54	11 99 82 98 54					
14 89 105 94 45 62 112 85 8 111 71 48 3 110 87 72 12 39 47 88 106 51 102 116 118 68 77 9 35 15 4 109 26 76 107 66 81 1 119 21 36 61 121 55 50 98 28 56 41 18 30 104 22 24 80 120 46 32 86 27 113 124 63 44 114 33 19 69 125 117 2 74 42 95 13 70 57 5 84 96 92 90 59 16 6 20 10 7 49 38 58 34 23 83 37 122 52 67 53 60 29 100 123 91 101 17 93 97 64 65 79 99 11 108 54 31 115 103 73 43 75 40 82 25 78	89 96 62 74 45 57 65 102 103 58 38 10 29 23 47 73 28 80 124 69 91 17 24 33 121 56 32 68 44 19 6 40 71 77 13 14 125 1 18 82 75 119 90 70 79 4 107 83 115 64 109 97 53 93 78 112 26 86 111 15 20 55 63 49 87 104 25 95 7 120 8 76 94 118 81 11 117 108 5 3 2 106 66 51 27 42 48 30 9 36 39 116 92 21 110 98 101 35 12 88 43 60 31 99 16 54 37 113 41 52 123 50 22 67 72 100 59 105 122 85 34 114 61 46 84	 <p>Objective Function Value over Iterations</p> <p>Initial value: 4750.0 Final value: 5339.0</p>	 <p>Probability Values over Iterations</p>	8.03	6326	3119

Rata-rata	Selisih avg = 84	9.32	6326	3152
-----------	------------------	------	------	------

Tabel 3.5.5.3 Hasil Pengujian Simulated Annealing Linear Schedule (T=10000 dan cooling rate = 0.05)

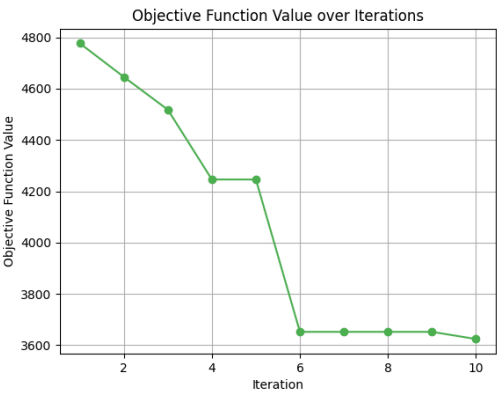
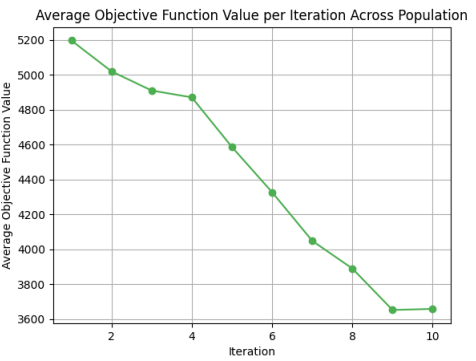
State Awal	State Akhir	Plot Objektif	Plot Probability	Durasi	Banyak Iterasi	Stuck Freq
13 92 34 23 24 2 18 74 3 43 111 61 89 39 123 120 17 103 9 117 40 70 19 56 50 99 26 65 20 25 58 114 52 27 72 69 87 90 77 37 62 14 95 35 53 108 1 75 121 4 10 33 86 55 67 106 112 6 49 122 60 96 63 76 66 97 41 109 110 30 80 94 101 125 119 38 15 44 36 22 102 46 105 31 98 42 47 107 48 57 32 85 84 8 11 118 12 100 21 83 124 82 54 116 115 64 104 88 28 45	13 92 34 23 24 2 18 74 3 43 111 61 89 39 123 120 17 103 9 117 40 70 19 56 50 99 26 65 20 25 58 114 52 27 72 69 87 90 77 37 62 14 95 35 53 108 1 75 121 4 10 33 86 55 67 106 112 6 49 122 60 96 63 76 66 97 41 109 110 30 80 94 101 125 119 38 15 44 36 22 102 46 105 31 98 42 47 107 48 57 32 85 84 8 11 118 12 100 21 83 124 82 54 116 115 64 104 88 28 45	 <p>Objective Function Value over Iterations</p> <p>Initial value: 5281.0 Final value: 4360.0</p>	 <p>Probability Values over Iterations</p>	0.801	633	335

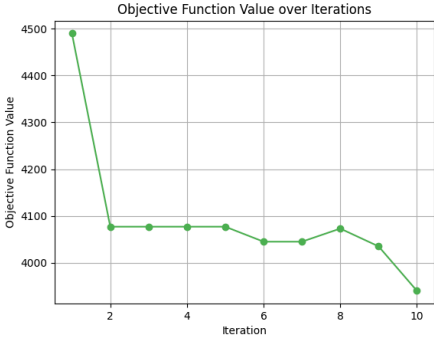
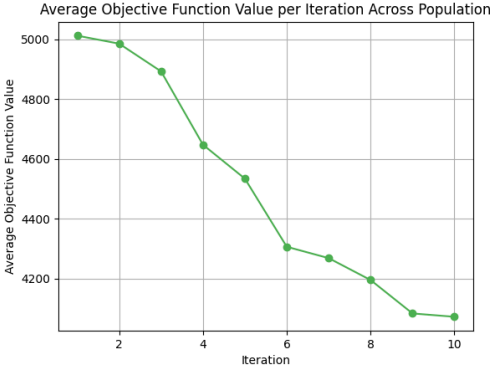
71 16 113 68 79 78 93 73 81 5 29 59 7 51 91	71 16 113 68 79 78 93 73 81 5 29 59 7 51 91					
85 56 10 83 11 76 90 74 104 42 119 106 35 14 92 17 45 53 55 29 50 80 96 2 62 54 39 95 16 89 82 78 37 99 71 34 22 33 122 27 109 30 97 93 49 73 20 75 41 113 105 15 7 100 117 103 59 48 65 3 47 98 63 61 44 5 108 19 88 81 1 86 67 94 79 70 38 6 64 24 40 23 18 12 36 60 118 25 13 112 9 31 107 111 77 69 8 123 4 116 121 114 46 68 28 32 124 66 43 110 87 91 58 57 84 101 52 72 115 120 102 51 125 21 26	38 85 32 93 19 27 106 121 15 125 5 1 123 73 99 41 69 84 94 7 89 57 115 47 29 81 14 107 86 13 66 35 70 88 102 120 108 45 16 25 116 54 6 22 83 104 98 30 11 8 60 59 2 23 51 33 50 77 72 10 42 78 63 43 48 91 119 17 117 40 101 118 112 18 55 9 92 34 75 80 76 96 113 65 52 28 37 105 44 111 110 109 39 24 103 95 21 87 58 31 100 26 49 97 61 20 79 36 68 122 74 56 64 82 90 53 4 114 12 71 62 3 46 124 67	 <p>Objective Function Value over Iterations</p> <p>Initial value: 5352.0 Final value: 4459.0</p>	 <p>Probability Values over Iterations</p>	1.84	633	323

121 98 28 23 97 112 102 79 125 75 114 39 92 37 88 49 47 3 9 105 100 122 123 86 89 91 13 66 73 4 59 38 109 14 43 90 32 1 53 67 99 83 72 87 104 69 36 12 21 62 45 107 120 44 20 8 6 93 113 101 58 108 63 70 68 11 110 16 25 30 24 27 118 54 64 46 56 76 106 84 65 31 2 74 95 7 42 29 78 116 77 40 103 17 55 19 48 115 52 71 10 96 111 18 80 61 82 117 124 41 119 57 26 35 34 85 94 15 5 50 60 51 81 33 22	86 100 60 29 107 65 115 84 114 27 120 17 42 50 20 32 89 96 57 103 22 19 105 25 117 92 75 18 83 72 31 7 76 71 121 3 94 59 104 38 54 111 55 36 6 11 21 77 44 66 91 43 5 62 46 97 26 10 53 24 37 48 63 33 78 41 28 125 61 30 90 34 102 40 39 99 52 81 4 13 106 35 56 93 124 79 122 51 98 82 118 87 74 58 69 8 14 23 113 116 80 101 88 109 2 49 95 67 15 119 1 70 108 73 123 64 45 110 112 47 68 85 16 12 9	 <p>Objective Function Value over Iterations</p> <p>Initial value: 5352.0 Final value: 4459.0</p>	 <p>Probability Values over Iterations</p>	2.07	633	327
Rata-rata		Selisih avg = 902		2.07	633	328

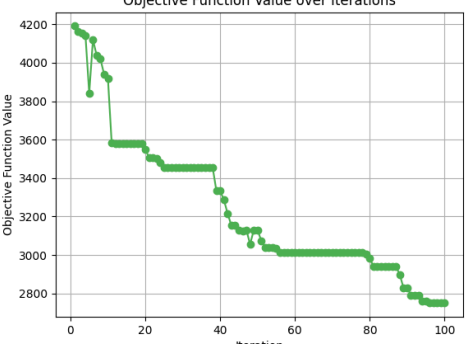
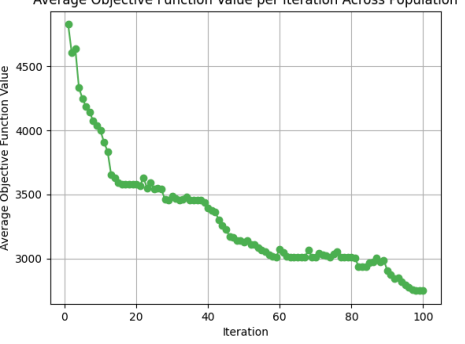
3.5.6. Pengujian Genetic Algorithm

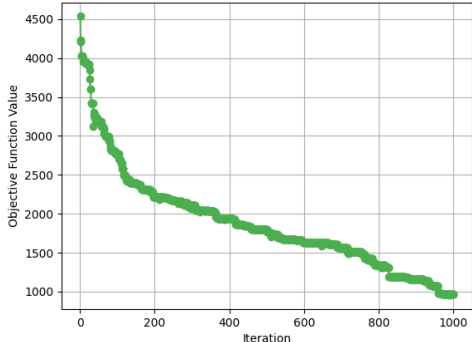
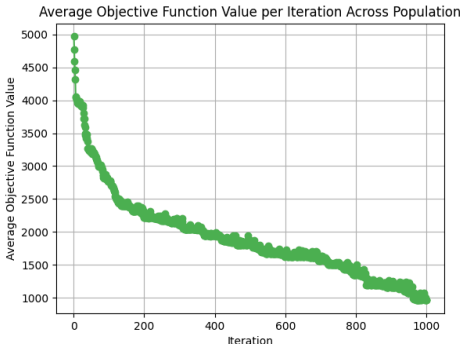
Tabel 3.5.6.1 Hasil Pengujian Genetic Algorithm dengan jumlah populasi (P) konstan (P = 10)

State Awal	State Akhir	Plot Objektif Maksimum	Plot Rata-Rata Objektif	Durasi	Banyak Iterasi (T)
10 individu	95 35 39 106 48 82 55 41 84 16 9 111 69 90 102 81 107 40 30 15 49 72 98 27 122 62 28 36 47 52 53 39 29 22 120 55 100 72 34 55 109 98 78 67 90 125 87 71 31 64 123 86 58 10 115 20 1 93 75 114 113 28 63 108 76 40 84 42 83 85 45 12 15 122 13 3 38 102 60 41 110 111 18 79 81 106 19 44 16 9 91 97 25 61 46 52 100 49 68 81	 <p>Initial Value = 4777 Optimum Value = 3624</p>	 <p>Avg. Initial Value = ±5200 Avg. Final Value = ±3650</p>	0.71	10

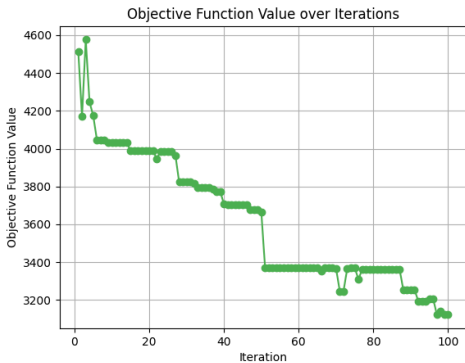
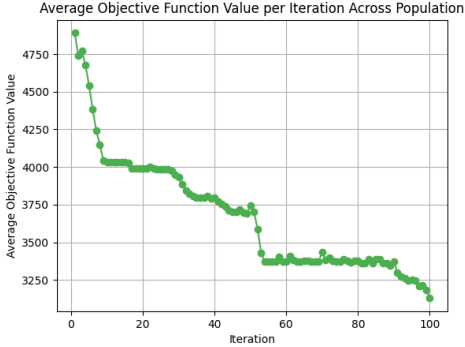
	30 86 25 116 47 80 94 82 56 11 48 32 119 57 69 68 96 46 74 121 59 8 92 70 30				
10 individu	13 92 34 23 24 2 18 74 3 43 111 61 89 39 123 120 17 103 9 117 40 70 19 56 50 99 26 65 20 25 58 114 52 27 72 69 87 90 77 37 62 14 95 35 53 108 1 75 121 4 10 33 86 55 67 106 112 6 49 122 60 96 63 76 66 97 41 109 110 30 80 94 101 125 119 38 15 44 36 22 102 46 105 31 98 42 47 107 48 57 32 85 84 8 11 118 12 100 21 83 124 82 54 116 115 64 104 88 28 45 71 16 113 68 79 78 93 73 81 5 29 59 7 51 91	 <p>Initial Value = 4738 Optimum Value = 4414</p>		1.3	10

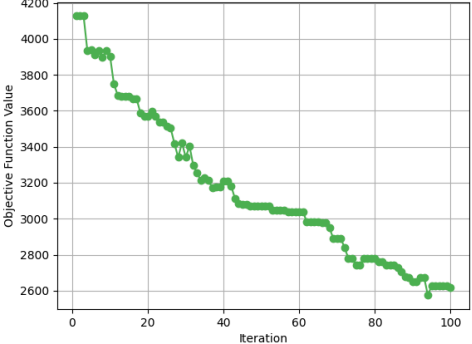
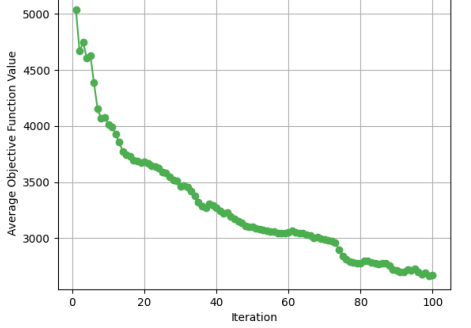
<div>10 individu</div> <div><div><div>94 51 105 50 67</div><div>17 59 33 24 35</div><div>61 90 55 77 22</div><div>111 2 57 52 36</div><div>9 82 73 56 121</div></div><div><div>31 32 6 44 39</div><div>116 69 25 74 103</div><div>85 18 8 19 68</div><div>115 110 60 92 64</div><div>95 58 87 99 5</div></div><div><div>46 15 66 76 4</div><div>122 72 98 102 27</div><div>71 28 63 100 53</div><div>93 89 40 112 79</div><div>81 108 65 42 41</div></div><div><div>45 124 101 97</div><div>104</div><div>48 3 80 114 49</div><div>26 107 21 88 20</div><div>70 123 125 13 62</div><div>75 96 7 78 119</div></div><div><div>14 84 23 117 43</div><div>34 38 109 37 11</div><div>10 30 83 16 12</div><div>29 118 1 47 120</div><div>86 54 106 91 113</div></div></div>	<div><div><div>Objective Function Value over Iterations</div><table><thead><tr><th>Iteration</th><th>Objective Function Value</th></tr></thead><tbody><tr><td>1</td><td>4730</td></tr><tr><td>2</td><td>4650</td></tr><tr><td>3</td><td>4750</td></tr><tr><td>4</td><td>4750</td></tr><tr><td>5</td><td>4750</td></tr><tr><td>6</td><td>4750</td></tr><tr><td>7</td><td>4650</td></tr><tr><td>8</td><td>4550</td></tr><tr><td>9</td><td>4420</td></tr><tr><td>10</td><td>4420</td></tr></tbody></table></div><div><div>Initial Value = 4490</div><div>Optimum Value = 3940</div></div></div>	Iteration	Objective Function Value	1	4730	2	4650	3	4750	4	4750	5	4750	6	4750	7	4650	8	4550	9	4420	10	4420	<div><div><div>Average Objective Function Value per Iteration Across Population</div><table><thead><tr><th>Iteration</th><th>Average Objective Function Value</th></tr></thead><tbody><tr><td>1</td><td>4950</td></tr><tr><td>2</td><td>4920</td></tr><tr><td>3</td><td>4840</td></tr><tr><td>4</td><td>4800</td></tr><tr><td>5</td><td>4760</td></tr><tr><td>6</td><td>4760</td></tr><tr><td>7</td><td>4750</td></tr><tr><td>8</td><td>4680</td></tr><tr><td>9</td><td>4610</td></tr><tr><td>10</td><td>4580</td></tr></tbody></table></div></div>	Iteration	Average Objective Function Value	1	4950	2	4920	3	4840	4	4800	5	4760	6	4760	7	4750	8	4680	9	4610	10	4580	<div>1.28</div>	<div>10</div>
Iteration	Objective Function Value																																															
1	4730																																															
2	4650																																															
3	4750																																															
4	4750																																															
5	4750																																															
6	4750																																															
7	4650																																															
8	4550																																															
9	4420																																															
10	4420																																															
Iteration	Average Objective Function Value																																															
1	4950																																															
2	4920																																															
3	4840																																															
4	4800																																															
5	4760																																															
6	4760																																															
7	4750																																															
8	4680																																															
9	4610																																															
10	4580																																															
<div>Avg iterasi = 10, P =10 average Optimum value = 3992.6</div>			<div>1.09</div>	<div>10</div>																																												

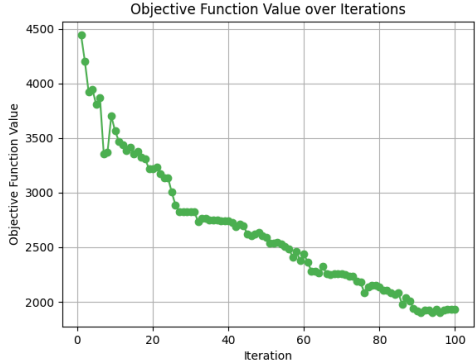
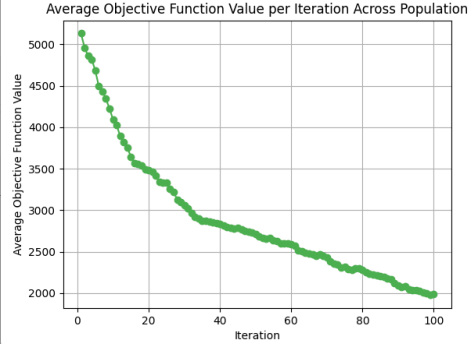
<p>10 individu</p>	<p>27 84 109 50 26 100 62 35 56 53 93 38 125 76 9 116 4 54 79 106 43 120 22 47 57 104 30 13 68 99 15 122 10 114 65 118 2 84 21 112 33 111 29 47 98 58 46 86 81 48 2 105 113 7 61 12 6 74 90 70 77 3 63 102 16 121 73 27 38 29 105 124 49 16 99 78 29 10 41 59 101 85 46 22 88 68 115 32 123 82 57 47 86 60 39 46 8 56 71 52 79 37 3 124 91 111 51 70 41 54 76 42 92 50 11 5 88 85 66 53 47 81 97 35 65</p>	<p>Objective Function Value over Iterations</p>  <p>Initial Value = 4200 Optimum Value = 2750</p>	<p>Average Objective Function Value per Iteration Across Population</p>  <p>Avg. Initial Value = ±4850 Avg. Final Value = ±2800</p>	<p>6.60</p>	<p>100</p>
--------------------	--	---	--	-------------	------------

10 individu	<div>107 29 105 4 77</div> <div>10 122 67 112 24</div> <div>53 52 15 79 89</div> <div>37 46 56 121 36</div> <div>85 53 75 3 87</div> <div>50 104 39 113 2</div> <div>14 46 114 26 114</div> <div>114 22 80 53 36</div> <div>9 88 62 43 95</div> <div>105 30 19 89 70</div> <div>79 36 107 26 70</div> <div>61 5 83 123 37</div> <div>95 50 63 82 64</div> <div>43 101 36 24 90</div> <div>30 105 26 77 77</div> <div>60 75 21 95 118</div> <div>82 93 46 39 46</div> <div>18 110 117 18 49</div> <div>111 64 36 36 90</div> <div>48 36 90 120 14</div> <div>24 60 77 77 53</div> <div>118 57 5 24 105</div> <div>45 80 35 83 90</div> <div>78 15 114 100 6</div> <div>56 105 107 26 62</div>	<div><div>Objective Function Value over Iterations</div></div> <div><div>Initial Value = 4616</div><div>Optimum Value = 952</div></div>	<div><div>Figure 2</div></div> <div><div>Avg. Initial Value = ±5000</div><div>Avg. Final Value = ±950</div></div>	61.45	1000
Rata-rata			22.92	370	

Tabel 3.5.6.1 Hasil Pengujian Genetic Algorithm dengan jumlah iterasi konstan (T = 100)

State Awal	State Akhir	Plot Objektif Maksimum	Plot Rata-Rata Objektif	Durasi	Jumlah Populasi (P)
10 individu	7 81 37 101 89 68 39 91 67 93 37 29 94 26 97 79 106 8 26 98 86 34 59 36 18 33 43 123 94 88 107 115 23 66 1 104 13 50 70 19 45 49 99 55 29 24 113 103 49 37 100 14 74 117 46 7 98 97 101 121 30 123 63 20 47 108 17 18 3 59 114 33 42 55 84 7 68 108 16 100 105 74 33 97 48 19 57 24 96 36 67 82 86 67 18 60 85 1 40 75 112 121 3 44 8 10 30 65 87 38 12 37 90 35 109 11 84 102 104 104 27 40 73 39 56	 <p>1st Gen. Min. Value = 4513 Optimum Value = 3122</p>	 <p>Avg. Initial Value = ±4900 Avg. Final Value = ±3100</p>	5.90	10

<p>10 individu</p>	<p>58 89 86 17 93 18 87 69 9 113 83 99 76 64 28 59 62 3 121 23 104 27 82 103 53</p> <p>39 13 108 125 10 38 30 89 47 91 85 96 15 19 111 109 101 95 47 64 56 1 31 73 119</p> <p>98 100 22 4 106 79 40 58 104 37 66 74 82 54 60 29 52 70 122 43 2 26 89 16 17</p> <p>108 30 27 84 95 50 43 64 2 41 57 34 66 78 86 9 95 89 71 44 70 115 34 84 40</p> <p>45 96 69 51 29 115 42 40 59 70 56 17 104 94 80 40 109 57 35 76 36 74 13 86 75</p>	<p>Objective Function Value over Iterations</p>  <p>1st Gen. Min. Value = 4150 Optimum Value = 2574</p>	<p>Average Objective Function Value per Iteration Across Population</p>  <p>Avg. 1st Gen Value = ±5100 Avg. Final Value = ±2600</p>	<p>43.44</p>	<p>20</p>
--------------------	---	---	--	--------------	-----------

10 individu	 <p>1st Gen. Min. Value = 4450 Optimum Value = 1899</p>	 <p>Avg. 1st Gen Value = ± 5100 Avg. Final Value = ± 2000</p>	605.20	50
Rata-rata			218.18	26.67

BAB IV ANALISIS

4.1 Perbandingan Hasil Tiap Algoritma

Tabel 4.1 Perbandingan Hasil Terbaik Tiap Algoritma

Algoritma	Iterasi	Nilai Objektif Terbaik	Waktu (second)
Steepest Ascent Hill-Climbing	9	3420	8.39
Sideways Move Hill-Climbing	7	3250	9.28
Stochastic Hill-Climbing	5000	3339	32.83
Restart Hill-Climbing	100 Restart	2941	559.59
Simulated Annealing	64	4050	0.122
Genetic Algorithm	1000	952	61.45

4.1.1 Algoritma Hill Climbing

Pada algoritma Hill Climbing, algoritma yang menghasilkan nilai objektif terbaik yang mendekati nilai objektif nol adalah algoritma *Random-Restart Hill Climbing*, diikuti dengan *sideways*, *steepest ascent*, dan *stochastic*. Algoritma Hill Climbing menghasilkan nilai objektif sekitar 3000.

Pada Algoritma Steepest Ascent dilakukan pembangkitan semua state tetangga dan dipilih tetangga terbaik. Selanjutnya akan berpindah ke lokasi tetangga tersebut jika nilainya lebih baik. Hal ini membuat *state* bisa mendekati hasil akhir objektif yang lebih kecil dengan mudah. Selain itu, algoritma ini termasuk algoritma yang *simple* sehingga waktu yang dibutuhkan untuk melakukan pencarian dengan algoritma ini relatif cepat dan hasil yang

dihasilkan cukup konsisten bisa dilihat pada tabel 3.5.5.2 bahwa range hasil objektif yang diberikan relatif mirip dan konsisten. Namun, algoritma ini rentan *stuck* di lokal optima karena hanya berpindah ke *state* yang memiliki nilai objektif yang lebih baik saja tanpa mempertimbangkan *state* yang memiliki nilai lebih buruk, padahal belum tentu *state* yang lebih buruk tersebut merupakan rute yang buruk.

Algoritma *Sideways move* menghasilkan nilai yang sedikit lebih baik daripada *steepest ascent* karena algoritma ini juga berpindah ke *state* yang memiliki *value* yang sama dengan kondisi *state* saat ini. Waktu yang dibutuhkan untuk menjalankan algoritma ini juga relatif cepat karena algoritma ini relatif *simple*. Hasil dari percobaan yang dilakukan juga cenderung konsisten yaitu sekitar 3000. Namun, algoritma ini juga rentan terjebak di lokal optima karena hanya mempertimbangkan nilai yang lebih baik atau sama dengan kondisi *state* saat ini.

Random Restart Algorithm menghasilkan nilai terbaik dari semua algoritma *hill climbing* karena mencoba men-*generate* ulang tiap *state* saat bertemu dengan kondisi *state* yang lebih buruk. Hal ini membuat algoritma ini memiliki peluang untuk bertemu dengan kondisi *initial state* yang lebih baik dan mendekatkan nilai objektifnya mendekati nilai dari global optima yang dituju. Untuk perhitungan sama seperti *Steepest Ascent* dimana dilakukan pembangkitan setiap *state* tetangga dan dipilih *state* yang paling baik. Jika ditemukan kondisi *state* yang paling buruk maka akan dilakukan *restart* algoritma dengan *initial state* baru. Hal ini dilakukan hingga batas maksimal *restart* yang dilakukan. Waktu yang dibutuhkan untuk algoritma ini cukup lama karena sama saja dengan melakukan algoritma *steepest ascent* secara berulang kali hingga batas maksimal *restart* yang dilakukan. Namun, hasil objektif yang didapat lebih optimal dan lebih dekat dengan nilai objektif pada global optima. Dari hasil percobaan juga diketahui hasil yang dihasilkan algoritma ini cukup optimal dan banyak restart yang ada berpengaruh secara linear pada nilai objektif yang ada. Semakin banyak jumlah restart maka nilai objektif maksimum yang didapat juga semakin kecil.

Pada *Stochastic Algorithm*, algoritma akan berjalan berdasarkan masukan jumlah iterasi dan pembangkitan tetangga pada algoritma ini berbeda dengan algoritma *hill climbing* lainnya. Algoritma ini melakukan pembangkitan *state* secara acak dan diberikan fungsi *heuristik* untuk mengurangi aspek acak dalam melakukan pembangkitan. Setiap bertemu dengan *state* yang lebih baik maka *state* akan berpindah pada *state* tersebut, sedangkan jika tidak maka akan tetap pada *state* saat ini dan jumlah *nmax* iterasi akan berkurang. Dari segi waktu yang dibutuhkan

cenderung cepat dan berbanding lurus dengan iterasi yang dimasukkan. Hasil yang diberikan cenderung konstan di sekitar 3000 karena adanya perbandingan hanya dengan satu *state* yang random. Berbeda dengan algoritma Hill Climbing lain yang membangkitkan tiap tetangga sehingga dapat memilih untuk berpindah ke *state* yang lebih baik secara lebih jelas. Kondisi nilai objektif yang dihasilkan cenderung konsisten dan sama berapapun iterasinya tergantung dengan kondisi *state random* yang didapatkan.

Dari Algoritma Hill Climbing didapat Random Restart memiliki nilai objektif yang paling mendekati dengan global optima, meski begitu waktu yang dibutuhkan sangat lama karena harus melakukan iterasi berkali-kali pada setiap pengulangannya. Banyak *restart* yang dilakukan sebanding dengan nilai objektif yang didapat. Semakin banyak *restart*, semakin dekat nilai objektifnya dengan global optima.

4.1.2 Algoritma Simulated Annealing Algorithm

Pada simulated annealing, penerapan dilakukan dengan membangkitkan successor secara acak, yang kemudian dibandingkan nilai objektifnya dengan current state. Salah satu keunggulan utama algoritma ini adalah kemampuannya untuk keluar dari local optima karena adanya probabilitas dengan menggunakan perhitungan suhu, *cooling rate* dan bilangan euler. Jika successor memiliki nilai objektif lebih buruk, algoritma akan menggunakan probabilitas tertentu untuk tetap berpindah ke successor state. Proses ini memungkinkan eksplorasi yang lebih luas dalam ruang solusi dan meningkatkan kemungkinan menemukan solusi optimal secara global.

Pada Algoritma ini dilakukan perhitungan dengan beberapa *cooling schedule*, yaitu Linear, Eksponensial, *Logarithmic*, dan Kuadratik. Pada pengujian ini, kami memilih linear scheduling daripada kuadratik karena adanya keseimbangan antara eksplorasi dan konvergensi tanpa memperlambat eksekusi. Linear cooling memungkinkan proses untuk tetap mempertahankan variasi suhu yang menurunkan probabilitas menerima solusi yang lebih buruk secara bertahap, namun tidak terlalu lambat.

Dari hasil percobaan hasil objektifnya cenderung bernilai 4000 dimana hal ini lebih buruk dari algoritma *Hill Climbing* karena pada algoritma ini melakukan pembangkitan tetangga secara acak dan mempertimbangkan untuk berpindah *state* yang lebih buruk di mana terkadang hal ini justru membawa kondisi *state* jauh menjadi lebih buruk dari sebelumnya. Hal ini dibuktikan pada pengujian tabel 3.5.5.1 percobaan kedua, didapatkan hasil akhir lebih buruk daripada *initial*

state. Selain itu, nilai *initial Temperature* juga berpengaruh terhadap hasil objektif dari *state* karena semakin besar temperatur, kemungkinan *state* untuk berpindah ke kondisi yang lebih buruk menjadi lebih besar. Berdasarkan hasil analisis, semakin tingginya temperatur yang digunakan akan terjadi banyak pergerakan pula, meningkatkan kemungkinan *random* ini. Akibatnya algoritma dapat mengeksplor solusi lebih luas, tetapi juga meningkatkan frekuensi *stuck* di lokal optima. Perlu dicantumkan pula bahwa temperatur yang sangat tinggi dapat mengakibatkan nilai probabilitas mendekati 1 sehingga grafik yang didapatkan akan selalu berada pada angka 1 dan justru malah mendorongnya mengambil lebih banyak *worse state*.

Namun, algoritma ini memiliki keunggulan dibandingkan dengan algoritma lainnya yaitu waktu yang sangat cepat, salah satunya solusi terbaik dapat dicapai hanya dengan waktu 0.112 detik. Hal ini dikarenakan algoritma ini cukup *simple* dan tidak perlu dibangkitkan dan dilakukan pengecekan untuk setiap *state* tetangga seperti *Hill Climbing*, algoritma ini hanya melakukan perbandingan dengan *state random* saja.

4.1.3 Algoritma Genetic Algorithm

Genetic algorithm melakukan penyelesaian secara paralel sesuai dengan jumlah populasi yang ditentukan serta melakukan pemilihan *parent* secara acak dan menghasilkan individu baru melalui adanya *crossover* dan *mutation*. Kedua hal tersebut umumnya bertujuan untuk memperbaiki nilai *fitness* serta *objective function* dari individu populasi. Namun, kedua hal tersebut juga dapat memperburuk nilai *fitness*, karena penggabungan yang acak dapat menyebabkan angka dalam kubus semakin acak dan jauh dari tujuan yang ingin dicapai. Selain itu, waktu pencapaian solusi sangat bergantung pada kompleksitas *initial state* dan ukuran populasi.

Genetic Algorithm menunjukkan hasil yang mendekati global optima, dengan nilai objektif 952, lebih baik daripada algoritma *local search* lainnya. Hal ini disebabkan oleh penggunaan *crossover* dan *mutation* yang memperluas eksplorasi dalam ruang solusi. Seleksi *parent* dan mekanisme acak dari *crossover* serta *mutation* memberi peluang yang lebih tinggi bagi algoritma untuk mencapai solusi optimal global, meski terkadang ada risiko individu mengalami pengurangan nilai *fitness* karena penggabungan secara acak. Dibandingkan dengan algoritma *local search*, seperti *Hill Climbing* dan *Simulated Annealing*, GA memiliki pendekatan paralel dan kemampuan eksplorasi lebih luas.

Genetic Algorithm lebih cepat dibandingkan dengan Algoritma Hill Climbing, terutama pada *random restart* karena algoritma ini melakukan pencarian secara paralel dari banyak individu dalam populasi. Waktu komputasi GA juga dipengaruhi oleh ukuran populasi dan jumlah iterasi. Dari hasil percobaan, algoritma ini cenderung menghasilkan hasil yang cukup konsisten, terutama jika ukuran populasi dan jumlah iterasi dioptimalkan.

Semakin banyak iterasi, semakin besar peluang GA untuk menghasilkan solusi yang lebih baik karena memberi lebih banyak waktu untuk seleksi, crossover, dan mutation. Dengan jumlah iterasi yang lebih tinggi, Algoritma ini dapat mengeksplorasi lebih banyak kombinasi, sehingga meningkatkan peluang mendekati global optima. Ukuran populasi juga berpengaruh namun tidak seberpengaruh iterasi karena semakin banyak populasi maka semakin berat untuk melakukan penyelesaian algoritma dan juga dapat menambah waktu yang dibutuhkan juga. Namun, populasi yang lebih besar dapat menyediakan lebih banyak peluang individu untuk *crossover* dan *mutation*

4.2 Algoritma Terbaik

Dari hasil percobaan, dapat disimpulkan bahwa *Genetic Algorithm* memiliki algoritma yang paling **mangkus dan sangkil** karena hasil yang didapat paling mendekati dengan nilai objektif global optima dan waktu yang diperlukan relatif cepat untuk menemukan solusi karena melakukan pencarian secara paralel dan variasi populasi untuk menghindari jebakan local optima dan variasi populasi untuk menghindari jebakan local optima.

BAB V

KESIMPULAN DAN SARAN

5.1. Kesimpulan

Dari hasil analisis, dapat disimpulkan bahwa setiap algoritma memiliki kelebihan dan kekurangan masing-masing dalam pencarian solusi optimal. Algoritma Hill Climbing, terutama Random Restart Hill Climbing, mendekati global optima dengan cukup baik namun membutuhkan waktu lebih lama dibandingkan varian lain seperti Steepest Ascent dan Sideways Move yang lebih cepat namun rentan terhadap local optima. Simulated Annealing memperlihatkan keunggulan dalam menghindari jebakan local optima dengan probabilitas perpindahan pada kondisi state yang lebih buruk. Di sisi lain, *Genetic Algorithm* menunjukkan kinerja yang unggul dalam mencapai nilai objektif yang paling mendekati global optima dengan waktu yang relatif singkat, dengan memanfaatkan pencarian paralel dan variasi populasi untuk menghindari jebakan local optima dan variasi populasi untuk menghindari jebakan lokal optima.

5.2 Saran

Selama pengerjaan tugas besar ini kami memiliki beberapa saran untuk pengerjaan tugas besar berikutnya, di antaranya:

1. Untuk pengembangan berikutnya, bisa dilakukan variasi terhadap *heuristic function* untuk optimalisasi.
2. Kode program yang digunakan bisa dibuat lebih optimal dengan memanfaatkan paralelisasi pada setiap algoritma dalam melakukan pencarian.

LAMPIRAN

Repository

https://github.com/slntklr01/IF3170_Tubes1_DiagMagicCube

DAFTAR PUSTAKA

- Algorithm Afternoon. Chapter 2: Temperature and Cooling Schedules. Diakses pada tanggal 2 November 2024, dari [Chapter 2 - Temperature and Cooling Schedules | Algorithm Afternoon](#).
- Dawood, O. (January, 2016). Generalized Method for Constructing MagicCube by Folded Magic Squares. Diakses pada tanggal 24 September 2024 dari https://www.researchgate.net/publication/292177827_Generalized_Method_for_Constructing_Magic_Cube_by_Folded_Magic_Squares. DOI: 10.5815/ijisa.2016.01.01
- JavaTpoint. Hill Climbing Algorithm in Artificial Intelligence. Diakses pada tanggal 24 September 2024 dari <https://www.javatpoint.com>.
- Rooy, N. Effective Simulated Annealing with Python. Diakses pada tanggal 3 November 2024, dari [Effective Simulated Annealing with Python](#)
- Trump, W. (2005). Perfect Magic Cubes. Diakses pada tanggal 24 September 2024 dari <https://www.trump.de/magic-squares/magic-cubes/cubes-1.html>
- Weisstein, Eric W. "Magic Cube." From MathWorld--A Wolfram Web Resource. <https://mathworld.wolfram.com/MagicCube.html>
- Xie, T., & Kang, L. (2003). An evolutionary algorithm for magic squares. The 2003 Congress on Evolutionary Computation, 2003. CEC '03., 2, 906-913 Vol.2.