

PENCARIAN SOLUSI DIAGONAL MAGIC CUBE DENGAN LOCAL SEARCH

Tugas Besar 1

Dikumpulkan sebagai salah satu tugas mata kuliah IF3170 Inteligensi Artifisial

Oleh:

Muhammad Fadli Alfarizi	13121140
Roby Pratama Sitepu	13121142
Muhammad Arviano Yuono	13621034
Hafizh Renanto Akhmad	13621060



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

2024

DAFTAR ISI

DAFTAR ISI	i
BAB 1 DESKRIPSI persoalan.....	1
BAB 2 PEMBAHASAN	4
2.1 Pemilihan Objective <i>Function</i>	4
2.2 Penjelasan Implementasi Algoritma Local Search.....	4
2.2.1 Fungsi dan Kelas Pembantu.....	4
2.2.2 Steepest Ascent Hill-Climbing.....	6
2.2.3 Hill-Climbing with Sideways Move	7
2.2.4 Stochastic Hill-Climbing.....	9
2.2.5 Random Restart Hill-Climbing	9
2.2.6 Simulated Annealing	11
2.2.7 Genetic Algorithm.....	12
2.3 Hasil Eksperimen	17
2.3.1 Steepest Ascent Hill-Climbing.....	18
2.3.2 Hill-Climbing with Sideways Move	22
2.3.3 Stochastic Hill-Climbing.....	25
2.3.4 Random Restart Hill-Climbing	28
2.3.5 Simulated Annealing	31
2.3.6 Genetic Algorithm.....	36
2.3.7 Analisis.....	57
BAB 3 KESIMPULAN DAN SARAN	60
PEMBAGIAN TUGAS TIAP ANGGOTA KELOMPOK.....	61
REFERENSI	62

BAB 1

DESKRIPSI PERSOALAN

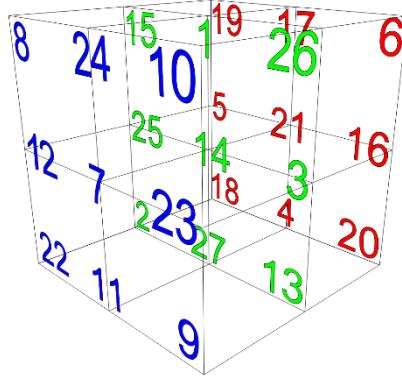
Pada laporan, akan dilakukan implementasi algoritma-algoritma *local search* untuk mencari solusi permasalahan *Diagonal Magic Cube*.

Diagonal Magic Cube atau ‘*Perfect*’ *Magic Cube* merupakan kubus yang tersusun dari angka 1 hingga n^3 tanpa pengulangan dengan n adalah panjang sisi pada kubus tersebut. Sebuah *magic cube* memiliki property *magic number*, yang bernilai [1] [2]:

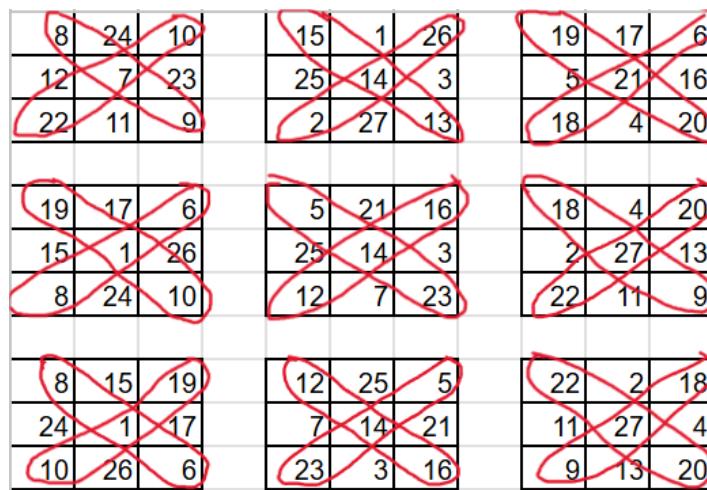
$$M_3(n) = \frac{n(n^3 + 1)}{2}$$

Angka-angka pada *magic cube* tersusun sedemikian rupa sehingga properti-properti berikut terpenuhi:

- 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*. Contoh:
 - Misalkan kita memiliki kubus dengan panjang sisi 3 seperti pada **Error! Reference source not found.**, maka diagonal-diagonal potongan bidang dari kubus tersebut adalah sebagai ditunjukkan oleh Gambar 1.2.



Gambar 1.1. Contoh Basic Magic Cube



Gambar 1.2: Diagonal Potongan Bidang dari Cube pada Gambar 1

Komponen baris, kolom, tiang, diagonal ruang, serta diagonal potongan bidang disebut dengan garis lurus. Untuk sebuah *Diagonal Magic Cube* dengan panjang sisi m , total banyak garis lurus adalah sebanyak $3m^2 + 6m + 4$; m^2 baris, m^2 kolom, m^2 tiang, $6m$ diagonal ruang, dan 4 diagonal potongan bidang [2].

Pada laporan ini, akan diselesaikan permasalahan *Diagonal Magic Cube* berukuran 5x5x5, dengan ketentuan sebagai berikut:

- *Initial state* dari suatu kubus adalah susunan angka 1 hingga 5^3 secara acak.
- Tiap iterasi pada algoritma *local search*, langkah yang boleh dilakukan adalah menukar posisi dari 2 angka (tidak harus bersebelahan) pada kubus tersebut

- *Khusus untuk genetic algorithm*, boleh dilakukan penukaran posisi lebih dari 2 angka sekaligus dalam satu iterasi.

Tujuan dari laporan ini adalah untuk membandingkan performa berbagai algoritma *local search* dalam menyelesaikan masalah *diagonal magic cube*.

BAB 2

PEMBAHASAN

2.1 Pemilihan Objective Function

Objective function adalah fungsi yang mengukur kualitas atau kinerja solusi dalam sebuah masalah optimisasi, dengan memberikan nilai berdasarkan seberapa baik solusi tersebut memenuhi tujuan yang ditentukan. Dimana pada permasalahan *magic cube*, *objective function* digunakan untuk mengukur seberapa jauh susunan angka pada kubus terhadap kondisi yang diinginkan [3].

Fungsi objektif yang digunakan ialah fungsi yang menghitung total negatif dari *conflict* yang terjadi, atau dimana jumlah elemen dalam suatu garis lurus tidak sama dengan *magic number*. Pencarian dilakukan dengan memaksimalkan fungsi (*maximizing*) tersebut, meminimalisasi *conflict* yang terjadi. Secara matematis, fungsi objektif tersebut dapat dinyatakan dalam:

$$f(state) = -\text{Count}(S \text{ where } \text{Sum}(S)! = M_3(n) \text{ for } S \text{ in } \mathcal{S}(state))$$

dimana $M_3(n)$ adalah nilai *magic number* untuk kubus dengan panjang sisi $\mathcal{S}(state)$ adalah semua garis lurus dari kubus pada *state* tersebut. Persamaan ini menghitung negative dari banyak jumlah baris, kolom, tiang, diagonal bidang, dan diagonal kubus yang jumlah angkanya tidak sama dengan magic number.

2.2 Penjelasan Implementasi Algoritma Local Search

2.2.1 Fungsi dan Kelas Pembantu

Untuk implementasi dalam laporan ini, digunakan beberapa kelas dan fungsi pembantu.

2.2.1.1 Kelas State

Kelas ini merepresentasikan suatu *state* dalam algoritma *local search*. Objek dari kelas ini memiliki atribut *cube* yakni sebuah larik 3D yang memenuhi kriteria sebuah *magic cube* namun belum tentu tersusun sempurna, *magic number* dari *cube*, dimensi dari *cube*, dan nilai objektif dari *state* berdasarkan nilai-nilai pada *cube*.

inisialisasi objek dari kelas ini dapat menggunakan input berupa larik 3D atau ukuran dari larik 3D. Jika input berupa ukuran larik 3D, akan digenerate sebuah larik 3D secara *random* dan dijadikan atribut *cube* dari objek tersebut. Nilai *magic number*, dimensi, dan nilai objektif dari *state* langsung dihitung ketika objek terinisialisasi.

2.2.1.2 Kelas Utility

Kelas ini berisi *static methods* yang digunakan sepanjang penggunaan *local search*, antara lain:

1. `getNeighbor`: menerima input *state* dan sepasang koordinat posisi angka, mengembalikan output berupa *neighbor state* dengan menukar posisi angka pada kubus *current state* pada dua koordinat tersebut.
2. `generateCoordinatePairs`: menerima input berupa dimensi dari kubus, mengembalikan semua pasangan koordinat yang mungkin untuk kubus tersebut.
3. `getRandomSuccessor`: menerima input berupa *state*, mengembalikan satu *neighbor state* secara acak.
4. `getBestSuccessor`: menerima input berupa *state*, mengembalikan *neighbor state* dengan nilai objektif terbaik.

2.2.1.3 Kelas BaseResult

Kelas `BaseResult` adalah kelas yang digunakan untuk mengelola informasi dan menyimpannya, terkait hasil dari proses iteratif yang dilakukanl, seperti histori state, objective value, durasi, jumlah iterasi, dan state terbaik yang didapatkan, dengan fungsi-fungsi yaitu:

1. `add_state`: menerima input berupa objek `State` dan menambahkan state ini ke histori. Fungsi ini juga menyimpan *objective value* dari state tersebut ke dalam histori *objective value* untuk setiap iterasi.
2. `export_history`: Menerima input berupa nama file dan mengeksport histori state yang tersimpan ke dalam file tersebut.
3. `state_history`: Property ini menyimpan daftar semua state yang pernah dicapai selama proses iteratif.

4. `objective_function_history`: Property ini menyimpan nilai objektif yang terkait dengan setiap state dalam histori.
5. `duration`: Property ini menyimpan durasi total waktu yang dihabiskan untuk proses iteratif.
6. `iteration`: Property ini menyimpan jumlah iterasi yang telah dilakukan dalam proses iteratif.
7. `best_state`: Property ini mengembalikan pasangan berupa `State` dan `objective value` terbaik yang dicapai selama proses iteratif. Kriteria "terbaik" didasarkan pada nilai objektif tertinggi dalam histori nilai objektif. Jika histori state kosong, property ini akan mengembalikan error.

2.2.1.4 Kelas BaseAlgorithm

Kelas `BaseAlgorithm` adalah kelas yang digunakan untuk menyediakan metode visualisasi hasil dari proses algoritma dengan hasil yang disimpan dalam objek `BaseResult`. Fungsi-fungsi utama yang digunakan ialah adalah:

1. `Visualisasi`: fungsi ini digunakan untuk menampilkan grafik 2D dari perkembangan objective value selama iterasi. Grafik menunjukkan objective values pada setiap iterasi.
2. `Visualisize3D`: fungsi ini bertujuan untuk menampilkan representasi 3D dari state terakhir (terbaik) dalam `BaseResult`.
3. `Visualize3D-subplots`: fungsi ini menampilkan visualisasi 3D dari dari state awal dan state terbaik.

2.2.2 Steepest Ascent Hill-Climbing

Algoritma dimulai dengan mengambil *initial state* berupa kubus secara *random* dan membangkitkan semua *neighbor* yang ada dengan menukar dua pasang angka yang mungkin. Kemudian, algoritma akan memilih "tetangga" dengan *objective value* yang lebih besar daripada *current state* saat itu. Tahap terminasi dilakukan jika tidak terdapat lagi tetangga yang memiliki *objective value* yang lebih besar. Algoritma ini rentan terhadap solusi *local optimum* ataupun *shoulder*.

Pada implementasi dalam laporan ini, ditambahkan *stopping criteria* berupa terminasi algoritma jika didapatkan *current state* dengan *objective value* tertinggi atau *global maximum*, yakni nol. Dengan kriteria ini, algoritma tidak akan membangkitkan *neighbor* yang ada dari *state* tersebut dan mengurangi komputasi yang perlu dilakukan.

Gambaran lebih jelas dari algoritma ini dapat dituliskan sebagai tahap-tahap berikut:

1. Bangkitkan sebuah *initial state* dan jadikan sebagai *current state*.
2. Cek apakah *current state* sudah memenuhi nilai *global maximum*. Jika ya, hentikan pencarian dan ambil *current state* sebagai solusi.
3. Bangkitkan semua *neighboring state* dengan melakukan semua pertukaran dua angka yang mungkin.
4. Bandingkan nilai objektif dari setiap *neighboring state* dengan *current state*. Jika terdapat *neighboring state* dengan *objective value* lebih besar, maka pilih *neighboring state* tersebut sebagai *successor* dan perbaharui *current state* dengan *successor* dan ulangi tahap 2–4. Jika semua *neighboring state* memiliki *objective value* lebih kecil atau sama dengan *current state*, maka hentikan pencarian dan ambil *current state* sebagai solusi.

Berikut adalah *source code* dari implementasi algoritma *steepest ascent hill-climbing*:

[Steepest ascent hill-climbing](#)

2.2.3 Hill-Climbing with Sideways Move

Algoritma ini memiliki prinsip kerja yang mirip dengan algoritma *steepest ascent hill-climbing*, namun algoritma ini memungkinkan *current state* untuk berpindah pada *neighboring state* dengan nilai *objective function* yang sama. Algoritma hanya melakukan terminasi pada saat nilai *objective function* dari tetangga lebih kecil dari nilai *objective value* dari *current state*. Dengan demikian, algoritma ini dapat melewati *shoulder* dan meningkatkan peluang mencapai solusi *global optimum*.

Pada implementasi dalam laporan ini, ditambahkan *stopping criteria* berupa terminasi algoritma jika didapatkan *current state* dengan *objective value* tertinggi

atau *global maximum*, yakni nol. Dengan kriteria ini, algoritma tidak akan membangkitkan *neighbor* yang ada dari *state* tersebut dan mengurangi komputasi yang perlu dilakukan. Selain itu, ditambahkan pula parameter “*maximum sideway moves*” untuk, salah satu-nya, mencegah gerakan “bulak-balik” antara *state* dengan *objective value* yang sama.

Gambaran lebih jelas dari algoritma ini dapat dituliskan sebagai tahap-tahap berikut:

1. Bangkitkan sebuah *initial state* dan jadikan sebagai *current state*.
2. Cek apakah *current state* sudah memenuhi nilai *global maximum*. Jika ya, hentikan pencarian dan ambil *current state* sebagai solusi.
3. Bangkitkan semua *neighboring state* dengan melakukan semua pertukaran dua angka yang mungkin.
4. Bandingkan nilai objektif dari setiap *neighboring state* dengan *current state*.
 - a) Jika terdapat *neighboring state* dengan *objective value* lebih besar, maka pilih *neighboring state* tersebut sebagai *successor* dan perbaharui *current state* dengan *successor* dan ulangi tahap 2–4 dan *reset* jumlah *sideway move* yang dilakukan.
 - b) Jika tidak ada *neighboring state* dengan *objective value* lebih besar namun terdapat *neighboring state* dengan *objective value* yang sama,
 - i. jika jumlah *sideway move* sudah mencapai nilai maksimum, maka hentikan pencarian dan ambil *current state* sebagai solusi.
 - ii. jika jumlah *sideway move* belum mencapai nilai maksimum, pilih *neighboring state* tersebut sebagai *successor* dan perbaharui *current state* dengan *successor* dan ulangi tahap 2–4.
 - c) Jika semua *neighboring state* memiliki *objective value* lebih kecil atau sama dengan *current state*, maka hentikan pencarian dan ambil *current state* sebagai solusi.

Berikut adalah *source code* dari implementasi algoritma *hill-climbing with sideways-move*: [Hill-climbing with sideways move](#)

2.2.4 Stochastic Hill-Climbing

Algoritma ini memiliki prinsip yang sama dengan algoritma *hill-climbing*, namun memilih *successor* bukan dari *neighbor* yang memiliki *objective value* tertinggi, namun dengan membangkitkan *neighbor* secara acak. Algoritma melakukan terminasi berdasarkan jumlah iterasi maksimum yang ditentukan. Dengan algoritma ini, kemungkinan terjebak pada *local maximum* berkurang, tetapi jumlah iterasi yang diperlukan menjadi sangat banyak, dan tidak terlalu meningkatkan kemungkinan mencapai *global maximum*,

Pada implementasi dalam laporan ini, ditambahkan *stopping criteria* berupa terminasi algoritma jika didapatkan *current state* dengan *objective value* tertinggi atau *global maximum*, yakni nol. Dengan kriteria ini, algoritma tidak akan membangkitkan *neighbor* yang ada dari *state* tersebut dan mengurangi komputasi yang perlu dilakukan.

Gambaran lebih jelas dari algoritma ini dapat dituliskan sebagai tahap-tahap berikut:

1. Bangkitkan sebuah *initial state* dan jadikan sebagai *current state*.
2. Cek apakah *current state* sudah memenuhi nilai *global maximum*. Jika ya, hentikan pencarian dan ambil *current state* sebagai solusi.
3. Bangkitkan sebuah *neighboring state* secara acak dari semua pertukaran dua angka yang mungkin.
4. Bandingkan nilai objektif dari *neighboring state* dengan *current state*. Jika terdapat *neighboring state* dengan *objective value* lebih besar, maka pilih *neighboring state* tersebut sebagai *successor* dan perbaharui *current state*.
5. Ulangi tahap 2–4 sebanyak (iterasi maksimum – 1)

Berikut adalah *source code* dari implementasi algoritma *stochastic hill-climbing*:

[Stochastic hill-climbing](#)

2.2.5 Random Restart Hill-Climbing

Algoritma ini, pada prinsipnya, merupakan *steepest-ascent hill climbing* yang dilakukan berkali-kali hingga mencapai *global maximum*. Misalkan pada

sebuah *steepest-ascent hill climbing* telah mencapai terminasi namun *global maximum* belum dicapai, algoritma ini akan membangkitkan sebuah *successor* secara *random* untuk dijadikan *initial state* yang baru dari algoritma *steepest-ascent hill climbing* tersebut. Algoritma ini menghindari dari *local maksimum* dan meningkatkan kemungkinan mendapatkan *global maximum*.

Pada implementasi dalam laporan ini, ditambahkan *stopping criteria* berupa terminasi algoritma jika didapatkan *current state* dengan *objective value* tertinggi atau *global maximum*, yakni nol. Dengan kriteria ini, algoritma tidak akan membangkitkan *neighbor* yang ada dari *state* tersebut dan mengurangi komputasi yang perlu dilakukan. Selain itu, ditambahkan pula parameter “*maximum restart*” untuk membatasi jumlah *restart* yang dilakukan agar algoritma tidak berjalan terlalu lama karena belum menemukan *global maximum*.

Gambaran lebih jelas dari algoritma ini dapat dituliskan sebagai tahap-tahap berikut:

1. Bangkitkan sebuah *initial state* dan jadikan sebagai *current state*.
2. Cek apakah *current state* sudah memenuhi nilai *global maximum*. Jika ya, hentikan pencarian dan ambil *current state* sebagai solusi.
3. Bangkitkan semua *neighboring state* dengan melakukan semua pertukaran dua angka yang mungkin.
4. Bandingkan nilai objektif dari setiap *neighboring state* dengan *current state*.
 - a) Jika terdapat *neighboring state* dengan *objective value* lebih besar, maka pilih *neighboring state* tersebut sebagai *successor* dan perbaharui *current state* dengan *successor* dan ulangi tahap 2–4.
 - b) Jika semua *neighboring state* memiliki *objective value* lebih kecil atau sama dengan *current state*, jika jumlah *restart* sudah mencapai maksimum, maka hentikan pencarian dan ambil *current state* sebagai solusi. Jika jumlah *restart* belum mencapai maksimum, maka bangkitkan sebuah *initial state* secara *random* dan ulangi tahap 1–4 .

Berikut adalah *source code* dari implementasi algoritma *random restart hill-climbing*: [Random restart hill-climbing](#)

2.2.6 Simulated Annealing

Algoritma *hill climbing* memiliki satu kekurangan utama, yaitu ia tidak pernah berpindah yang ke *state* dengan *objective value* lebih rendah atau “turun” dari kondisi saat ini, sehingga ada kemungkinan ia terjebak dalam *local maximum*. Metode *simulated annealing* merupakan improvisasi dari metode hill climbing, dengan menggabungkannya dengan *random walk* dengan cara tertentu, sehingga terdapat kemungkinan untuk “turun” dari *local maximum* dan dapat menuju *global maximum*.

Sama seperti *stochastic hill-climbing*, *successor* diambil dari *neighbour state* yang dibangkitkan secara acak. Untuk *successor* yang lebih baik, maka *successor* tersebut diterima atau *current state* akan berpindah ke *successor* tersebut. Jika *successor* yang didapat lebih buruk, terdapat fungsi probabilitas yang akan menentukan apakah *successor* tersebut dapat diterima. Dalam hal ini, dikenal istilah “temperatur”. Secara matematis, nilai probabilitas tersebut dapat dihitung dengan $e^{\frac{\Delta E}{T}}$, dimana T merupakan ‘temperatur’ dan ΔE merupakan selisih nilai *objective function* antara *successor* dengan *current state*.

Semakin besar T , maka kemungkinan *successor* lebih buruk terpilih akan semakin besar. Salah satu cara penentuan apakah *successor* dengan nilai lebih buruk dapat diterima adalah dengan membandingkan nilai probabilitas tersebut dengan suatu nilai konstan sebagai *threshold*. Nilai T diatur oleh fungsi yang disebut dengan *scheduling function* dan berkang setiap iterasi bertambah.

Pada implementasi dalam laporan ini, ditambahkan *stopping criteria* berupa terminasi algoritma jika didapatkan *current state* dengan *objective value* tertinggi atau *global maximum*, yakni nol. Dengan kriteria ini, algoritma tidak akan membangkitkan *neighbor* yang ada dari *state* tersebut dan mengurangi komputasi yang perlu dilakukan serta mencegah berpindah ke *state* yang lebih buruk ketika temepratur masih besar. Selain itu, juga diberikan dua jenis *scheduling function*: linear atau eksponensial.

Gambaran lebih jelas dari algoritma ini dapat dituliskan sebagai tahap-tahap berikut:

1. Jadikan *initial state* sebagai *current state*

2. Hitung nilai “temperatur” (T) berdasarkan fungsi *schedule* yang ditentukan dengan input banyak iterasi yang sudah dilalui dan jenis fungsi *schedule*.
3. Jika $T = 0$, hentikan iterasi dan ambil *current state* sebagai solusi.
4. Bangkitkan sebuah *neighboring state* secara acak dari semua pertukaran dua angka yang mungkin.
5. Hitung ΔE atau selisih *objective value* antara *neighboring state* dan *current state*.
 - a) Jika $\Delta E > 0$, jadikan *neighboring state* sebagai *successor* dan perbaharui *current state* dengan *successor* serta ulangi tahap 2–4.
 - b) Jika $\Delta E < 0$, maka dihitung nilai probabilitas, yakni $e^{\frac{\Delta E}{T}}$. Jika nilai probabilitas lebih besar dari *threshold* yang ditentukan, jadikan *neighboring state* sebagai *successor* dan perbaharui *current state* dengan *successor* serta ulangi tahap 1–4. Jika nilai probabilitas lebih kecil dari *threshold* yang ditentukan, ulangi tahap 2–4 tanpa mengganti *current state*.

Berikut adalah *source code* dari implementasi algoritma *random restart hill-climbing*:

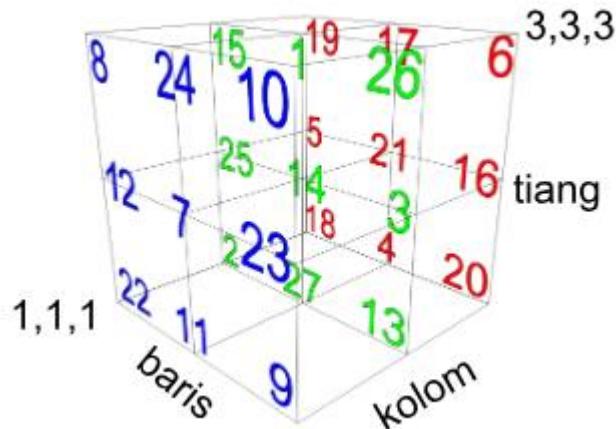
[Simulated Annealing](#)

2.2.7 Genetic Algorithm

Genetic algorithm (GA) merupakan tipe dari *evolutionary algorithm* (EA). Algoritma ini terinspirasi dari seleksi alam di biologi, dimana terdapat kumpulan individu (*state*) dan hanya individu-individu yang paling layak (*fittest, state* dengan *highest value*) dapat memproduksi keturunan (*successor state*) yang akan memproduksi generasi selanjutnya. Proses ini dikenal sebagai rekombinasi.

EA dimulai dengan beberapa state yang dibangkitkan secara random atau populasi. Untuk GA, setiap individu dalam populasi direpresentasikan sebagai string. Pada kasus Diagonal Magic Cube ini, individu direpresentasikan oleh string yang tersusun dari angka pada tiap koordinat diurutkan dari baris ke-1, kolom ke-1, dan tiang ke-1. Misalkan untuk cube yang ditunjukkan oleh Gambar 1.1, maka representasi individu untuk GA adalah sebagai berikut:

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



Gambar 2.1. Cube Pada Gambar 1.1 dengan Koordinatnya

Informasi yang bisa didapatkan dari string tersebut adalah pada koordinat (1, 1, 1) atau baris ke-1, kolom ke-1, dan tiang ke-1 terdapat angka 22; pada koordinat (1, 1, 2) atau baris ke-1, kolom ke-1, dan tiang ke-2 terdapat angka 12; dan seterusnya.

Dalam GA, digunakan beberapa *parent state* atau individu untuk memproduksi satu/beberapa *offspring state*. Sebelum memproduksi *offspring*, setiap individu diseleksi terlebih dahulu dengan ditentukan kelayakannya oleh *fitness function*. Karena sebuah *fitness function* haruslah lebih besar untuk individu yang lebih baik, maka dipilih *fitness function* berikut:

$$\text{fitness(state)} = \text{Count}(S \text{ where } \text{Sum}(S) == M_3(n) \text{ for } S \text{ in } \mathcal{S}(\text{state}))$$

dimana $M_3(n)$ adalah nilai *magic number* untuk kubus dengan panjang sisi $\mathcal{S}(\text{state})$ adalah semua garis lurus dari kubus pada state tersebut. Fungsi ini menghitung banyak garis lurus dengan jumlah elemen-elemennya senilai *magic number* pada sebuah individu. Fungsi *fitness* ini dipilih karena terdapat nilai *global maximum* dapat diperoleh, yakni banyak garis lurus pada *magic cube* tersebut, senilai $3n^2 + 6n + 4$. Jika diperhatikan, fungsi *fitness* ini berhubungan dengan fungsi objektif yang telah dibuat. Karena fungsi objektif merupakan negatif banyak garis lurus dengan jumlah elemen *tidak senilai* *magic number*, fungsi *fitness* ini menghitung komplemen nilai positifnya. Dengan demikian, bisa didapatkan hubungan:

$$\text{fitness(state)} = (3n^2 + 6n + 4) + f(\text{state})$$

Nilai *fitness* yang didapat akan dipakai untuk menghitung probabilitas dipilihnya individu tersebut dalam proses seleksi. Salah satu cara menghitung probabilitasnya adalah dengan cara berikut:

$$p(\text{state}) = \frac{\text{fitness}(\text{state})}{\text{sum}(\text{fitness}(\text{state}))}$$

Setelah didapatkan probabilitas, salah satu cara melakukan seleksi adalah dengan mengurutkan tiap individu berdasarkan *fitness value*, mengubah probabilitas menjadi nilai kumulatif, dan didapatkan *range* tiap individu dari 0 hingga 100%. Kemudian, diambil nilai secara *random* antara 0 hingga 100%, dan dipilih individu sesuai dengan posisi nilai *random* tersebut pada *range* probabilitas kumulatif.

Setelah didapatkan mendapatkan pasangan individu yang telah diseleksi oleh *fitness function*, dilakukan proses *crossover*. Ada beberapa metode *crossover* yang dapat dilakukan. Untuk masalah *Diagonal Magic Cube* dalam bentuk representasi yang telah dijelaskan, setiap angka hanya boleh muncul sekali. Dengan kata lain, yang dicari adalah permutasi terbaik dari elemen-elemen dalam *magic cube* tersebut. Salah satu metode *crossover* yang dapat dilakukan adalah *partially-mapped crossover* (PMX). PMX ditemukan oleh Goldberg dan Lingle (1985) untuk masalah *travelling salesman* [3]. Misalkan terdapat dua *parent state* sebagai berikut:

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

Berikut adalah tahapan yang dilakukan operator PMX:

1. Secara *random* memilih dua lokasi pemotongan pada *string*. Substring antara dua lokasi pemotongan disebut dengan *mapping sections*. Misalkan untuk contoh di atas, dipilih lokasi pemotongan pertama antara elemen string ke-3 dan ke-4 dan lokasi pemotongan kedua antara elemen string ke-6 dan ke-7.

Parent 1:

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

Parent 2:

```
(22  2 18|11 27  4| 9 13 20 12 25  5 08 14 21 23  3 16  8 15 19 24  1 17 10 26  6)
```

2. Dengan demikian, didapatkan *mapping* $9 \leftrightarrow 11$, $13 \leftrightarrow 27$, dan $23 \leftrightarrow 4$. Setelah didapatkan *mapping*, dibuat string *offspring* dengan menyalin *mapping section parent* pertama ke *offspring* kedua dan sebaliknya.

Offspring 1:

Offspring 2:

3. Selanjutnya, elemen *offspring* ke-*i* diisi oleh elemen-elemen *parent* ke-*i*. Jika pada *offspring* tersebut sudah ada elemen yang akan mengisi, elemen tersebut diganti sesuai dengan *mapping*. Sebagai contoh, pada elemen ke-7 di *offspring* 2, seharusnya ia diisi oleh elemen ke-7 dari *parent* 1, yakni ‘9’. Namun, ‘9’ sudah ada di *offspring* tersebut. Dengan *mapping* $9 \leftrightarrow 11$, maka elemen ke-7 di *offspring* 2 diisi oleh ‘11’. Hasil dari operasi ini adalah sebagai berikut:

Offspring 1:

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

Offspring 2:

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

Setelah *offspring* didapatkan, setiap lokasi pada *string offspring* dapat dikenai *random mutation*. Umumnya, untuk GA, mutasi terjadi pada satu lokasi, dan nilai pada berubah secara acak dengan probabilitas yang rendah. Untuk representasi *Diagonal Magic Cube* yang dipilih, hal ini tidak bisa diterapkan, karena setiap angka harus ada dalam satu individu dan tidak boleh ada angka yang muncul lebih dari satu kali. Salah satu jenis mutasi yang dapat diterapkan adalah *swap mutation*. Pada metode ini, dipilih dua elemen secara *random* untuk ditukar [4]. Perlu dicatat bahwa mutasi terjadi dengan probabilitas tertentu yang kecil atau tidak selalu terjadi.

Pada implementasi dalam laporan ini, banyak *offspring* yang dihasilkan sama dengan banyak *parents* yang digunakan. PMX digunakan untuk mutasi *crossover* dan *swap mutation* digunakan untuk proses mutasi. Selain itu, pengecekan individu paling *fit* atau *fittest individual* dilakukan sebelum proses *crossover* untuk mengurangi komputasi yang dilakukan serta mencegah pembangkitan populasi dengan individu-individu yang lebih buruk. Kemudian, ditentukan pula maksimum iterasi sebagai input.

Gambaran lebih jelas dari algoritma ini dapat dituliskan sebagai tahap-tahap berikut:

1. *n-initial state* dibangkitkan secara acak dan disimpan sebagai individu-individu di *initial population* dan jadikan *initial population* sebagai *current population*. *n* harus bernilai genap.
2. Untuk setiap individu dalam *current population*, dihitung nilai *fitness*-nya dengan *fitness function* yang telah dijelaskan sebelumnya.
3. Setelah nilai *fitness* dihitung, dicek apakah terdapat individu yang memiliki nilai *fitness value optimum*. Jika demikian, algoritma diterminasi dan digunakan individu tersebut sebagai solusi. Jika tidak, dilakukan pemilihan *parent*.
4. Untuk memilih *parent*, dicari dahulu probabilitas kemunculan *individu* tersebut sebagai *parent* dengan menggunakan formula yang telah diterangkan sebelumnya. Kemudian, probabilitas ini akan diurutkan sehingga terbentuk *range* tiap probabilitas dalam rentang 0-100%. Selanjutnya, dipilih angka secara random antara 0-100% sebanyak *n* kali sehingga diperoleh $n/2$ pasang *parents*.
5. Untuk tiap pasang *parents*, dilakukan PMX untuk tiap pasang *parents*. Sebelum melakukan PMX, setiap *state parent* ditulis ulang dalam bentuk string representasi. Untuk tiap pasang, dipilih secara acak dua lokasi elemen *string* sebagai batas luar *mapping section*. Setelah itu, dilakukan operasi pengisian berdasarkan operasi PMX yang sudah dijelaskan sebelumnya. Setiap pasang *parents* menghasilkan 2 (dua) *offsprings*.
6. Untuk tiap *offspring*, ditentukan secara acak apakah mutasi akan terjadi. Jika mutasi terjadi, dipilih dua lokasi elemen *string* secara acak untuk

ditukar. Setelah terpilih, *swap mutation* dilakukan dengan menukar kedua elemen string tersebut. Kemudian, *offsprings* ditulis ulang dalam bentuk kubus 3D.

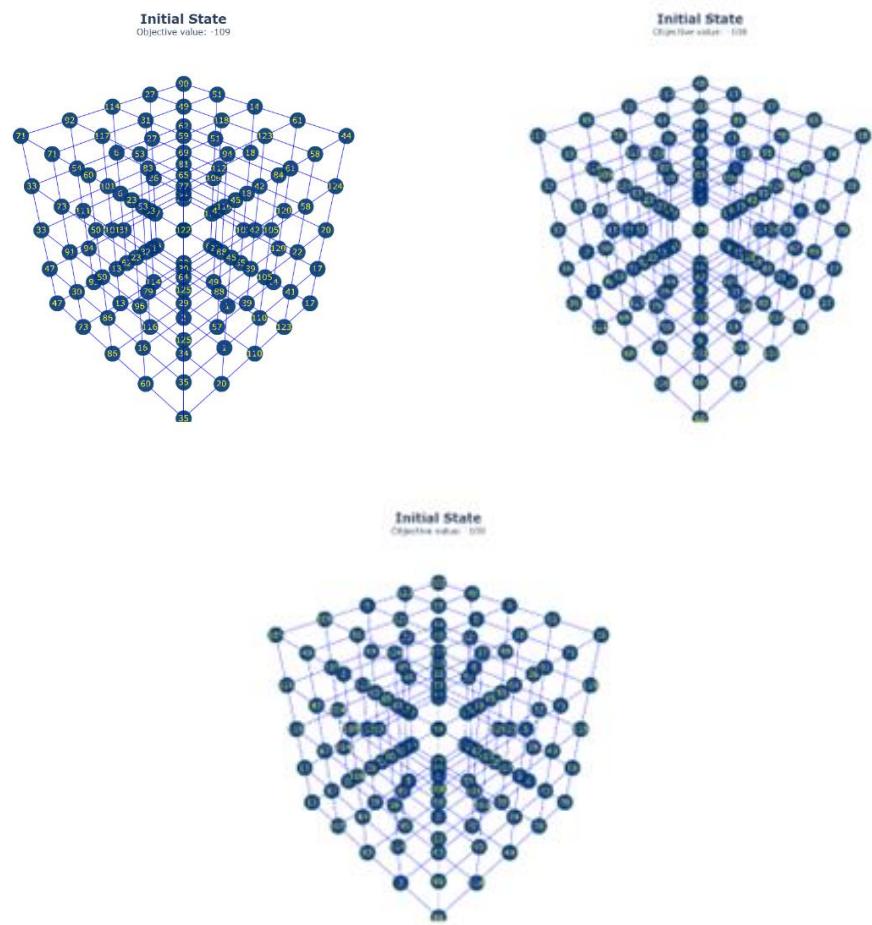
7. Setiap *offsprings* yang dihasilkan dikumpulkan menjadi sebuah populasi baru, yakni *new population*. Jika sudah tidak ada *offspring* yang ditambahkan, *new population* digunakan untuk memperbarui *current population*.
8. Tahap 2–7 diulangi maksimal sebanyak (iterasi maksimum – 1) kali sampai ditemukan populasi dengan individu yang mencapai *global maximum*.

Berdasarkan tahap-tahap ini, dengan adanya populasi awal yang dibangkitkan secara acak, *crossover* yang acak, dan mutasi yang acak, jika tidak diberi batas iterasi maksimum, maka *global maximum* masih ada kemungkinan didapatkan. Berikut adalah *source code* dari implementasi algoritma *random restart hill-climbing*:

Genetic Algorithm

2.3 Hasil Eksperimen

Metode eksperimen yang dilakukan adalah dengan menyiapkan tiga *state* awal dari eksperimen pada setiap algoritma. Terkecuali untuk algoritma genetic algorithm dengan total 18 eksperimen. Tiga *state* awal yang disiapkan sebagai berikut:

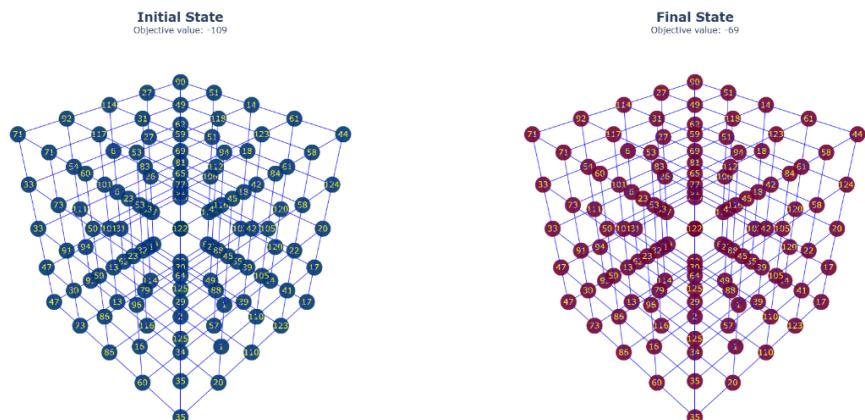


Gambar 2 Tiga State Awal Untuk Eksperimen Algoritma Local Search Kecuali Genetic Algorithm

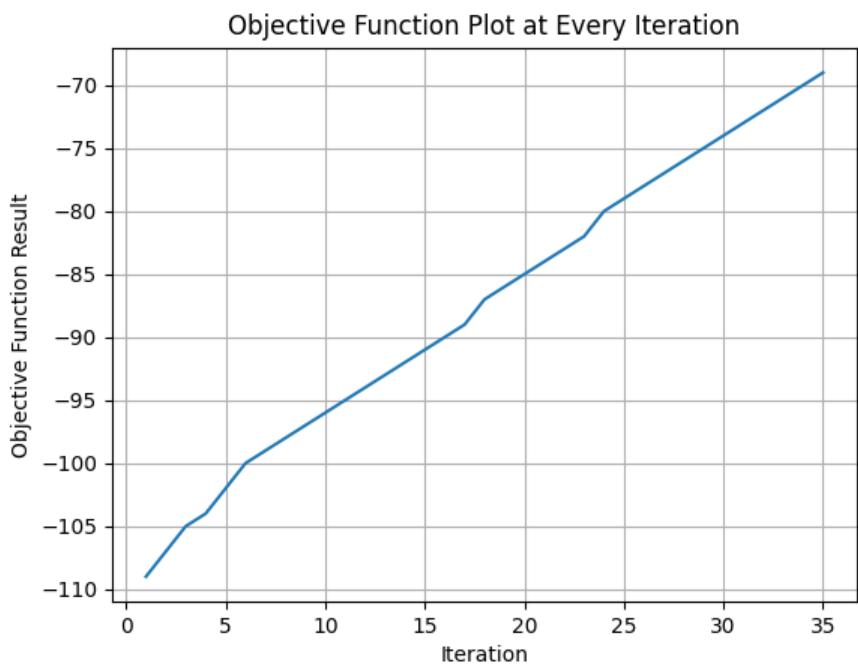
2.3.1 Steepest Ascent Hill-Climbing

Berikut adalah hasil eksperimen untuk tiap *state* awal:

1. State awal 1 dapat dituliskan ke dalam bentuk
 - State awal & akhir



- Plot Objektif vs Iterasi

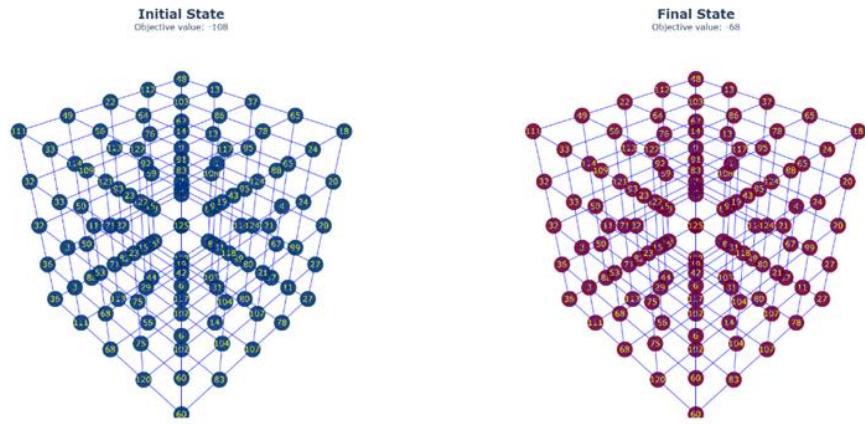


- Durasi & jumlah iterasi

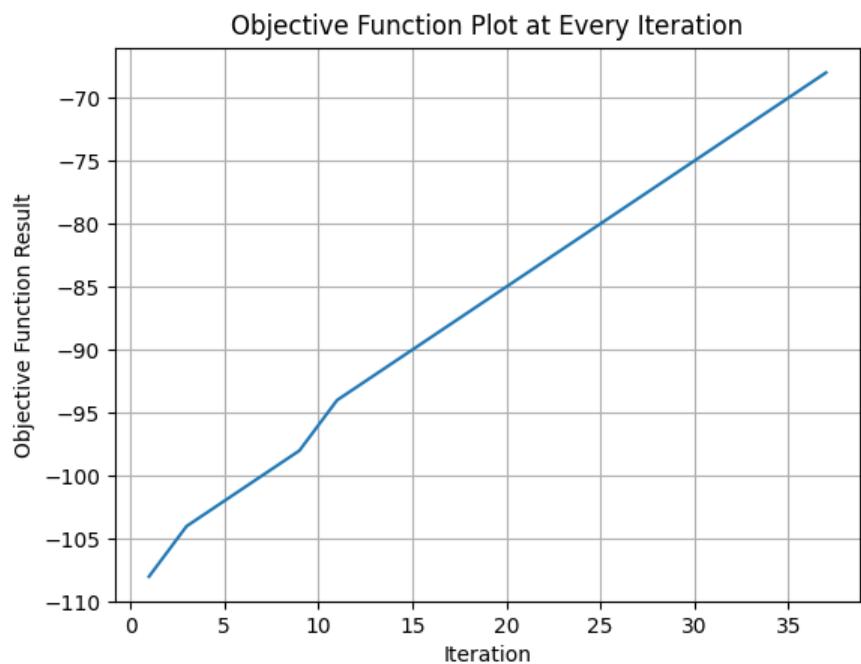
Steepest Ascent (Initial State 1)
Duration: 1.7923 seconds
Iterations: 34

2. State awal 2 dapat dituliskan ke dalam bentuk

- State awal & akhir



- Plot Objektif vs iterasi

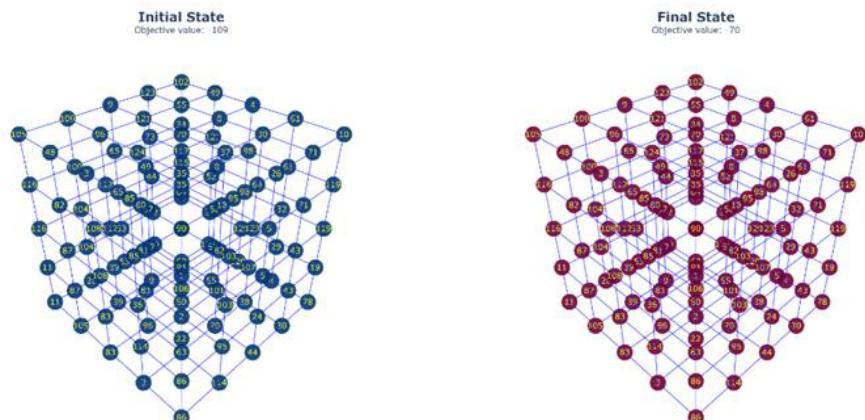


- Durasi & jumlah iterasi

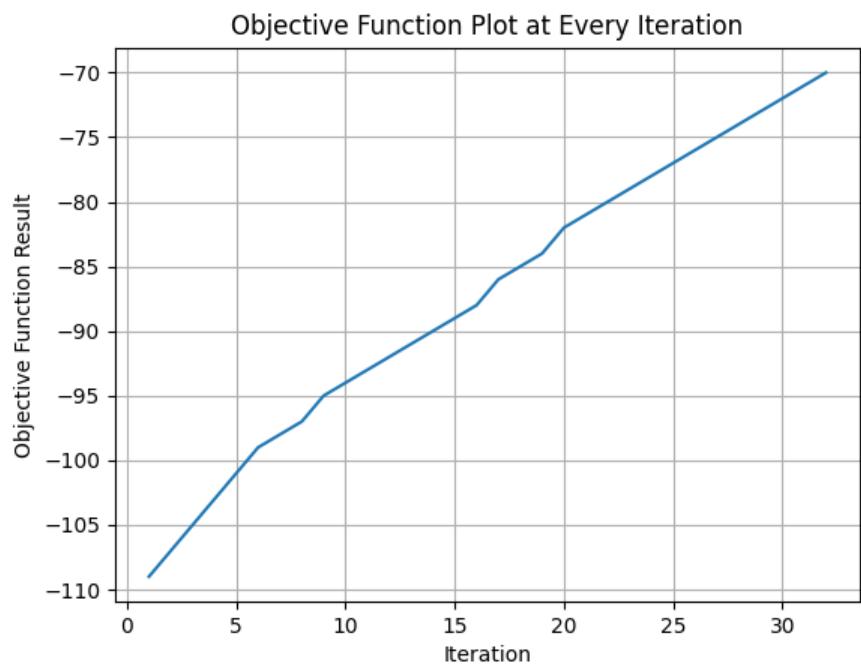
Steepest Ascent (Initial State 2)
Duration: 1.6611 seconds
Iterations: 36

3. State awal 3 dapat dituliskan ke dalam bentuk

- State awal & state akhir



- Plot Objektif vs Iterasi



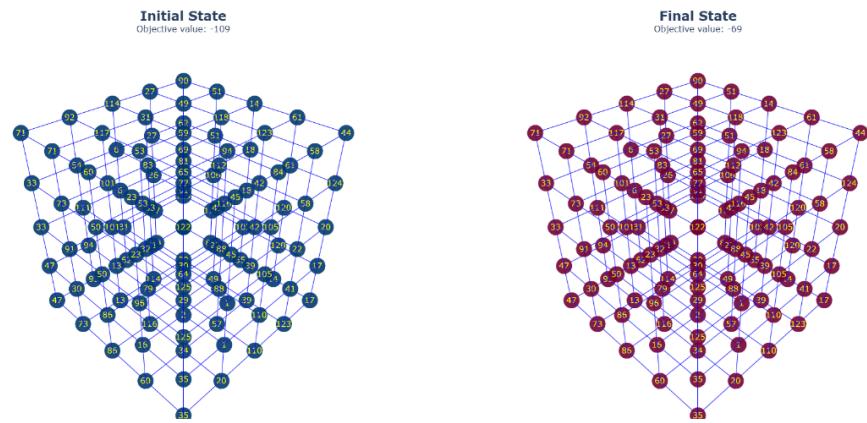
- Durasi & jumlah iterasi

Steepest Ascent (Initial State 3)
Duration: 1.5387 seconds
Iterations: 31

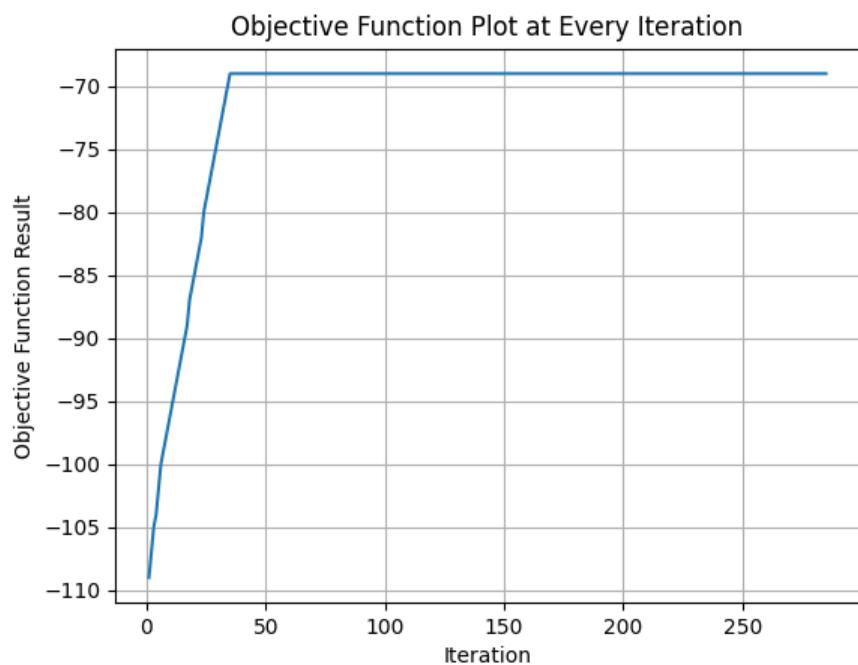
2.3.2 Hill-Climbing with Sideways Move

Berikut adalah hasil eksperimen untuk tiap *state* awal:

1. State awal 1 dapat dituliskan ke dalam bentuk
 - Maximum *sideways move* yang diperbolehkan dengan input dari pengguna **250**
 - State awal & akhir



- Plot Objektif vs Iterasi

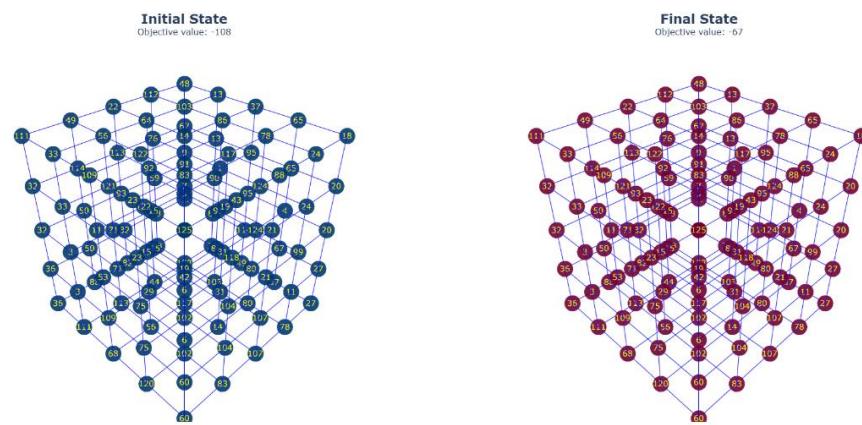


- Durasi & jumlah iterasi

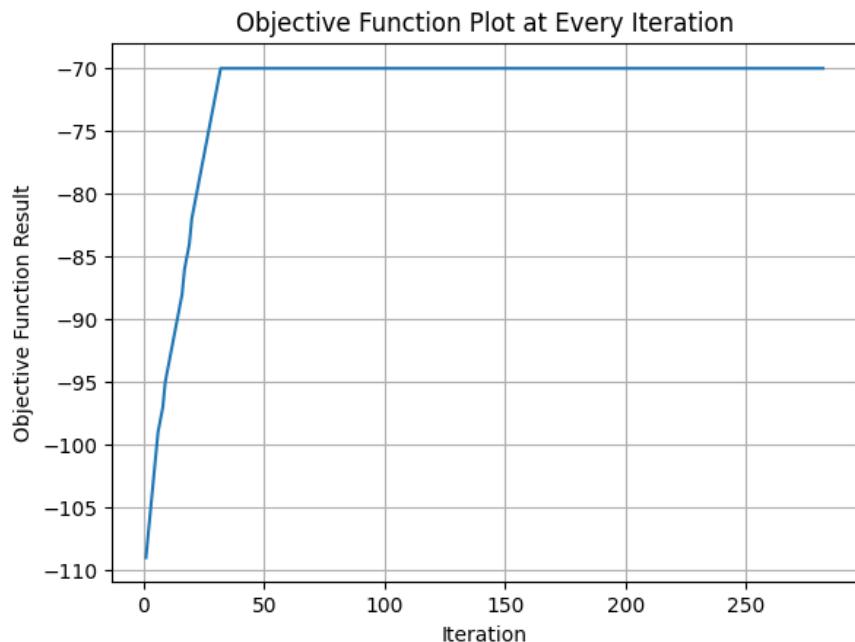
Sideways Move (Initial State 1)
Duration: 15.2752 seconds
Iterations: 284

2. State awal 2 dapat dituliskan ke dalam bentuk

- Maximum *sideways move* yang diperbolehkan dengan input dari pengguna **250**
- State awal & akhir



- Plot Objektif vs Iterasi

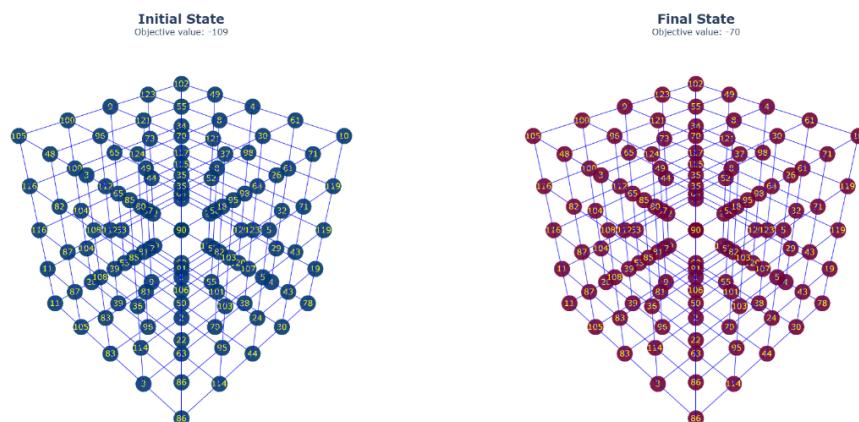


- Durasi & jumlah iterasi

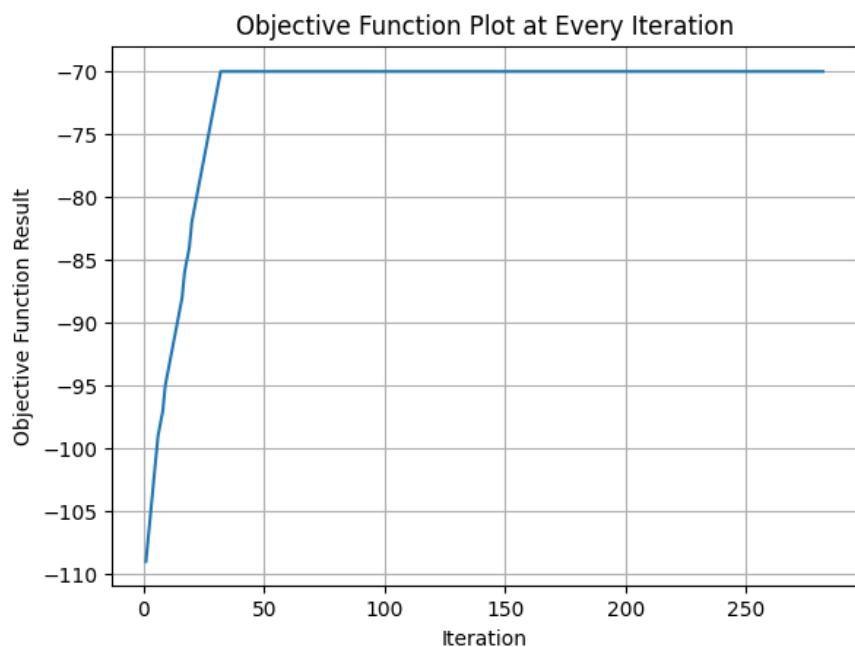
Sideways Move (Initial State 2)
Duration: 14.2646 seconds
Iterations: 288

3. State awal 3 dapat dituliskan ke dalam bentuk

- Maximum *sideways move* yang diperbolehkan dengan input dari pengguna **250**
- State awal & akhir



- Plot Objektif vs Iterasi

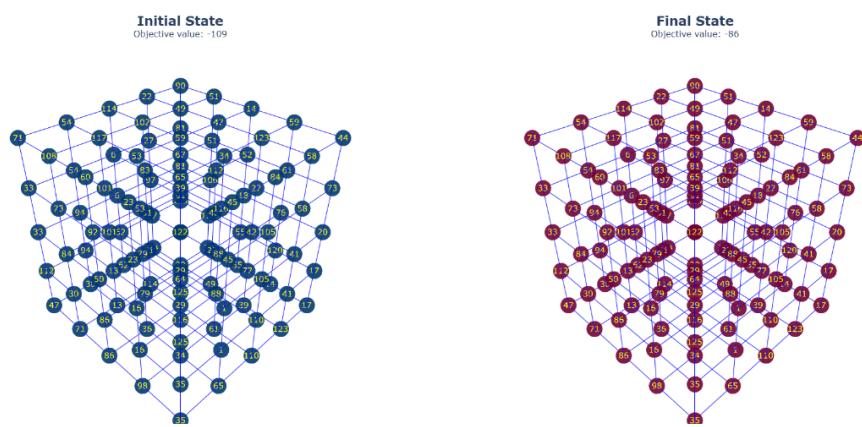


- Durasi & jumlah iterasi

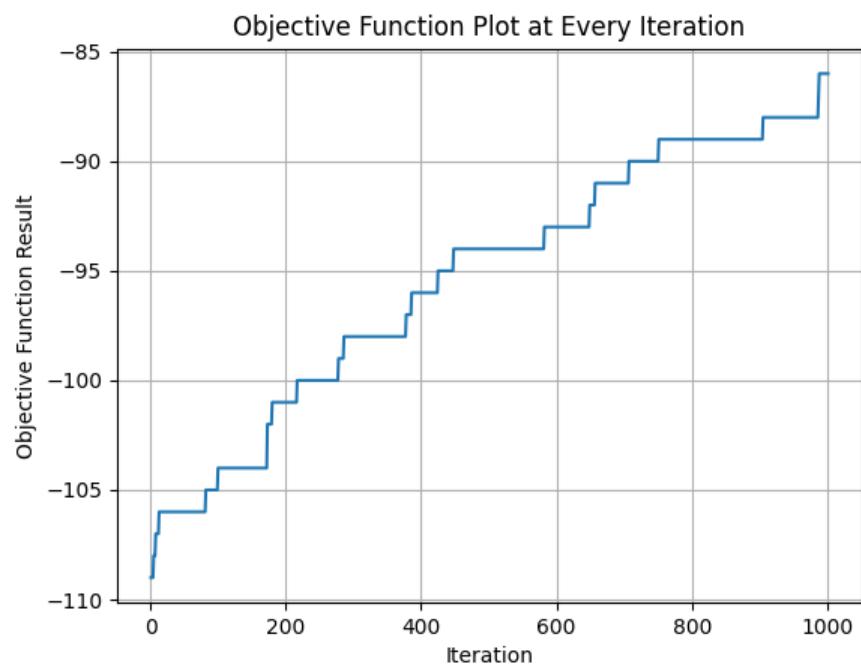
Sideways Move (Initial State 3)
Duration: 15.1880 seconds
Iterations: 281

2.3.3 Stochastic Hill-Climbing

1. State awal 1 dapat dituliskan ke dalam bentuk
 - Maximum iterasi yang diperbolehkan dengan input dari pengguna = 1000
 - State awal & akhir



- Plot Objektif vs Iterasi

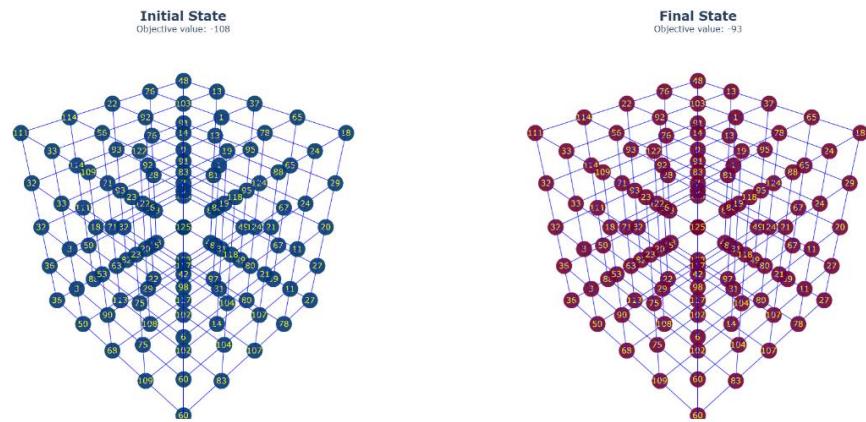


- Durasi & jumlah iterasi:

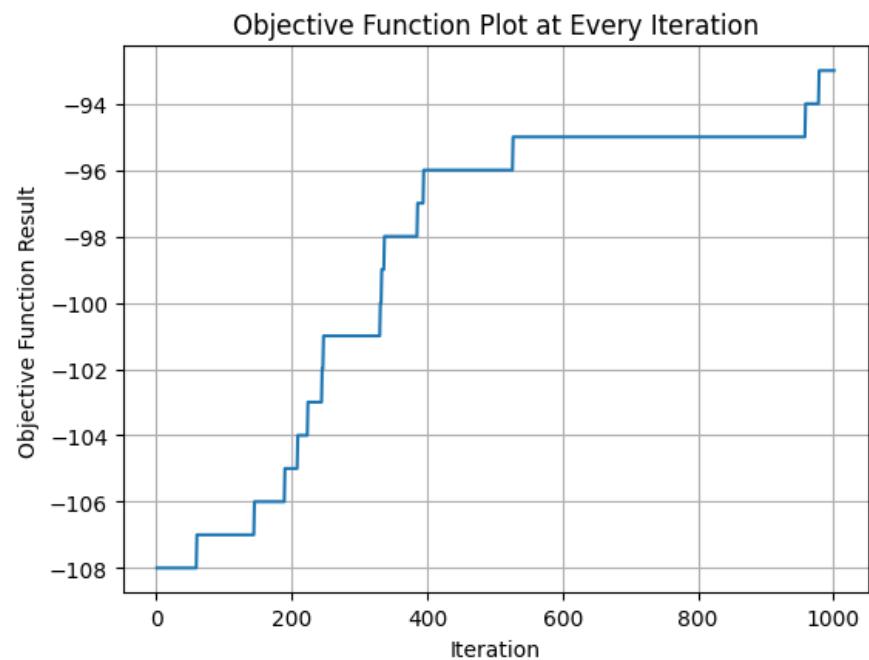
```
Stochastic Hill Climbing (Initial State 1)
Duration: 4.5225 seconds
Iterations: 1000
```

2. State awal 2 dapat dituliskan ke dalam bentuk

- Maximum iterasi yang diperbolehkan dengan input dari pengguna = 1000
- State awal & akhir



- Plot Objektif vs Iterasi



- Durasi & jumlah iterasi

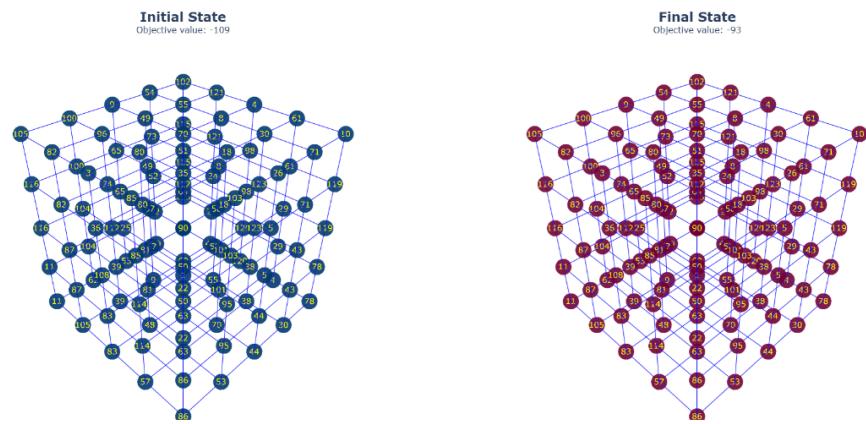
```

Stochastic Hill Climbing (Initial State 2)
Duration: 4.4097 seconds
Iterations: 1000

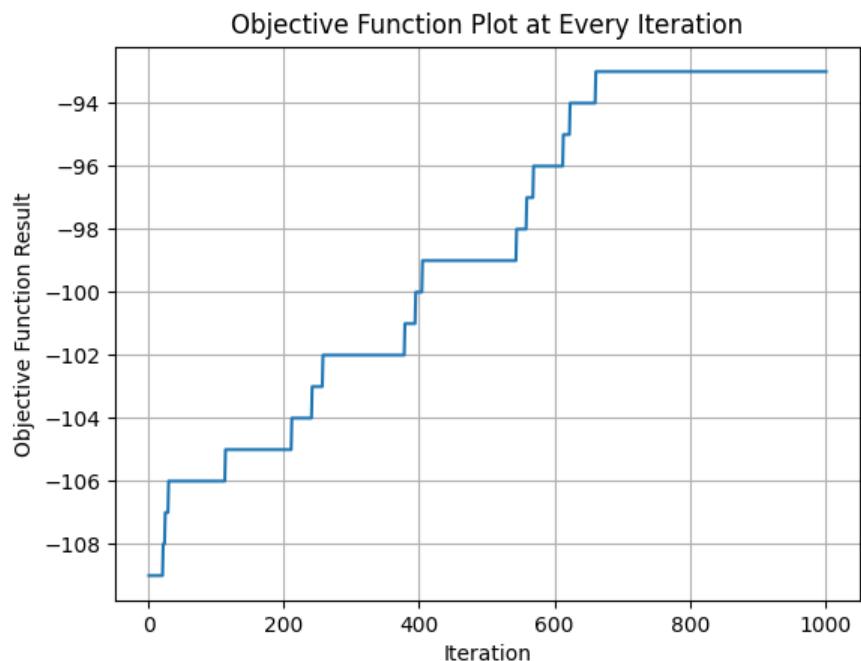
```

3. State awal 3 dapat dituliskan ke dalam bentuk

- Maximum iterasi yang diperbolehkan dengan input dari pengguna = 1000
- State awal & akhir



- Plot Objektif vs Iterasi



- Durasi & jumlah iterasi

```

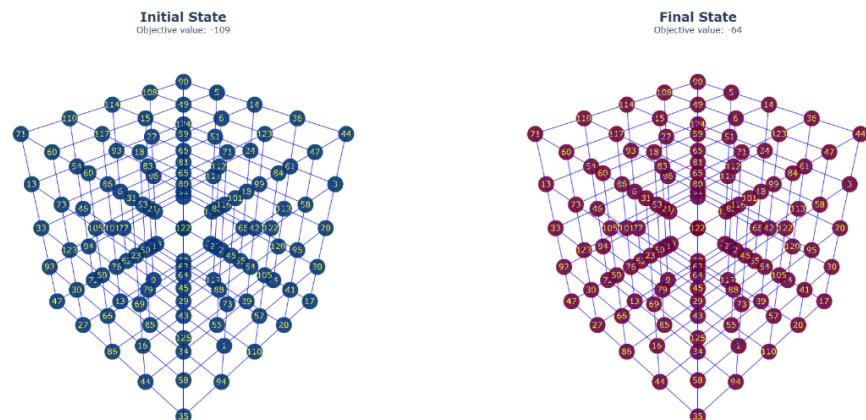
Stochastic Hill Climbing (Initial State 3)
Duration: 4.3053 seconds
Iterations: 1000

```

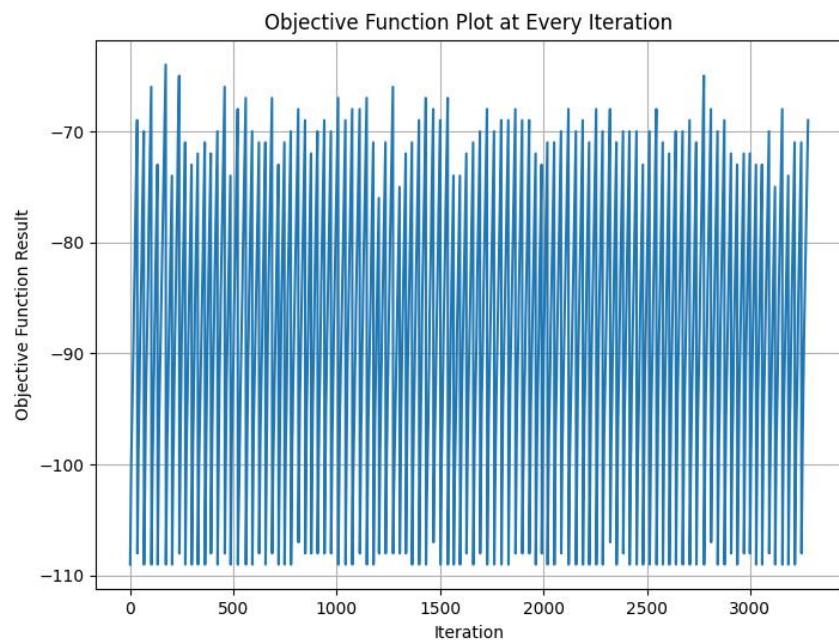
2.3.4 Random Restart Hill-Climbing

1. State awal 1 dapat dituliskan ke dalam bentuk

- Maximum restart yang digunakan dengan input dari pengguna =100
- State awal & akhir



- Plot Objektif vs Iterasi



- Durasi & jumlah iterasi

```

Random Restart (Initial State 1)
Duration: 169.8536 seconds
Iterations: 3279

```

- Banyak restart & banyak iterasi per re-start

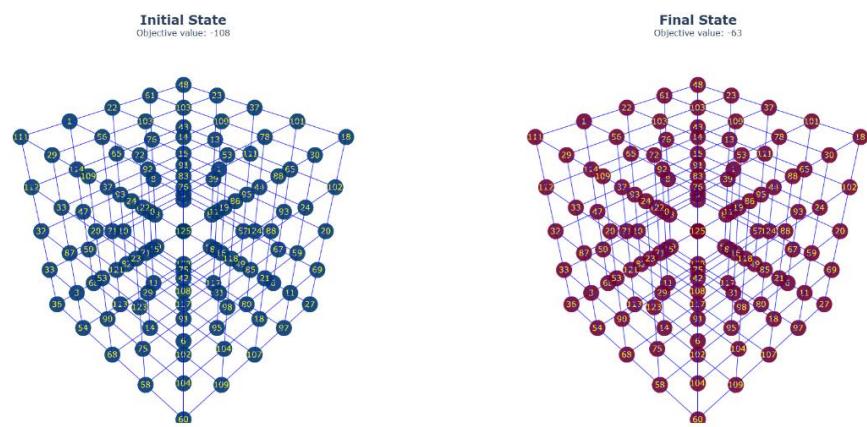
```

Total restarts: 108
Total iterations per restart: [34, 32, 36, 38, 40, 38, 35, 29, 31, 38, 34, 29, 32, 35, 28, 35, 39, 31, 33, 31, 32, 31, 29, 31, 36, 33, 29, 31, 33, 32, 35, 35, 33, 36, 34, 32, 27, 32, 36, 32, 38, 36, 33, 36, 34, 36, 28, 31, 31, 33, 35, 34, 36, 35, 34, 34, 33, 34, 31, 29, 28, 35, 33, 36, 33, 31, 34, 31, 36, 31, 32, 31, 34, 31, 32, 33, 31, 31, 32, 33, 34, 32, 37, 34, 32, 33, 32, 38, 32, 29, 38, 29, 34, 29, 35, 29, 31, 32]

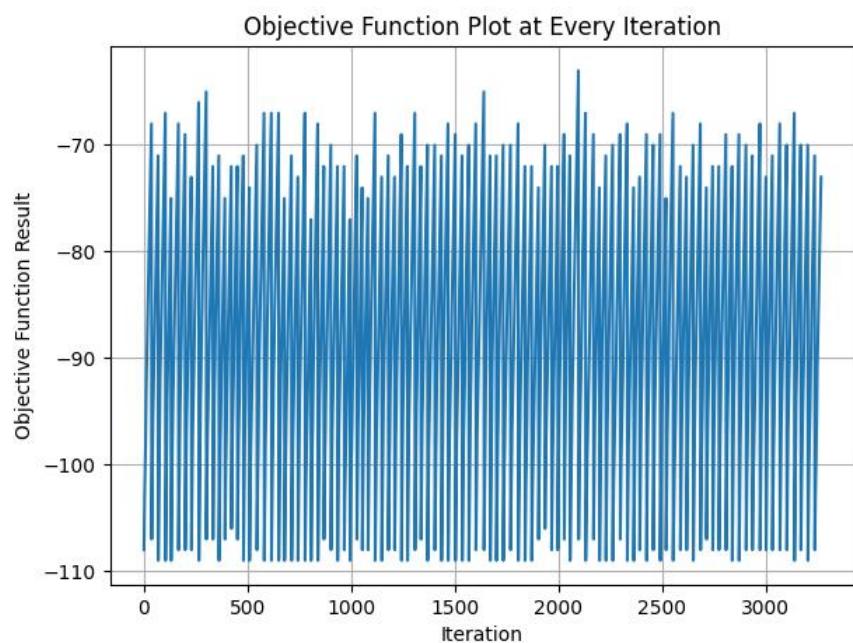
```

2. State awal 2 dapat dituliskan ke dalam bentuk

- Maximum restart yang digunakan dengan input dari pengguna = 100
- State awal & akhir



- Plot Objektif vs Iterasi



- Durasi & jumlah iterasi

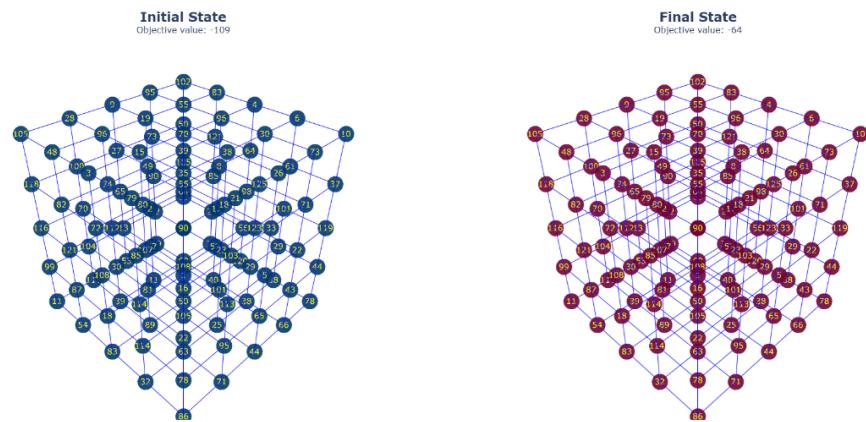
```
Random Restart (Initial State 2)
Duration: 174.4327 seconds
Iterations: 3265
```

- Banyak restart & banyak iterasi per-restart

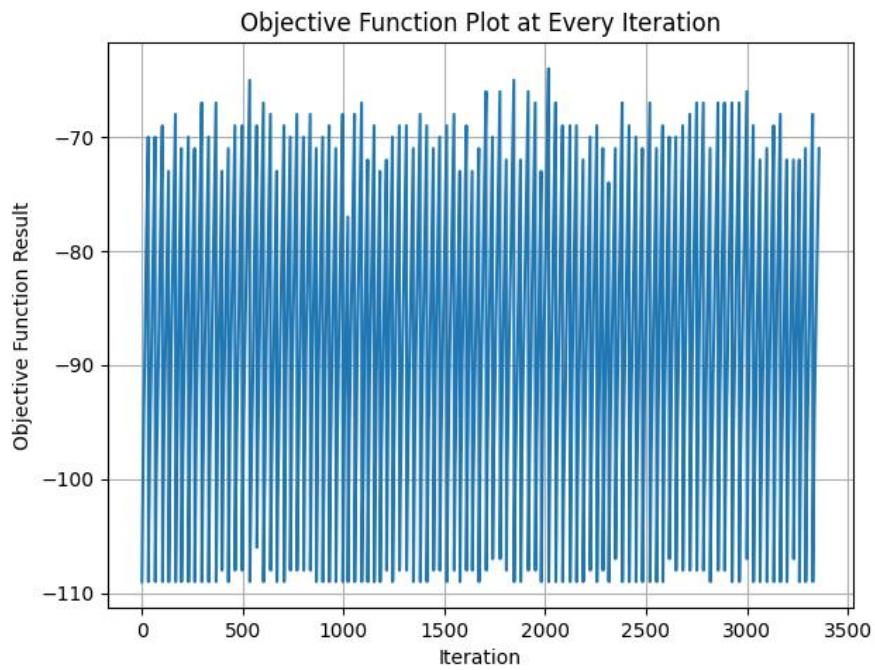
```
Total restarts: 100
Total iterations per restart: [36, 32, 35, 27, 36, 32, 30, 36, 36, 32, 29, 29, 29, 29, 36, 35, 35, 35, 28, 34, 31, 34, 29, 33, 29, 34, 32, 32, 28, 33, 26, 28, 34, 32, 32, 30, 33, 29, 36, 29, 32, 35, 32, 32, 35, 33, 32, 34, 39, 30, 38, 33, 35, 37, 33, 32, 33, 31, 32, 30, 31, 28, 41, 34, 38, 29, 31, 33, 36, 34, 31, 29, 33, 32, 34, 35, 30, 32, 35, 29, 30, 30, 34, 36, 33, 34, 36, 37, 29, 31, 36, 33, 37, 31, 34, 33]
```

3. State awal 3 dapat dituliskan ke dalam bentuk

- Maximum restart yang digunakan dengan input dari pengguna = 100
- State awal & state akhir



- Plot Objektif vs Iterasi



- Durasi & jumlah iterasi

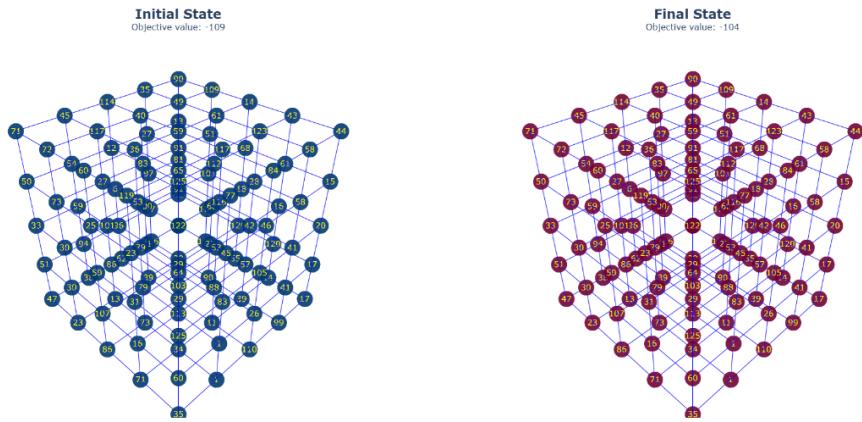
```
Random Restart (Initial State 3)
Duration: 185.4309 seconds
Iterations: 3357
```

- Banyak restart & banyak iterasi per re-start

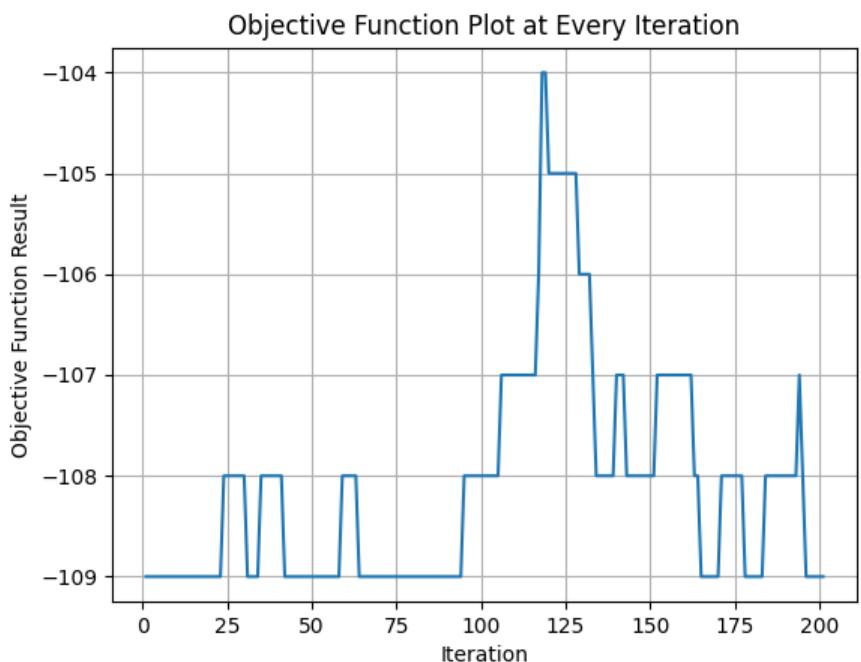
```
Total restarts: 100
Total iterations per restart: [33, 34, 36, 31, 34, 29, 35, 31, 35, 34, 37, 29, 32, 33, 35, 38, 35, 35, 35, 33, 36, 31, 33, 35, 33, 30, 32, 32, 33, 34, 26, 33, 35, 30, 32, 29, 32, 30, 34, 35, 34, 34, 33, 31, 35, 36, 29, 32, 29, 35, 36, 32, 37, 31, 38, 33, 38, 35, 29, 37, 35, 34, 36, 37, 32, 33, 35, 33, 30, 29, 35, 34, 34, 36, 32, 33, 31, 35, 33, 34, 39, 32, 37, 36, 38, 32, 33, 34, 34, 35, 32, 33, 29, 31, 35]
```

2.3.5 Simulated Annealing

1. State awal 1 dapat dituliskan ke dalam bentuk
 - State awal & ahir



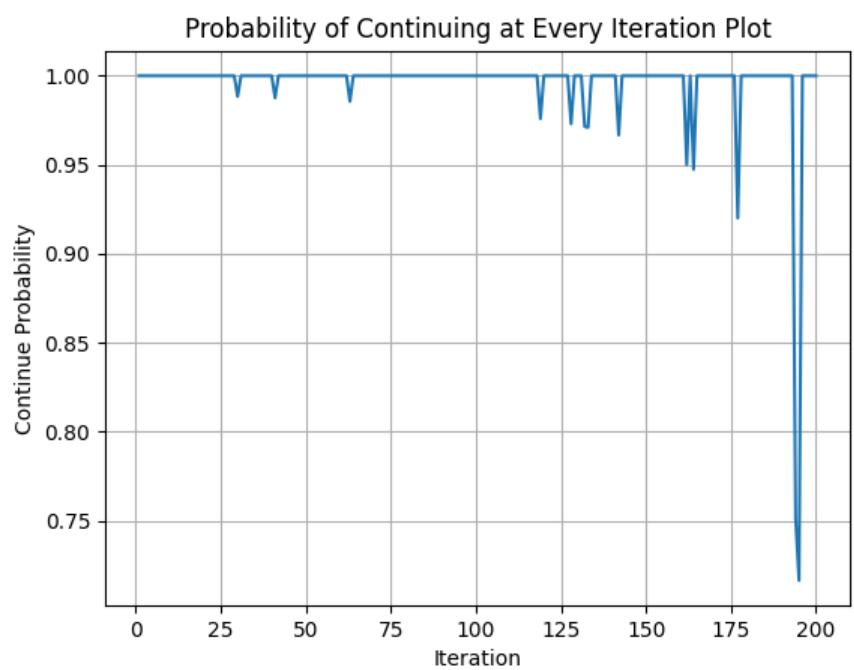
- Plot Objektif vs Iterasi



- Durasi, iterasi, dan frekuensi terjebak di lokal optimal

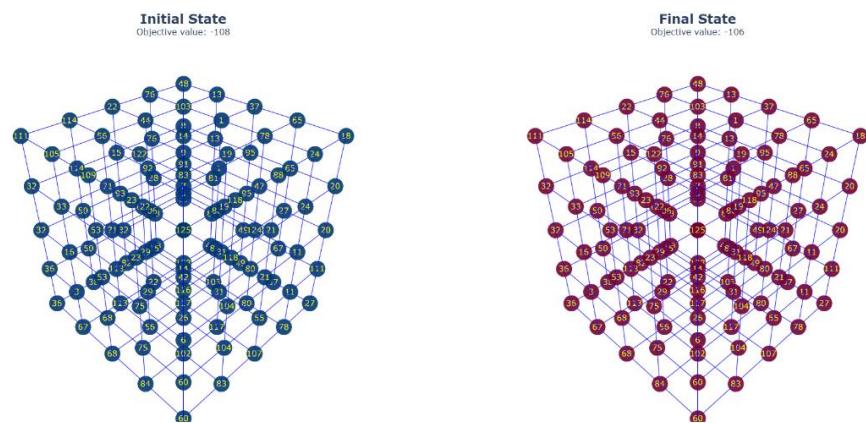
```
Simulated Annealing (Initial State 1)
Duration: 1.0052 seconds
Iterations: 200
Stuck in local optime frequency: 9
```

- Plot $e^{\Delta E/T}$ terhadap banyak iterasi yg dilewati

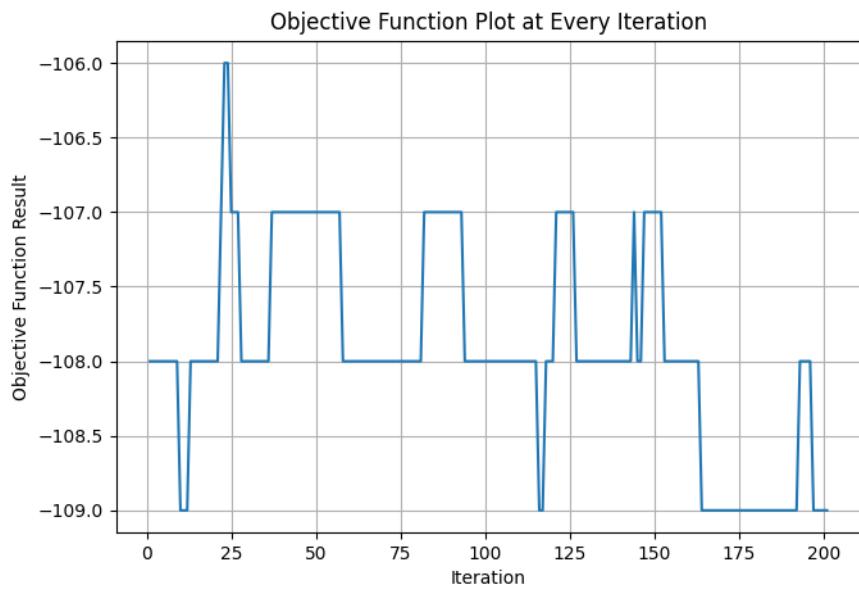


2. State awal 2 dapat dituliskan ke dalam bentuk

- State awal & akhir



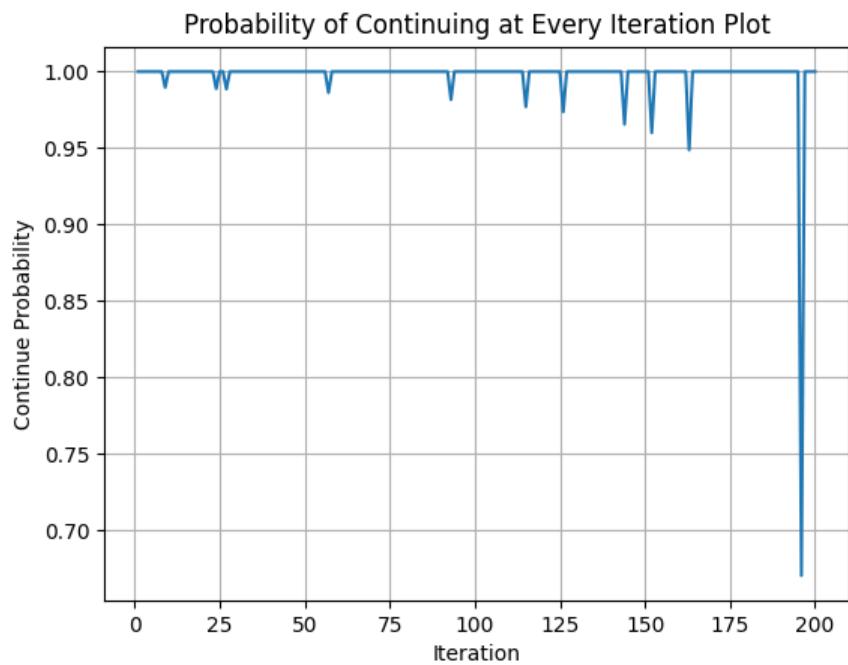
- Plot Objektif vs Iterasi



- Durasi, iterasi, dan frekuensi terjebak di lokal optimal

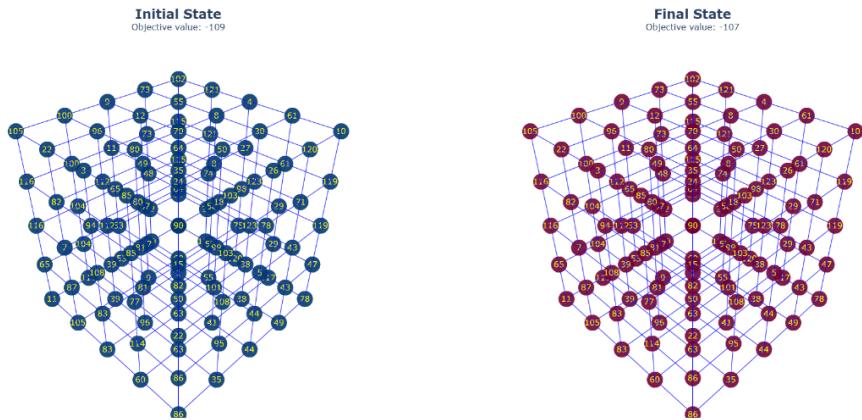
```
Simulated Annealing (Initial State 2)
Duration: 1.0602 seconds
Iterations: 200
Stuck in local optime frequency: 11
```

- Plot $e^{\Delta E/T}$ terhadap banyak iterasi yg dilewati

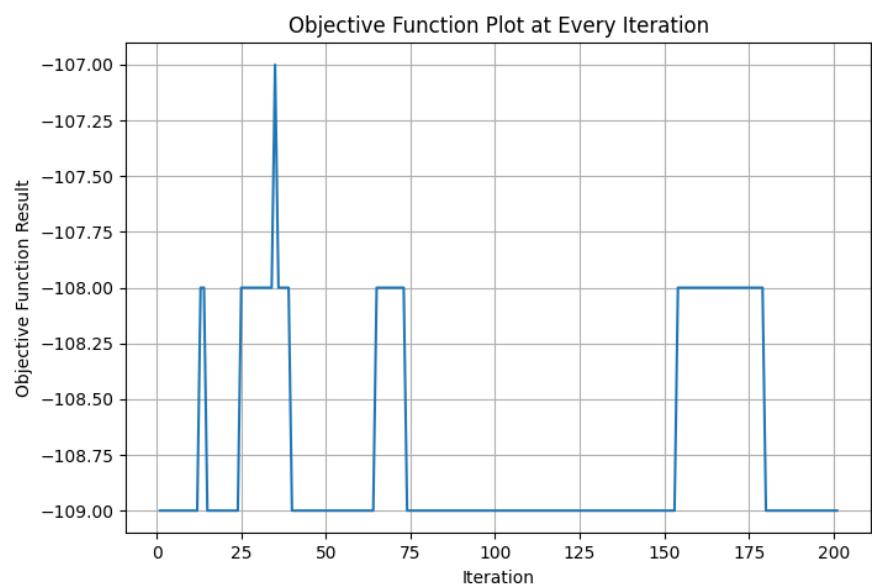


3. State awal 3 dapat dituliskan ke dalam bentuk

- State awal & akhir



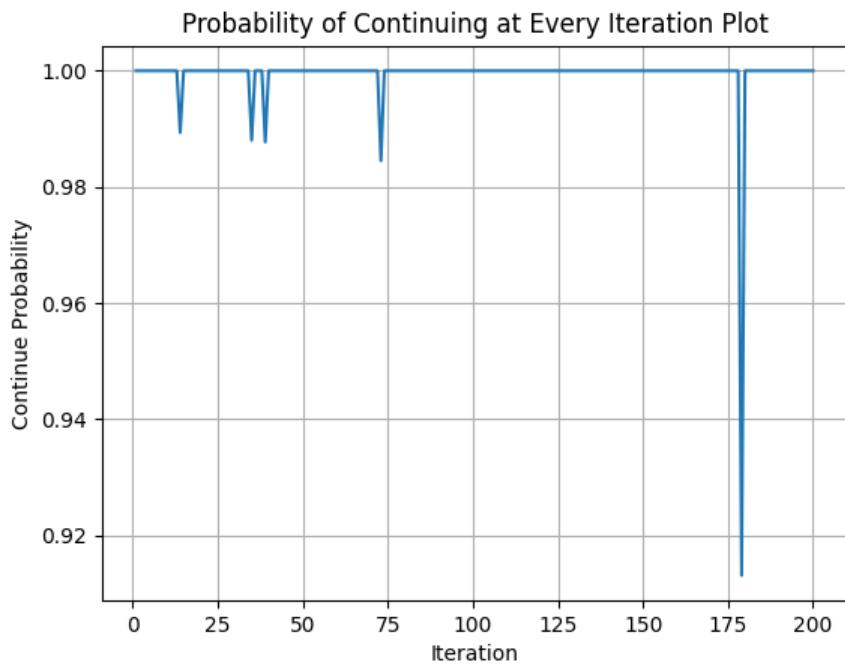
- Plot Objektif vs Iterasi



- Durasi, iterasi, dan frekuensi terjebak di lokal optimal

```
Simulated Annealing (Initial State 3)
Duration: 0.8767 seconds
Iterations: 200
Stuck in local optime frequency: 5
```

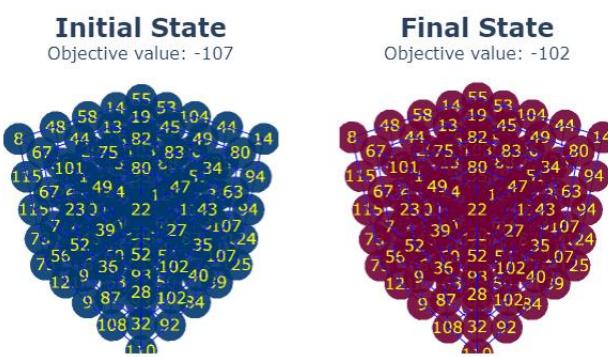
- Plot $e^{\Delta E/T}$ terhadap banyak iterasi yg dilewati



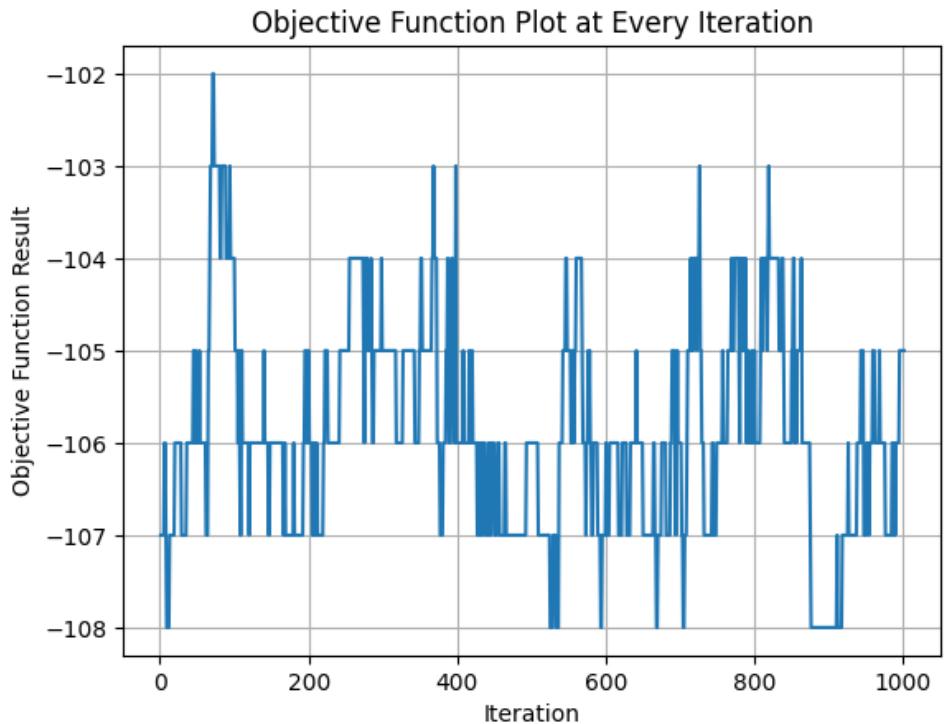
2.3.6 Genetic Algorithm

Pada algoritma ini, dilakukan 9 kali eksperimen, yakni 3 kasus input maksimum iterasi serta 3 kasus input jumlah populasi. Berikut adalah hasil dengan membandingkan banyak iterasi untuk tiap kasus populasi:

1. Populasi 1, iterasi 1, dapat dituliskan ke dalam bentuk
 - Jumlah populasi: 4
 - Iterasi maksimal: 1000
 - State awal & akhir



- Plot Objektif vs Iterasi

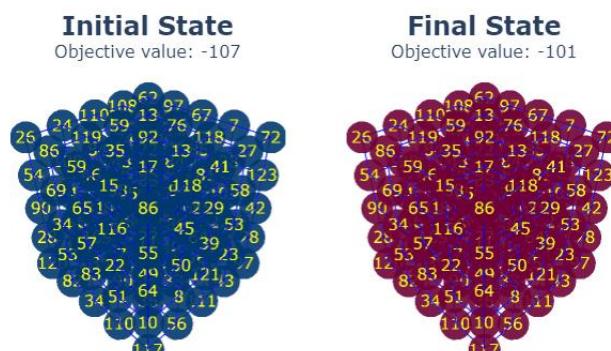


- Durasi & jumlah iterasi

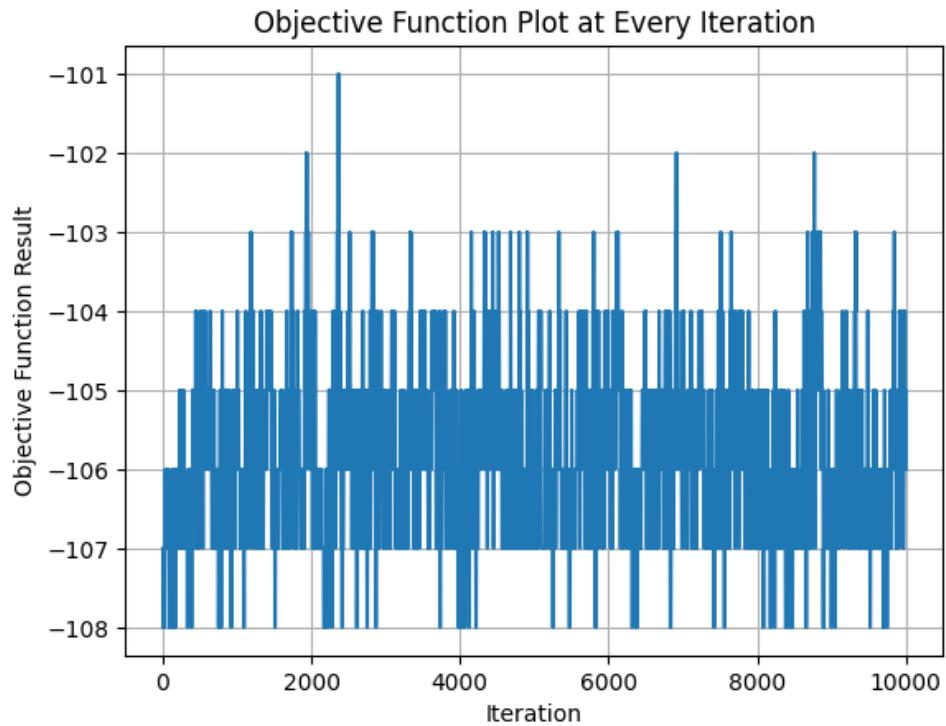
Iterations: 1000
Duration: 1.221075 seconds

2. Populasi 1, iterasi 2, dapat dituliskan ke dalam bentuk

- Jumlah populasi: 4
- Iterasi maksimal: 10000
- State awal & akhir



- Plot Objektif vs Iterasi

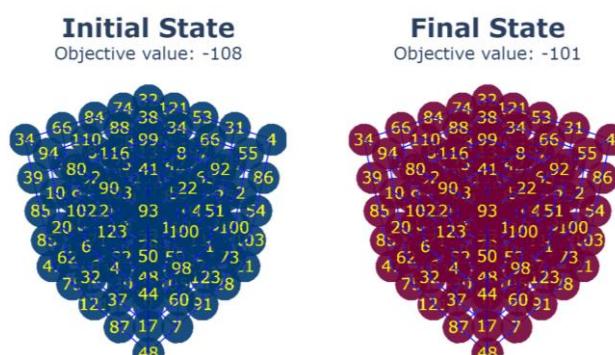


- Durasi & jumlah iterasi

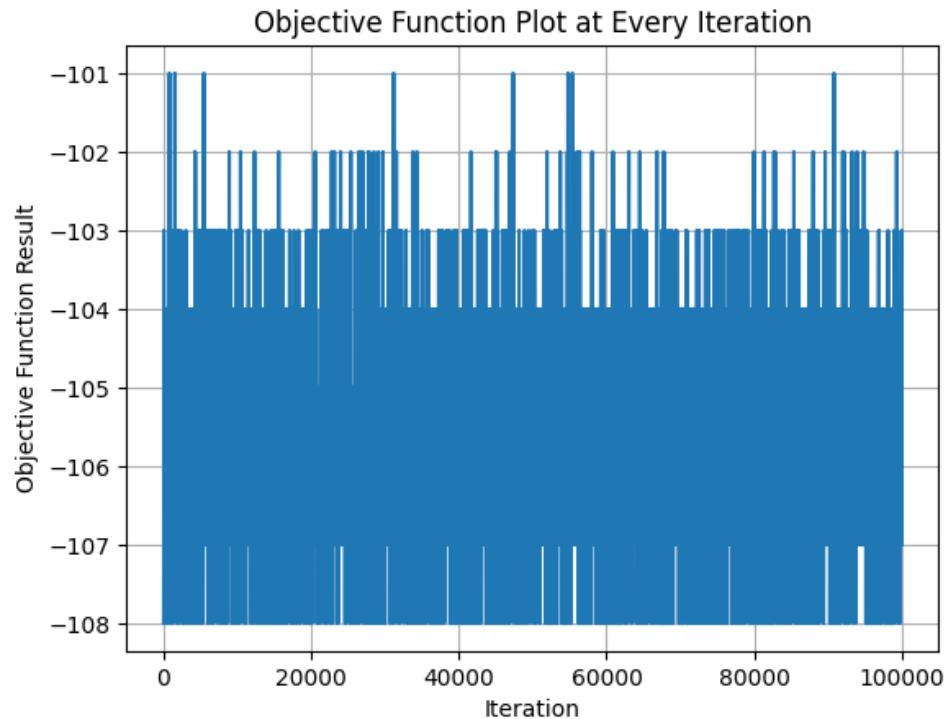
Iterations: 10000
Duration: 15.108501 seconds

3. Populasi 1, iterasi 3, dapat dituliskan ke dalam bentuk

- Jumlah populasi: 4
- Iterasi maksimal: 100000
- State awal & akhir



- Plot Objektif vs Iterasi

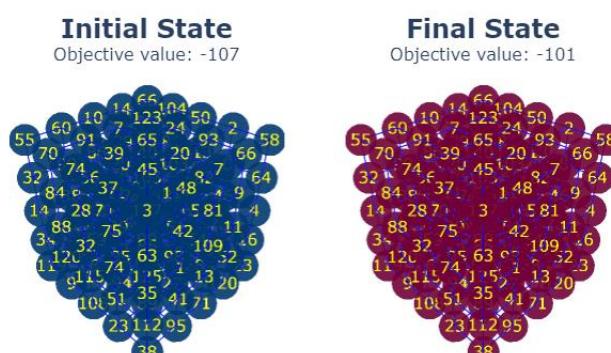


- Durasi & jumlah iterasi

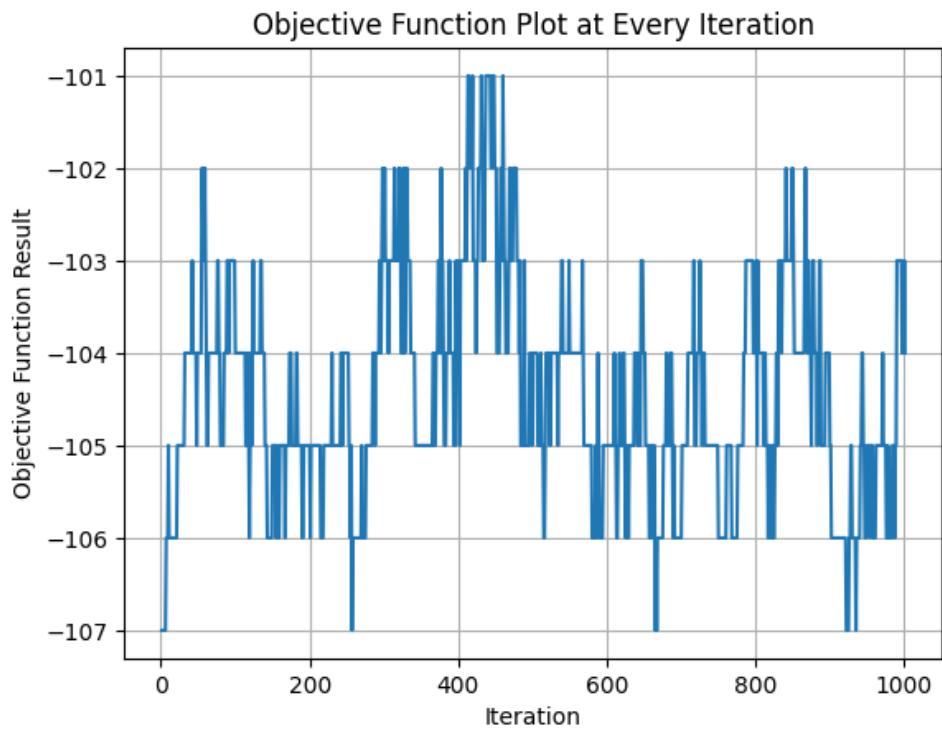
```
Iterations: 100000
Duration: 135.062885 seconds
```

4. Populasi 2, iterasi 1, dapat dituliskan ke dalam bentuk

- Jumlah populasi: 8
- Iterasi maksimal: 1000
- State awal & akhir



- Plot Objektif vs Iterasi

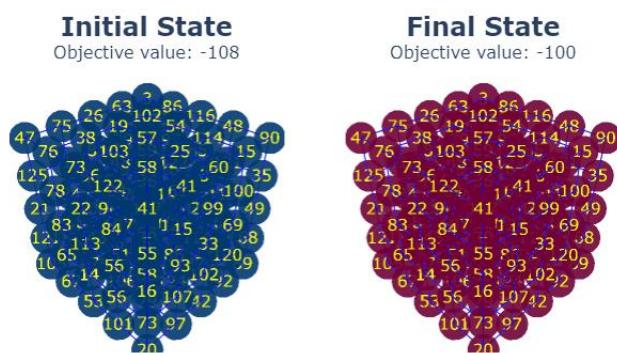


- Durasi & jumlah iterasi

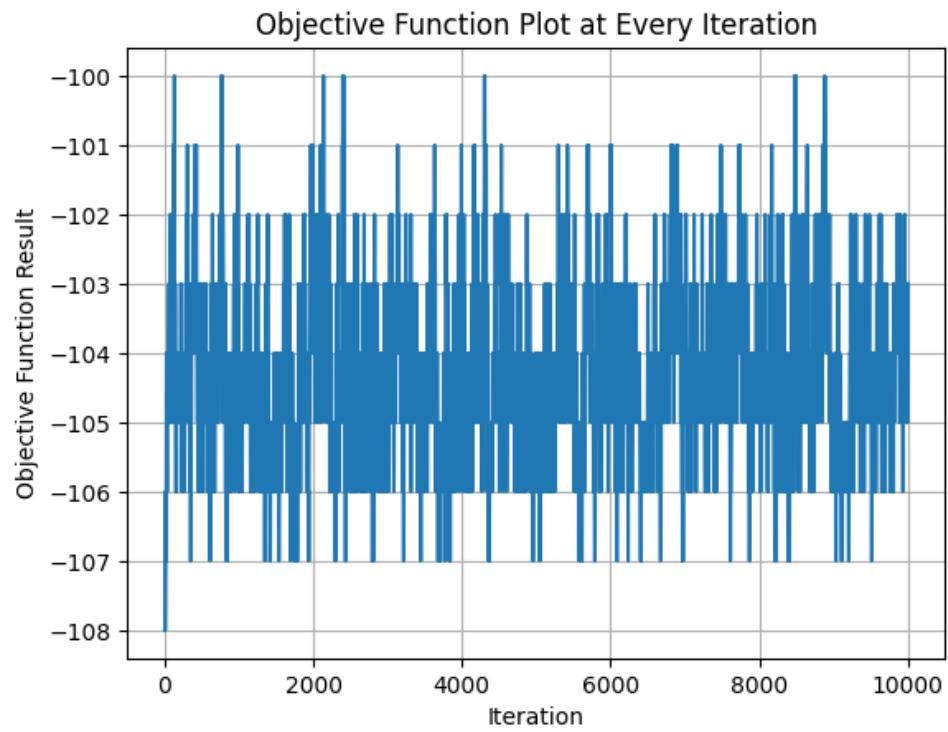
Iterations: 1000
Duration: 2.527229 seconds

5. Populasi 2, iterasi 2, dapat dituliskan ke dalam bentuk

- Jumlah populasi: 8
- Iterasi maksimal: 10000
- State awal & akhir



- Plot Objektif vs Iterasi

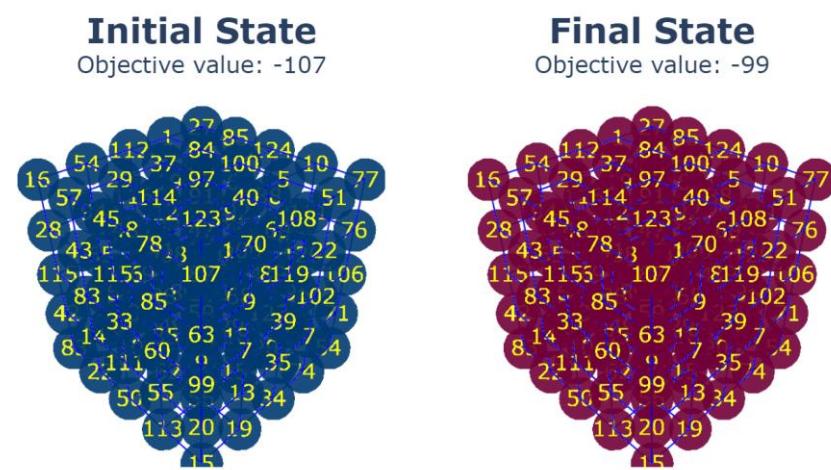


- Durasi & jumlah iterasi

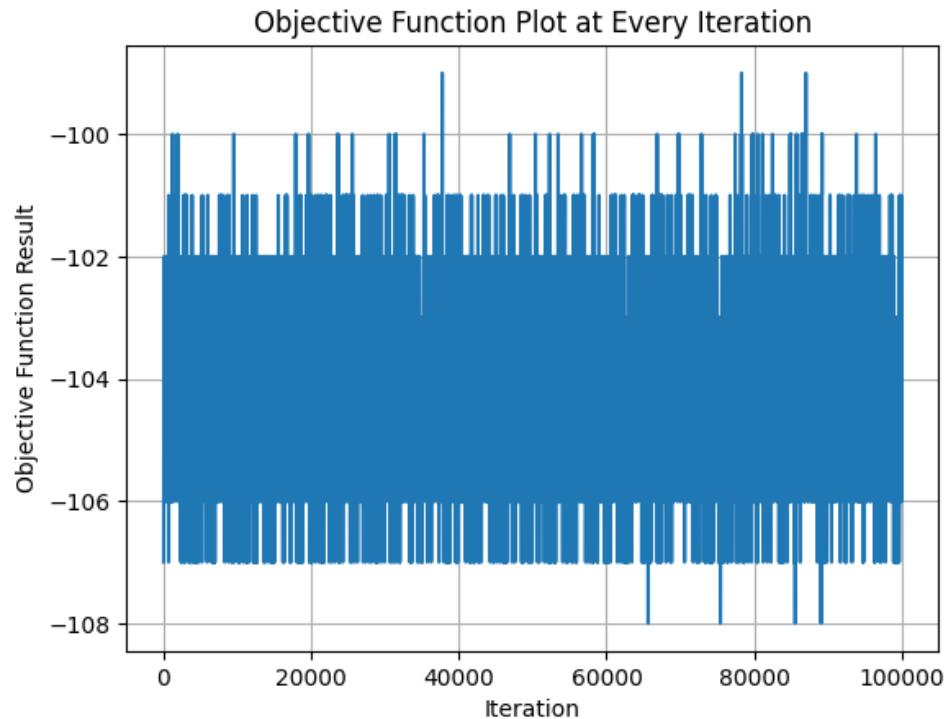
Iterations: 10000
Duration: 25.796117 seconds

6. Populasi 2, iterasi 3, dapat dituliskan ke dalam bentuk

- Jumlah populasi: 8
- Iterasi maksimal: 100000
- State awal & akhir



- Plot Objektif vs Iterasi

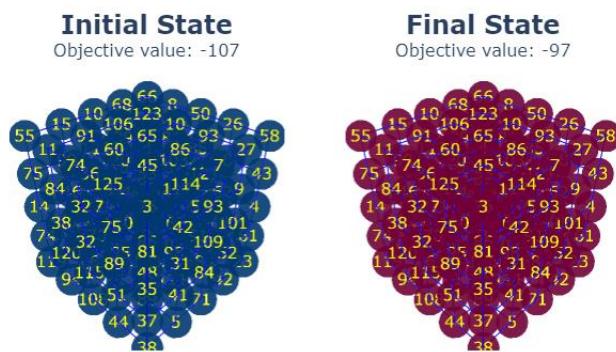


- Durasi & jumlah iterasi

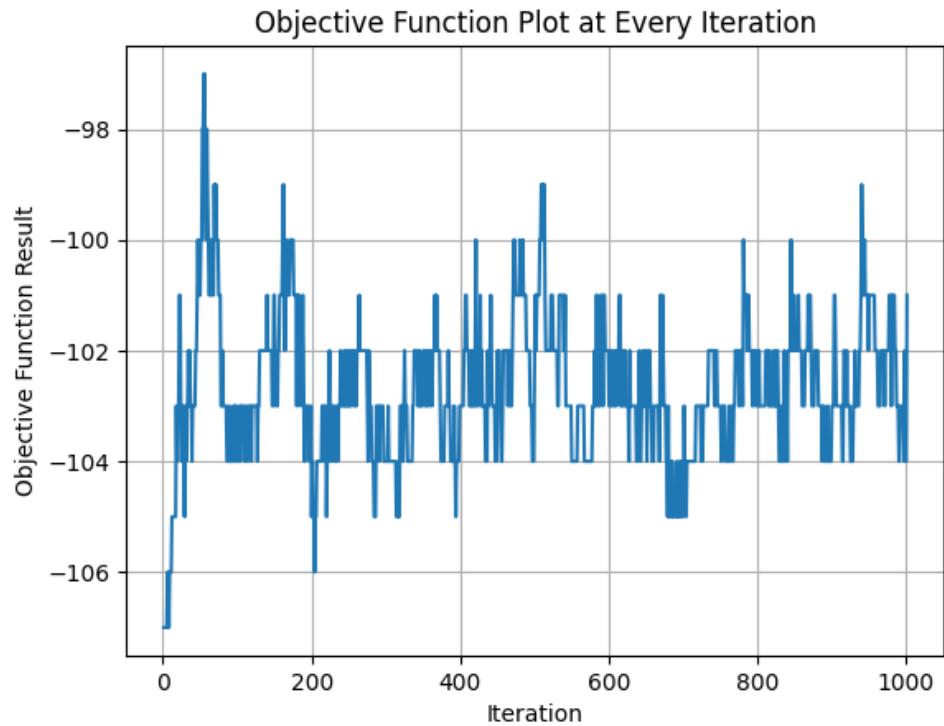
Iterations: 100000
Duration: 309.713641 seconds

7. Populasi 3, iterasi 1, dapat dituliskan ke dalam bentuk

- Jumlah populasi: 16
- Iterasi maksimal: 1000
- State awal & akhir



- Plot Objektif vs Iterasi:

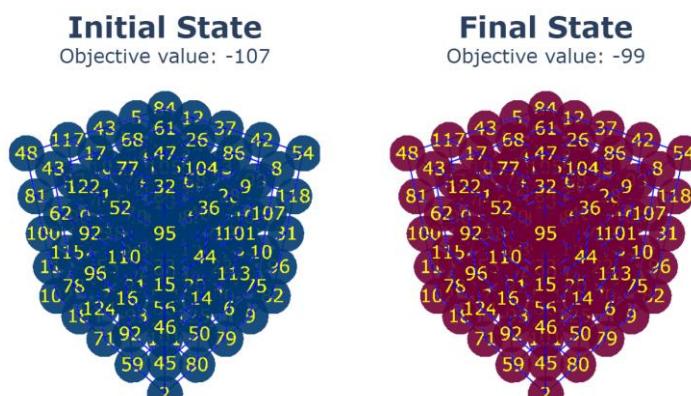


- Durasi & jumlah iterasi

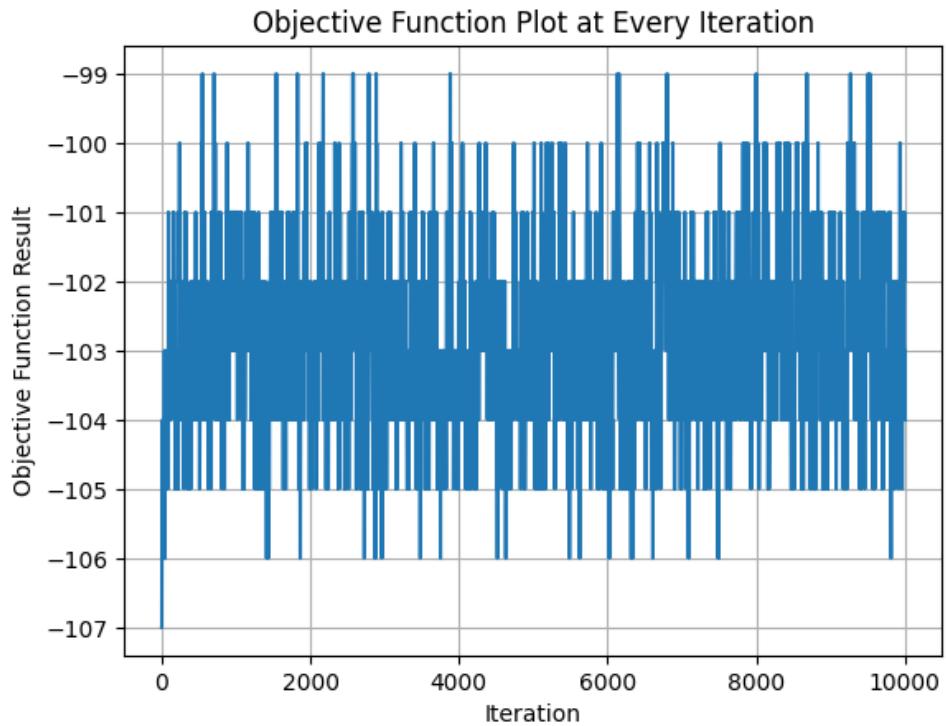
```
Iterations: 1000
Duration: 7.672157 seconds
```

8. Populasi 3, iterasi 2, dapat dituliskan ke dalam bentuk

- Jumlah populasi: 16
- Iterasi maksimal: 10000
- State awal & akhir



- Plot Objektif vs Iterasi

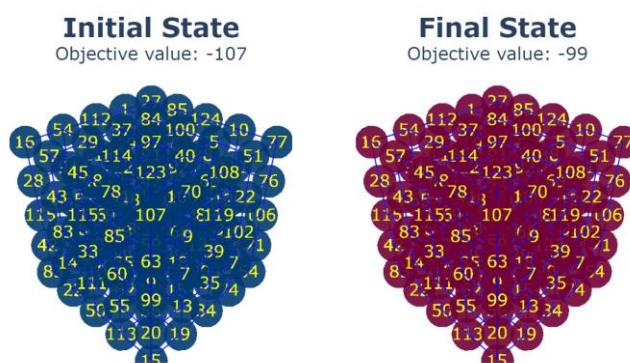


- Durasi & jumlah iterasi

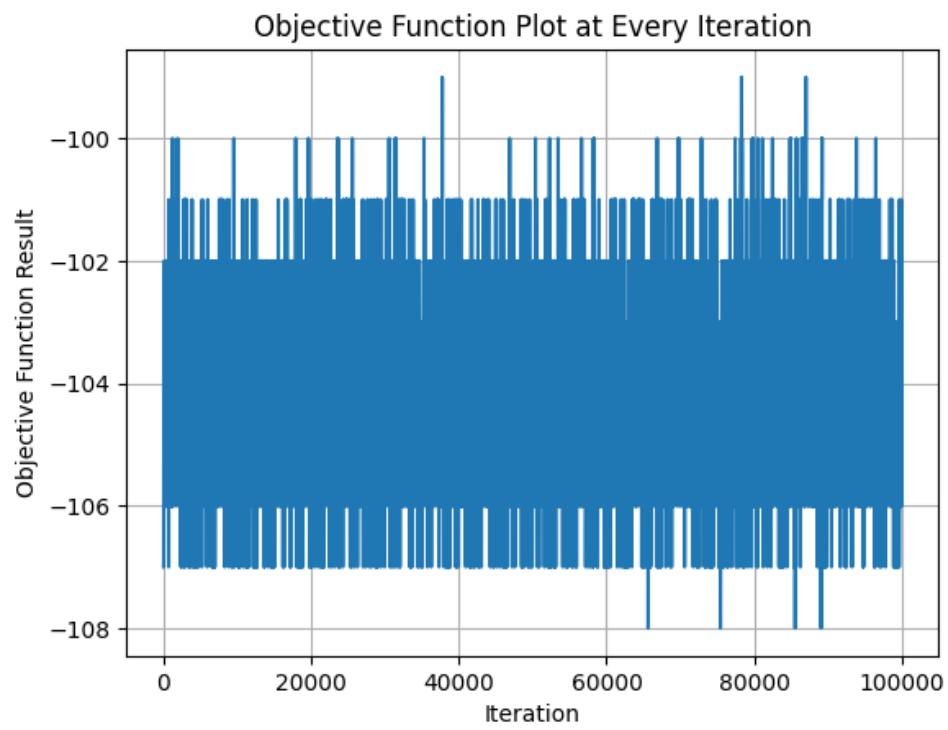
Iterations: 10000
Duration: 55.482302 seconds

9. Populasi 3, iterasi 3, dapat dituliskan ke dalam bentuk

- Jumlah populasi: 16
- Iterasi maksimal: 100000
- State awal & akhir



- Plot Objektif vs Iterasi

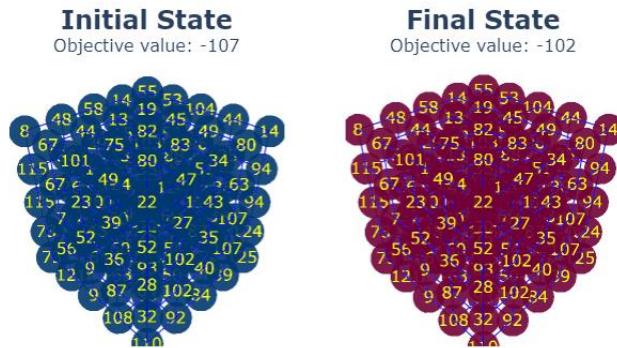


- Durasi & jumlah iterasi

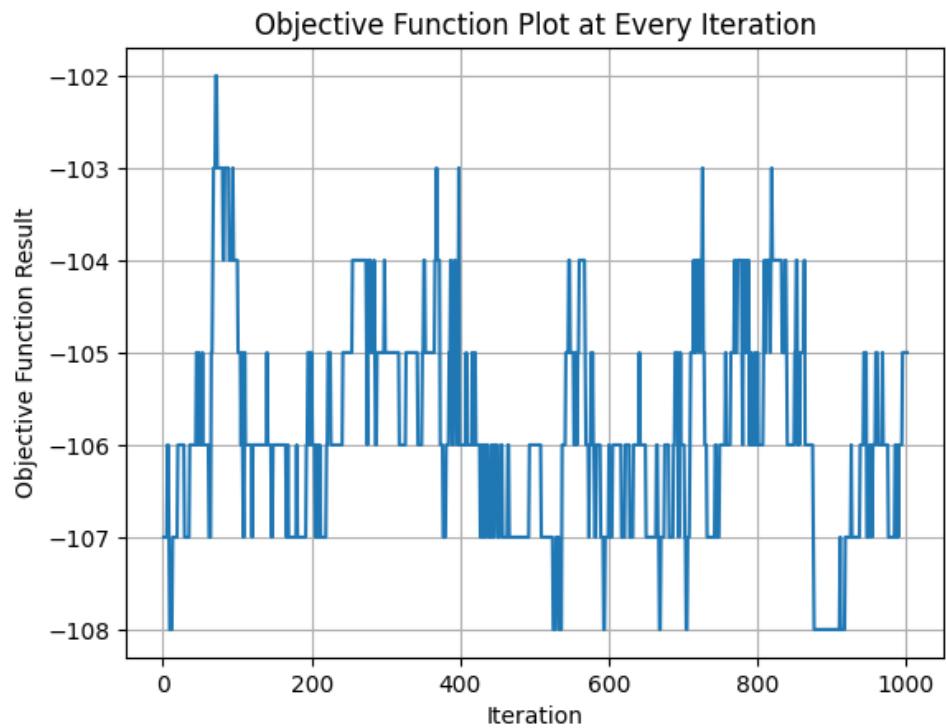
Iterations: 100000
Duration: 309.713641 seconds

Berikut adalah hasil dengan membandingkan jumlah populasi untuk tiap banyak kasus iterasi:

1. Iterasi 1, populasi 1, dapat dituliskan ke dalam bentuk
 - Jumlah populasi: 4
 - Iterasi maksimal: 1000
 - State awal & akhir



- Plot Objektif vs Iterasi

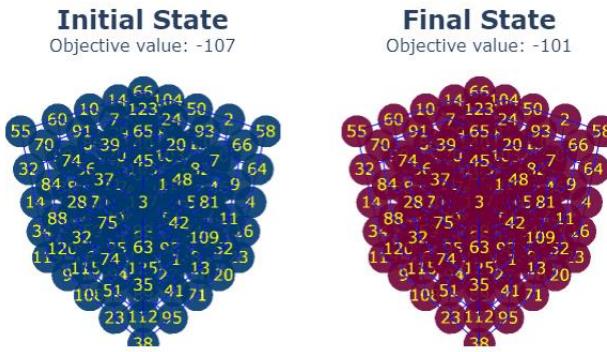


- Durasi & jumlah iterasi

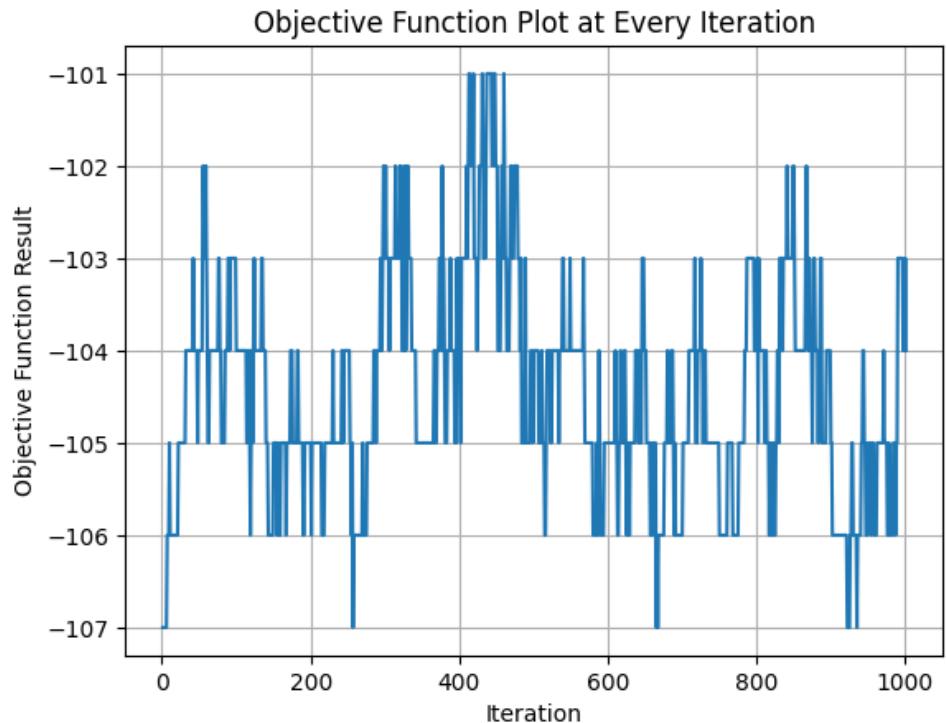
```
Iterations: 1000
Duration: 1.221075 seconds
```

2. Iterasi 1, populasi 2, dapat dituliskan ke dalam bentuk

- Jumlah populasi: 8
- Iterasi maksimal: 1000
- State awal & akhir



- Plot Objektif vs Iterasi

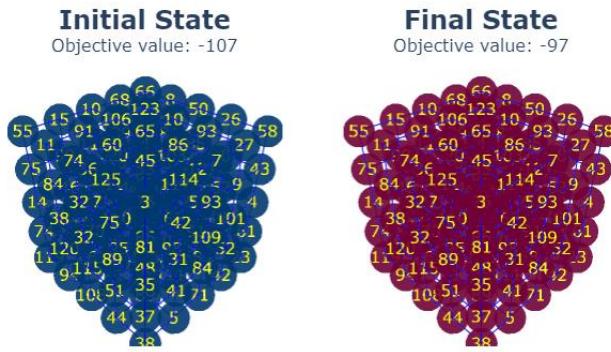


- Durasi & jumlah iterasi

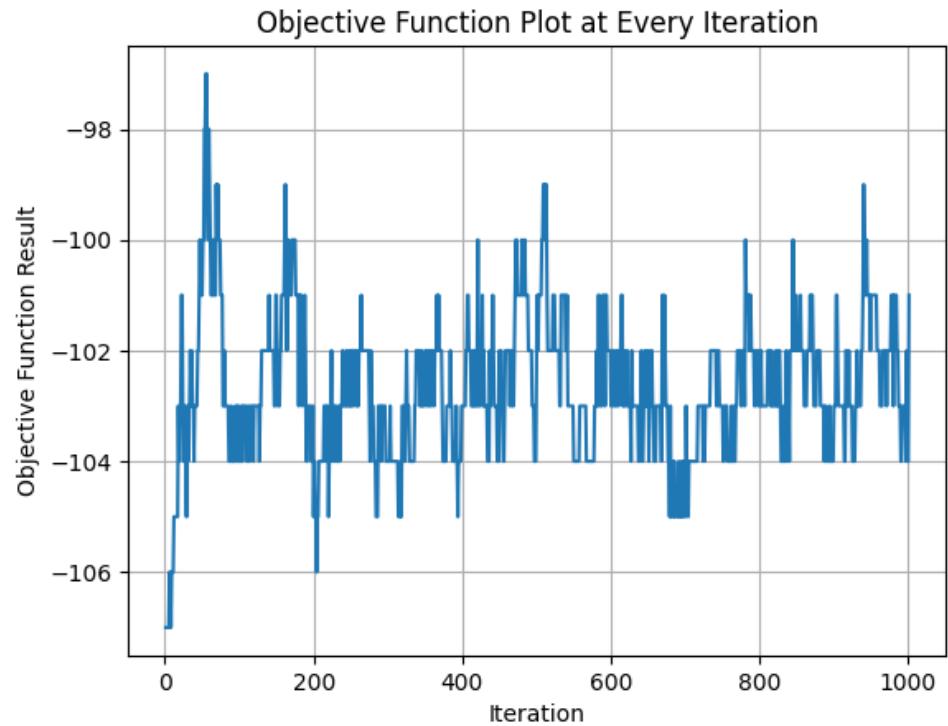
```
Iterations: 1000
Duration: 2.527229 seconds
```

3. Iterasi 1, populasi 3, dapat dituliskan ke dalam bentuk

- Jumlah populasi: 16
- Iterasi maksimal: 1000
- State awal & akhir



- Plot Objektif vs Iterasi



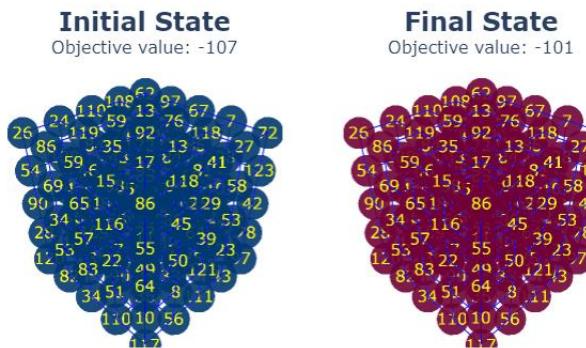
- Durasi & jumlah iterasi

```
Iterations: 1000
Duration: 7.672157 seconds
```

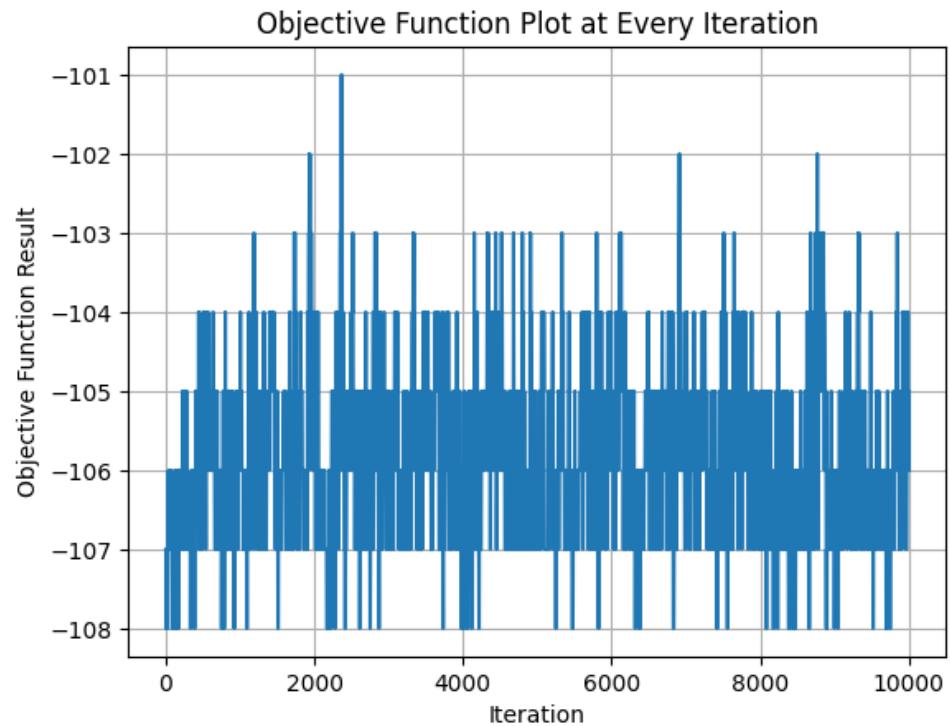
4. Iterasi 2, populasi 1, dapat dituliskan ke dalam bentuk

- Jumlah populasi: 4
- Iterasi maksimal: 10000

- State awal & akhir



- Plot Objektif vs Iterasi

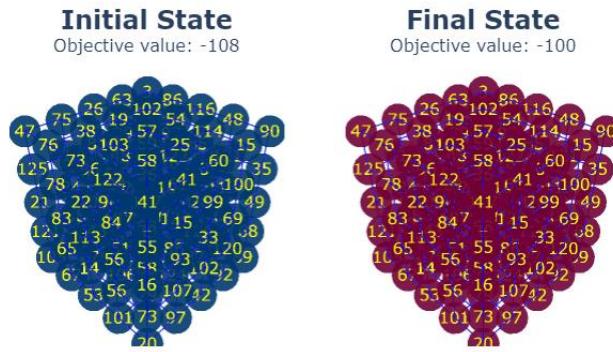


- Durasi & jumlah iterasi

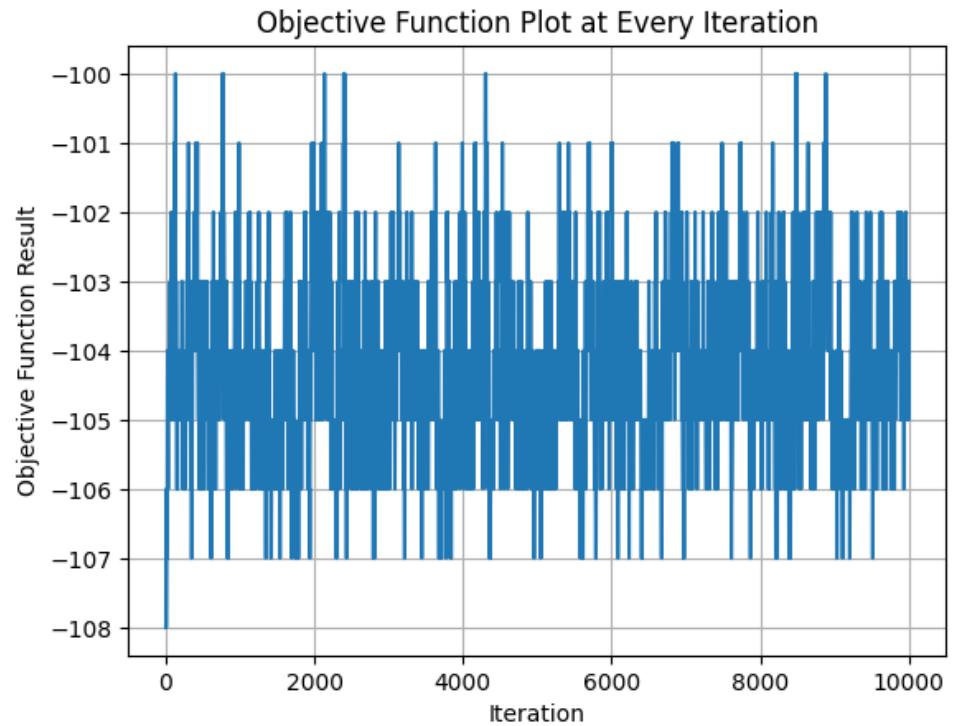
```
Iterations: 10000
Duration: 15.108501 seconds
```

5. Iterasi 2, populasi 2, dapat dituliskan ke dalam bentuk

- Jumlah populasi: 8
- Iterasi maksimal: 10000
- State awal & akhir



- Plot Objektif vs Iterasi

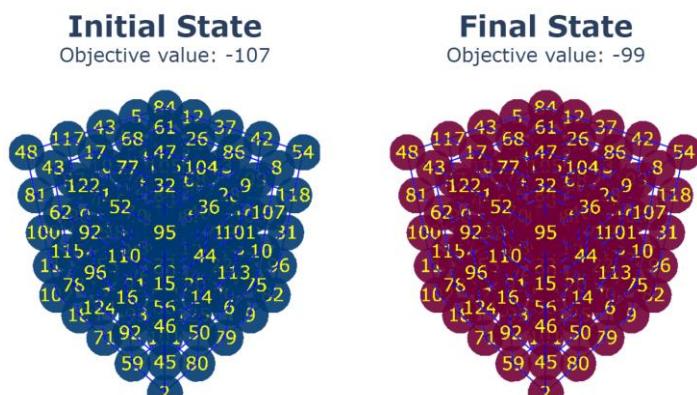


- Durasi & jumlah iterasi

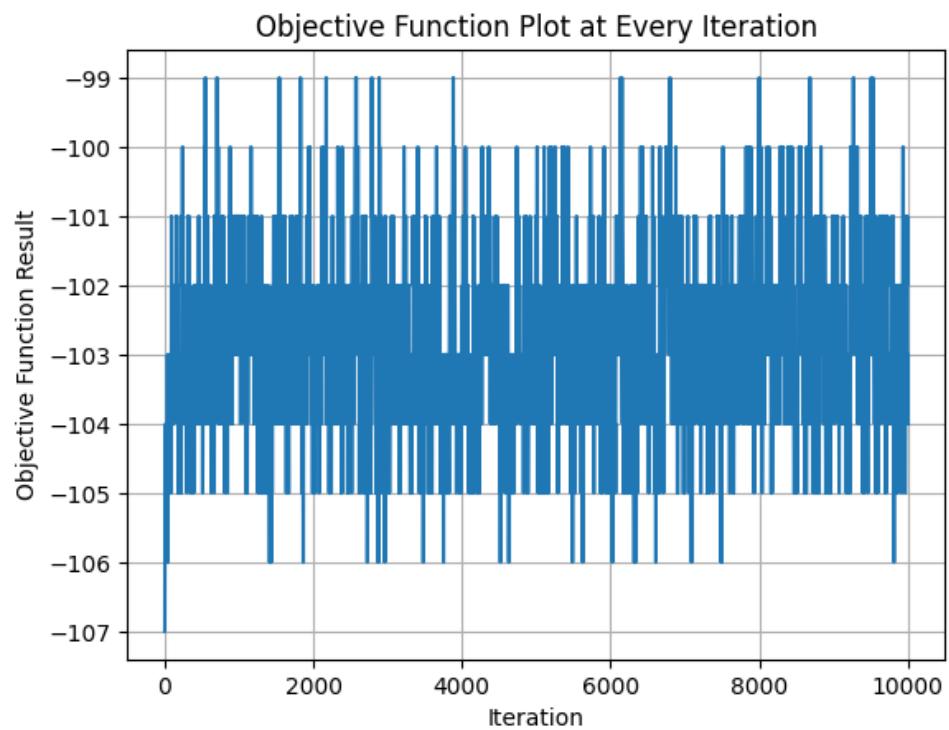
```
Iterations: 10000
Duration: 25.796117 seconds
```

6. Iterasi 2, populasi 3, dapat dituliskan ke dalam bentuk

- Jumlah populasi: 16
- Iterasi maksimal: 10000
- State awal & akhir



- Plot Objektif vs Iterasi

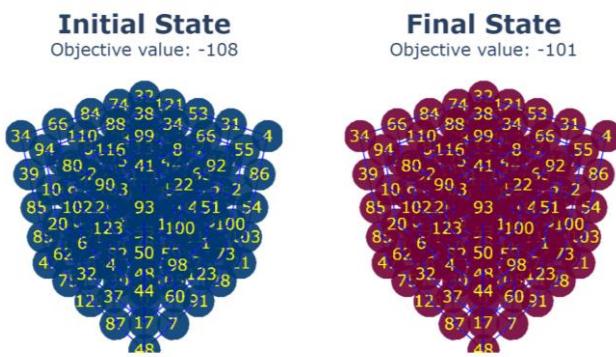


- Durasi & jumlah iterasi

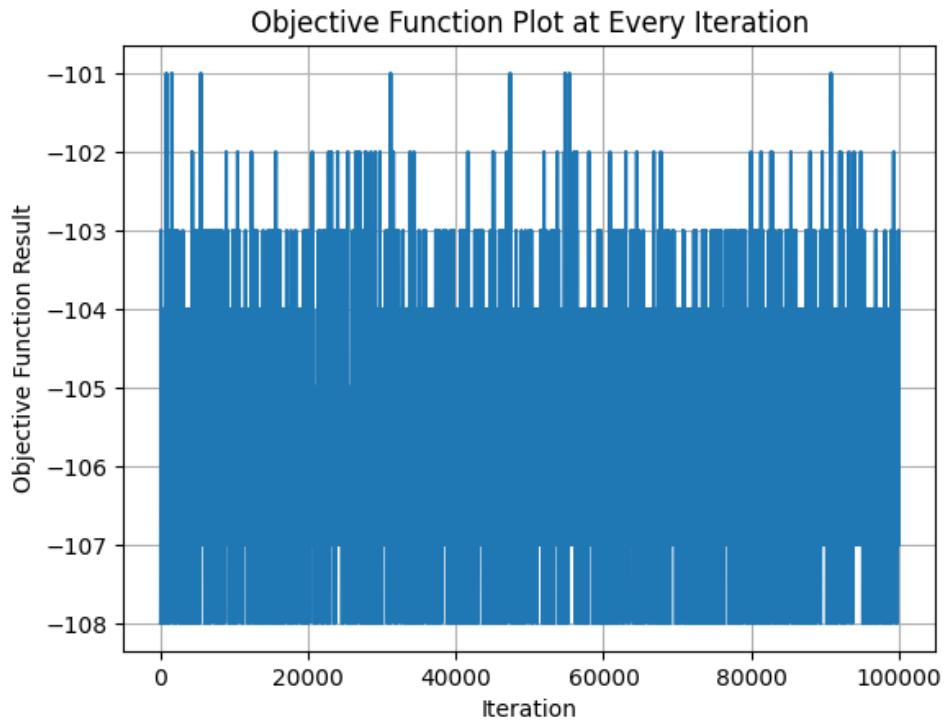
```
Iterations: 10000
Duration: 55.482302 seconds
```

7. Iterasi 3, populasi 1, dapat dituliskan ke dalam bentuk

- Jumlah populasi: 4
- Iterasi maksimal: 100000
- State awal & akhir



- Plot Objektif vs Iterasi:

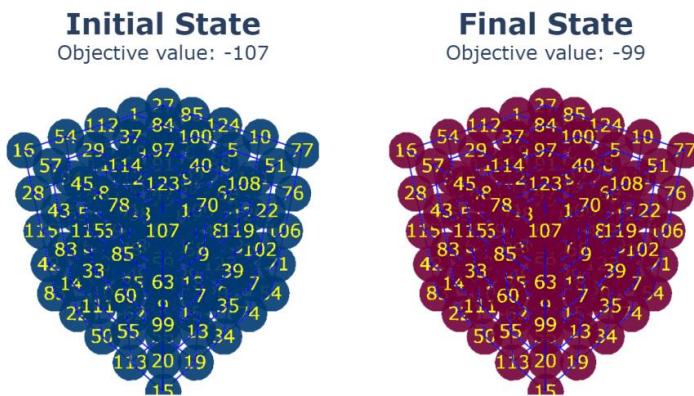


- Durasi & jumlah iterasi

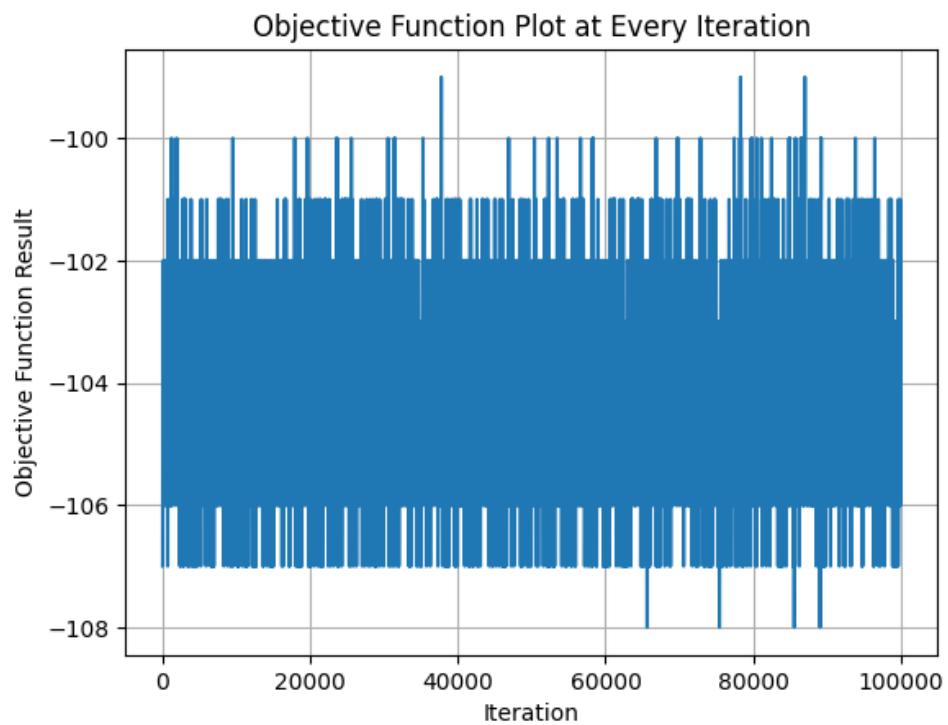
```
Iterations: 100000
Duration: 135.062885 seconds
```

8. Iterasi 3, populasi 2, dapat dituliskan ke dalam bentuk7

- Jumlah populasi: 8
- Iterasi maksimal: 100000
- State awal & akhir



- Plot Objektif vs Iterasi

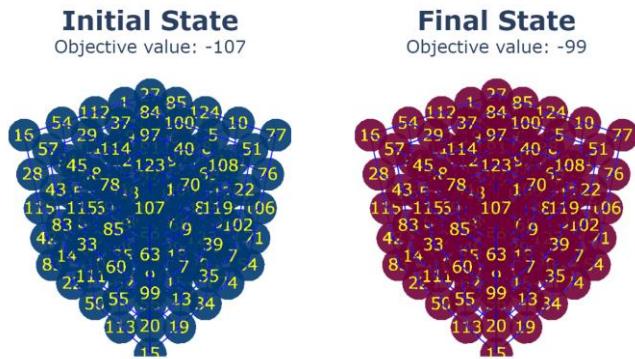


- Durasi & jumlah iterasi

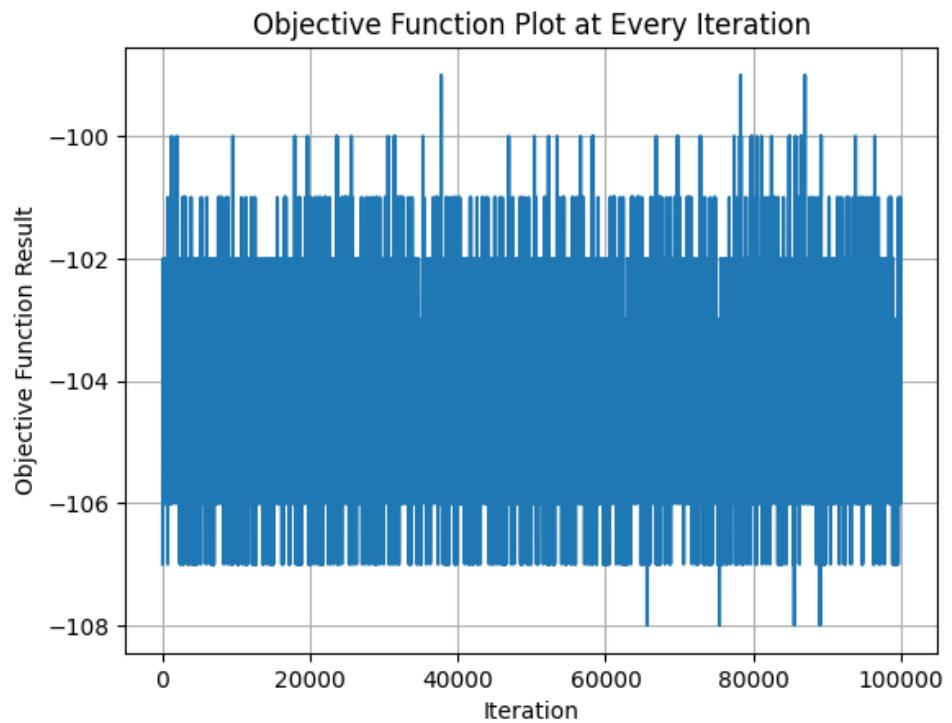
```
Iterations: 100000
Duration: 309.713641 seconds
```

9. Iterasi 3, populasi 3, dapat dituliskan ke dalam bentuk

- Jumlah populasi: 16
- Iterasi maksimal: 100000
- State awal & akhir



- Plot Objektif vs Iterasi

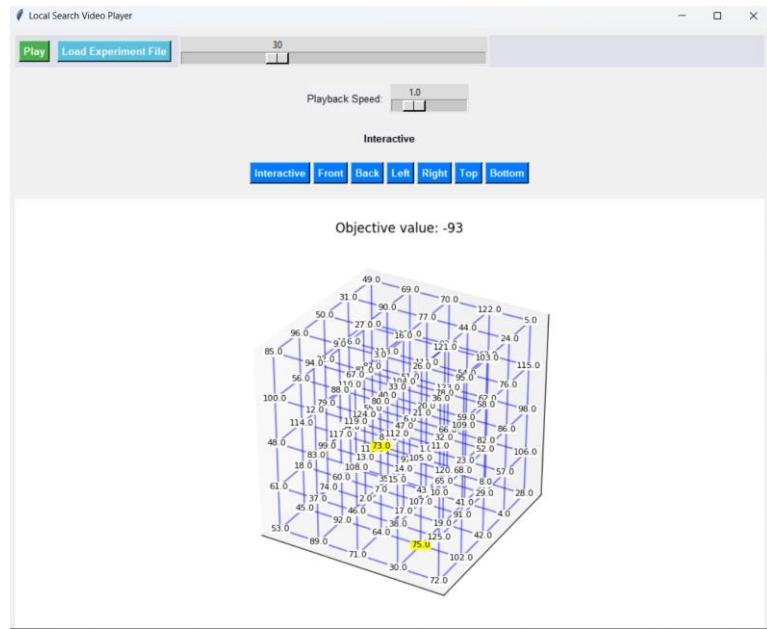


- Durasi & jumlah iterasi

```
Iterations: 100000
Duration: 309.713641 seconds
```

2.4 Video Player Hill-Climbing

Pada laporan ini, dikembangkan juga video player yang dapat menunjukkan hasil setiap state pada local search. Dimana ditunjukkan visualisasi state dan juga animasi perubahan state pada kubus. Ditunjukkan sesuai pada Gambar 2.3.



Gambar 2.3 Video Player

Format file yang digunakan ditunjukkan sebagai berikut:

```

Iteration 1:-108:None:None

22.0 4.0 28.0 30.0 72.0
45.0 92.0 47.0 12.0 64.0
74.0 2.0 17.0 19.0 6.0
25.0 35.0 43.0 41.0 89.0
118.0 97.0 120.0 8.0 71.0

61.0 37.0 46.0 103.0 125.0
18.0 60.0 7.0 107.0 91.0
99.0 13.0 14.0 65.0 29.0
34.0 84.0 1.0 23.0 57.0
55.0 42.0 66.0 82.0 106.0

48.0 83.0 108.0 15.0 10.0
114.0 117.0 73.0 105.0 68.0
79.0 124.0 102.0 32.0 52.0
110.0 40.0 70.0 59.0 86.0
87.0 51.0 123.0 62.0 98.0

100.0 75.0 119.0 112.0 11.0
56.0 88.0 80.0 21.0 109.0
53.0 81.0 104.0 67.0 58.0
116.0 113.0 111.0 54.0 76.0
39.0 101.0 93.0 63.0 115.0

85.0 94.0 78.0 33.0 36.0
96.0 9.0 3.0 26.0 95.0
50.0 27.0 16.0 121.0 38.0
31.0 90.0 77.0 44.0 24.0
49.0 69.0 20.0 122.0 5.0

Iteration 2:-106:0-2-4:1-4-1

22.0 4.0 28.0 30.0 72.0
45.0 92.0 47.0 12.0 64.0
74.0 2.0 17.0 19.0 42.0
25.0 35.0 43.0 41.0 89.0
118.0 97.0 120.0 8.0 71.0

61.0 37.0 46.0 103.0 125.0
18.0 60.0 7.0 107.0 91.0
99.0 13.0 14.0 65.0 29.0
34.0 84.0 1.0 23.0 57.0
55.0 6.0 66.0 82.0 106.0

48.0 83.0 108.0 15.0 10.0
114.0 117.0 73.0 105.0 68.0
79.0 124.0 102.0 32.0 52.0
110.0 40.0 70.0 59.0 86.0
87.0 51.0 123.0 62.0 98.0

100.0 75.0 119.0 112.0 11.0
56.0 88.0 80.0 21.0 109.0
53.0 81.0 104.0 67.0 58.0
116.0 113.0 111.0 54.0 76.0
39.0 101.0 93.0 63.0 115.0

85.0 94.0 78.0 33.0 36.0
96.0 9.0 3.0 26.0 95.0
50.0 27.0 16.0 121.0 38.0
31.0 90.0 77.0 44.0 24.0
49.0 69.0 20.0 122.0 5.0

```

Iteration [i]:[Nilai objective value]:[Posisi cell, frame-row-column pertama]:[Posisi cell, frame-row-column kedua] yang disimpan dalam file .txt

2.5 Analisis

Dari ketiga kasus awal yang dijalankan, algoritma *steepest ascent hill-climbing* mencapai nilai objektif antara -68 hingga -70. Hal ini cukup memenuhi ekspektasi dimana algoritma ini rentan mencapai lokal optima dan masih cukup jauh dari *global optimum*.

Di sisi lain, algoritma *hill-climbing with sideways move* mencapai nilai objektif antara -68 hingga -70 pula, sama dengan algoritma *steepest ascent hill-climbing*. Hal ini mengartikan bahwa lokal optima yang dicapai sebelumnya adalah sebuah “*shoulder*”, dan memungkinkan terjadinya gerakan “bulak-balik” pada *shoulder* tersebut. Hasil yang didapat masih cukup jauh dari *global optimum*.

Selain itu, pada algoritma *stochastic hill climbing*, dicapai nilai objektif antara -86 hingga -93. Hal ini dapat disebabkan karena terlalu banyak *neighbour* yang ada dari *state* tersebut sehingga dalam jumlah iterasi maksimum yang ditentukan, belum dapat ditemukan dan masih jauh dari *global maksimum*.

Kemudian, pada algoritma *random restart hill climbing*, nilai objektif terbesar mencapai antara -63 hingga -64. Hal ini dapat terjadi karena ia merupakan *steepest ascent hill climbing* yang dilakukan beberapa kali dan dengan fitur *restart*, ia bisa berpindah ke *initial state* lain yang dapat mencapai nilai objektif yang lebih baik. Namun, dari nilai yang didapat, belum bisa didapatkan dan masih cukup jauh dari *global optimum*.

Selanjutnya, pada algoritma *simulated annealing*, dicapai nilai objektif antara -107 hingga -104. Hal ini mungkin saja disebabkan karena algoritma ini memungkinkan untuk pindah state ke state yang lebih buruk dengan probabilitas.

Terakhir, untuk algoritma genetik (GA), dicapai nilai objektif antara -97 hingga -103. Hal ini terjadi karena implementasi GA melakukan *crossover* dan *mutation* pada parents dan menghasilkan offspring, tidak ada algoritma yang membandingkan nilai objektif dari offspring terhadap parents, sehingga dapat digenerate offspring yang lebih buruk maupun lebih baik yang tentunya akan mempengaruhi hasil akhir nilai objektif.

Dari hasil yang didapatkan, terlihat bahwa algoritma *steepest-ascent-hill-climbing-based* (*steepest-ascent hill climbing*, *hill climbing with sideway move*, dan *random restart hill-climbing*) memberikan hasil yang lebih baik daripada algoritma

lainnya. Hal ini dapat dilihat dari nilai objektif *state* akhir yang diperoleh. Hal ini dapat terjadi karena terlalu banyak *neighbor* yang mungkin sehingga algoritma dengan pembangkit berbasis *random* “lama” menemukan solusi. Selain itu, tidak ada algoritma yang dapat mencapai nilai *global optimum* karena terlalu banyak *local optima*.

Algoritma yang memberikan durasi paling cepat ialah *algoritma steepest ascent hill-climbing* karena pada saat sudah menemukan puncaknya maka proses pencarian akan selesai, sedangkan pada *sideways move hill-climbing* memberikan hasil dengan durasi yang lebih lama, karena terdapat waktu pada saat solusi yang didapatkan datar sehingga waktu yang diperlukan lebih lama. Algoritma *random restart hill-climbing* memberikan hasil dengan durasi yang lebih lama pula, namun hal ini terjadi karena ia hanyalah *steepest-ascent hill climbing* yang dilakukan berkali-kali, sehingga sesuai ekspektasi. Algoritma *stochastic hill-climbing* memberikan hasil durasi yang lebih lama pula dikarenakan jumlah iterasi yang jauh di atas iterasi yang dilakukan *steepest ascent* maupun *with-sideway-move hill-climbing*. Hal demikian juga berlaku pada algoritma *simulated annealing* dan *genetic algorithm*, dimana didapat durasi untuk melakukan pencarian dengan waktu yang lebih lama karena terdapat lebih banyak iterasi yang dilakukan. Selain itu, penggunaan library **numba**, juga mempercepat proses pencarian *best neighbor state* pada algoritma *steepest ascent* dan *with-sideway-move hill-climbing*.

Algoritma steepest ascent hill-climbing memiliki kekonsistennan hasil akhir yang cukup baik pada setiap eksperimentnya, dengan deviasi nilai akhir setiap eksperimen hanya berkisar ± 1 . Algoritma sideways hill-climbing memiliki kekonsistennan hasil akhir yang cukup baik pada setiap eksperimentnya, dengan deviasi nilai akhir setiap eksperimen hanya berkisar ± 1 . Algoritma random restart hill-climbing memiliki kekonsistennan hasil akhir yang cukup baik (paling baik dibanding lainnya) pada setiap eksperimentnya, dengan deviasi nilai akhir setiap eksperimen hanya berkisar ± 0.5 . Algoritma stochastic hill-climbing memiliki kekonsistennan hasil akhir yang cukup baik namun tidak sebaik lainnya pada setiap eksperimentnya, dengan deviasi nilai akhir setiap eksperimen hanya berkisar ± 3 . Algoritma simulated annealing memiliki kekonsistennan hasil akhir yang cukup baik pada setiap eksperimentnya, dengan deviasi nilai akhir setiap eksperimen hanya

berkisar ± 1.5 . Algoritma genetic algorithm memiliki kekonsistenan hasil akhir yang cukup baik pada setiap eksperimennya, dengan deviasi nilai akhir setiap eksperimen hanya berkisar ± 1 .

Pengaruh banyaknya iterasi dan jumlah populasi terhadap hasil akhir pencarian *genetic algorithm* tidak terlalu signifikan. Hal ini dapat dilihat pada plot-plot nilai objektif vs. iterasi dimana, nilai objektif hanya berosilasi sekitar -102 hingga -105 pada seluruh kombinasi jumlah populasi dan banyak iterasi. Hal ini terjadi karena proses crossover yang dilakukan tidak terlalu menghasilkan *offspring* yang lebih baik secara signifikan. Dapat dilihat dari beberapa plot bahwa terdapat kasus dimana *parent* sudah memiliki nilai objektif lebih tinggi, namun *offspring* yang dihasilkan memiliki nilai objektif lebih rendah, dan, karena tidak ada *heuristik* yang diterapkan, maka *offspring* tersebut tetap dipilih.

BAB 3

KESIMPULAN DAN SARAN

Pada laporan ini telah dilakukan implementasi algoritma *local search* dalam penyelesaian *diagonal magic cube*. Hasil menunjukkan bahwa terdapat keunikan metode dari masing-masing algoritma dalam mencari solusi global optimal dari permasalahan magic cube tersebut. Untuk algoritma *local search* yang memberikan solusi terbaik dimana merupakan solusi yang paling mendekati nilai global optima adalah algoritma *random restart hill-climbing* dengan nilai objektif yaitu berkisar pada -63 hingga -64. Lalu, untuk algoritma dengan durasi yang paling singkat dalam melakukan pencarian solusi global optima adalah *simulated annealing* dengan durasi rata-ratanya pada 1.0056 seconds.

Berikut saran terdapat eksperimen yang dilakukan untuk perkembangan selanjutnya:

1. Menggunakan teknik heuristic untuk memilih bests successors, karena banyak best successors dengan nilai objektif yang sama.
2. Menggunakan variasi lain pada GA seperti variasi crossover, mutation, dst.
3. Menggunakan teknik heuristic dalam pemilihan offspring dari GA.
4. Menggunakan paralel computing dalam menggenerate state.

PEMBAGIAN TUGAS TIAP ANGGOTA KELOMPOK

Tabel 1: Kontribusi Angota

NIM	Nama	Kontribusi
13121140	Muhammad Fadli Alfarizi	<ul style="list-style-type: none">• Membuat <i>class Utility</i> dan <i>State</i>.• Membuat program <i>video player</i>• Membantu analisis• Membantu formatting
13121142	Roby Pratama Sitepu	<ul style="list-style-type: none">• Membuat analisis laporan• Membuat formating laporan• Membuat kesimpulan
13621034	Muhammad Arviano Yuono	<ul style="list-style-type: none">• Implementasi algoritma <i>steepest-ascent hill climbing</i>.• Implementasi algoritma <i>hill climbing with sideways move</i>.• Implementasi algoritma <i>random restart hill climbing</i>.• Membuat <i>class BaseResult</i> dan <i>BaseAlgorithm</i>.• Membuat program visualisasi.
13621060	Hafizh Renanto Akhmad	<ul style="list-style-type: none">• Implementasi algoritma <i>stochastic hill climbing</i>.• Implementasi algoritma <i>simulated annealing</i>.• Implementasi algoritma <i>genetic algorithm</i>.• Membantu sebagai <i>editor</i> untuk Subbab 2.2.• Membantu analisis

REFERENSI

- [1] W. Trump, "The Successful Search for the Smallest Perfect Magic Cube," 25 February 2005. [Online]. Available: <https://www.trump.de/magic-squares/magic-cubes/cubes-1.html>.
- [2] E. W. Weisstein, "Magic Cube," 26 May 1999. [Online]. Available: <https://archive.lib.msu.edu/crcmath/math/math/m/m022.htm>.
- [3] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 3rd ed., Harlow, Essex: Pearson, 2009.
- [4] P. Larranaga, C. Kuijpers, R. H. Murga, I. Inza and S. Dizdarevic, "Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators," *Artificial Intelligence Review*, vol. 13, pp. 129-170, 1999.
- [5] A. Karazmoodeh, "Mutations in genetic algorithms," 31 December 2023. [Online]. Available: <https://www.linkedin.com/pulse/mutations-genetic-algorithms-ali-karazmoodeh-u94pf/>. [Accessed 1 October 2024].