

IF3070 DASAR INTELEGENSI ARTIFISIAL
TUGAS BESAR 1

Pencarian Solusi *Diagonal Magic Cube* dengan *Local Search*



Disusun oleh:

| | |
|---------------------------|----------|
| Thalita Zahra Sutejo | 18222023 |
| Irfan Musthofa | 18222056 |
| Eleanor Cordelia | 18222059 |
| Muhammad Faiz Atharrahman | 18222063 |

PRODI SISTEM DAN TEKNOLOGI INFORMASI
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

2024

DAFTAR ISI

| | |
|---|-----------|
| DAFTAR GAMBAR..... | 3 |
| BAB I | |
| DESKRIPSI persoalan..... | 1 |
| BAB II | |
| PEMBAHASAN..... | 4 |
| 2.1 Pemilihan Objective Function..... | 4 |
| 2.1.1 Objective Function Pertama..... | 4 |
| 2.1.2 Objective Function Kedua..... | 5 |
| 2.1.3 Kesimpulan Pemilihan Objective Function..... | 7 |
| 2.2 Penjelasan Implementasi Algoritma Local Search..... | 8 |
| 2.2.1 Steepest Ascent Hill-Climbing..... | 8 |
| 2.2.2 Hill-Climbing with Sideways Move..... | 14 |
| 2.2.3 Random Restart Hill-Climbing..... | 21 |
| 2.2.4 Stochastic Hill-Climbing..... | 30 |
| 2.2.5 Simulated Annealing..... | 37 |
| 2.2.6 Genetic Algorithm..... | 45 |
| 2.3 Hasil Eksperimen dan Analisis..... | 56 |
| 2.3.1 Steepest Ascent Hill-Climbing..... | 56 |
| 2.3.2 Hill-Climbing with Sideways Move..... | 59 |
| 2.3.3 Random Restart Hill-Climbing..... | 62 |
| 2.3.4 Stochastic Hill-Climbing..... | 65 |
| 2.3.5 Simulated Annealing..... | 68 |
| 2.3.6 Genetic Algorithm..... | 71 |
| BAB III | |
| KESIMPULAN DAN SARAN..... | 81 |
| 3.1 Kesimpulan..... | 81 |
| 3.2 Saran..... | 81 |
| BAB IV | |
| PEMBAGIAN TUGAS TIAP ANGGOTA KELOMPOK..... | 83 |
| REFERENSI..... | 85 |

DAFTAR GAMBAR

| | |
|--|----|
| Gambar 2.1 Testing 1 Algoritma Steepest Ascent Hill-Climbing..... | 56 |
| Gambar 2.2 Testing 2 Algoritma Steepest Ascent Hill-Climbing..... | 57 |
| Gambar 2.3 Testing 3 Algoritma Steepest Ascent Hill-Climbing..... | 57 |
| Gambar 2.4 Visualisasi Initial State Steepest Ascent Hill-Climbing..... | 58 |
| Gambar 2.5 Visualisasi Final State Steepest Ascent Hill-Climbing..... | 58 |
| Gambar 2.6 Testing 1 Algoritma Hill-Climbing with Sideways Move..... | 59 |
| Gambar 2.7 Testing 2 Algoritma Hill-Climbing with Sideways Move..... | 60 |
| Gambar 2.8 Testing 3 Algoritma Hill-Climbing with Sideways Move..... | 60 |
| Gambar 2.9 Visualisasi Initial State Hill-Climbing with Sideways Move..... | 61 |
| Gambar 2.10 Visualisasi Final State Hill-Climbing with Sideways Move..... | 61 |
| Gambar 2.11 Testing 1 Algoritma Random Restart Hill-Climbing..... | 62 |
| Gambar 2.12 Testing 2 Algoritma Random Restart Hill-Climbing..... | 63 |
| Gambar 2.13 Testing 3 Algoritma Random Restart Hill-Climbing..... | 63 |
| Gambar 2.14 Visualisasi Initial State Random Restart Hill-Climbing..... | 64 |
| Gambar 2.15 Visualisasi Final State Random Restart Hill-Climbing..... | 64 |
| Gambar 2.16 Testing 1 Algoritma Stochastic Hill-Climbing..... | 65 |
| Gambar 2.17 Testing 2 Algoritma Stochastic Hill-Climbing..... | 66 |
| Gambar 2.18 Testing 3 Algoritma Stochastic Hill-Climbing..... | 66 |
| Gambar 2.19 Visualisasi Initial State Stochastic Hill-Climbing..... | 67 |
| Gambar 2.20 Visualisasi Final State Stochastic Hill-Climbing..... | 67 |
| Gambar 2.21 Testing 1 Algoritma Simulated Annealing..... | 68 |
| Gambar 2.22 Testing 1 Algoritma Simulated Annealing..... | 68 |
| Gambar 2.23 Testing 2 Algoritma Simulated Annealing..... | 68 |
| Gambar 2.24 Testing 2 Algoritma Simulated Annealing..... | 68 |
| Gambar 2.25 Testing 3 Algoritma Simulated Annealing..... | 69 |
| Gambar 2.26 Testing 3 Algoritma Simulated Annealing..... | 69 |
| Gambar 2.27 Visualisasi Initial State Simulated Annealing..... | 70 |
| Gambar 2.28 Visualisasi Final State Simulated Annealing..... | 70 |
| Gambar 2.29 Testing 1 Plot Objective Value Genetic Algorithm..... | 71 |
| Gambar 2.30 Testing 1 Plot Avg Objective Value Genetic Algorithm..... | 71 |
| Gambar 2.31 Testing 2 Plot Objective Value Genetic Algorithm..... | 71 |
| Gambar 2.32 Testing 2 Plot Avg Objective Value Genetic Algorithm..... | 71 |
| Gambar 2.33 Testing 3 Plot Objective Value Genetic Algorithm..... | 72 |
| Gambar 2.34 Testing 3 Plot Avg Objective Value Genetic Algorithm..... | 72 |
| Gambar 2.35 Testing 4 Plot Objective Value Genetic Algorithm..... | 72 |
| Gambar 2.36 Testing 4 Plot Avg Objective Value Genetic Algorithm..... | 72 |

| | |
|---|----|
| Gambar 2.37 Testing 5 Plot Objective Value Genetic Algorithm..... | 72 |
| Gambar 2.38 Testing 5 Plot Avg Objective Value Genetic Algorithm..... | 72 |
| Gambar 2.39 Testing 6 Plot Objective Value Genetic Algorithm..... | 73 |
| Gambar 2.40 Testing 6 Plot Avg Objective Value Genetic Algorithm..... | 73 |
| Gambar 2.41 Visualisasi Initial State Genetic Algorithm..... | 74 |
| Gambar 2.42 Visualisasi Final State Genetic Algorithm..... | 74 |

BAB I

DESKRIPSI PERSOALAN

Diagonal magic cube adalah sebuah kubus berukuran $n \times n \times n$ yang terdiri dari angka 1 hingga n^3 tanpa pengulangan, dengan n adalah panjang sisi kubus tersebut. Angka-angka pada tersusun sedemikian rupa sehingga properti-properti di bawah ini terpenuhi.

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

Dalam permasalahan ini, kubus berukuran $5 \times 5 \times 5$ diisi dengan angka-angka dari 1 hingga 5^3 secara acak dan *distinct*, dengan tujuan menyusun angka-angka tersebut agar total penjumlahan elemen pada setiap baris, kolom, pilar, dan diagonal sama dengan nilai *magic constant* 315.

Magic number (M) dapat dicari dengan membandingkan nilai penjumlahan semua angka dari kubus tersebut dari perhitungan jumlah M yang mungkin dan penjumlahan angka 1 hingga n^3 . Nilai penjumlahan semua angka dari kubus berdasarkan perhitungan jumlah M yang mungkin adalah sebagai berikut:

$$M \times n^2 = M \times n \times n$$

Nilai penjumlahan semua angka dari kubus berdasarkan penjumlahan angka 1 hingga n^3 adalah sebagai berikut.

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

Dari kedua ekspresi tersebut, gunakan persamaan untuk mencari nilai M .

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

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

Dalam menyusun *diagonal magic cube* yang optimal, pendekatan menggunakan *objective function* diperlukan untuk meminimalkan perbedaan antara jumlah elemen pada baris, kolom, pilar, dan diagonal terhadap *magic constant* 315, yang akan menjadi indikator kualitas solusi. Untuk mencapai konfigurasi optimal yang memenuhi kriteria *magic constant* pada semua aspek tersebut, digunakan algoritma *local search*. Algoritma ini dianggap lebih cepat dibandingkan metode *complete search* karena *local search* fokus pada perbaikan bertahap dari *state* saat ini ke *state* selanjutnya, tetapi berpotensi untuk *stuck* di *local optima*. Operasi yang dilakukan dalam *local search* hanyalah menukar posisi dua angka di dalam kubus, dimana angka-angka yang ditukar tidak harus berdekatan, yang secara matematis dinotasikan sebagai operasi *swap* antara dua angka pada koordinat yang berbeda.

Berikut adalah aturan aksi *swap* dalam notasi matematika sederhana dengan asumsi x, y, z, a, b , dan c mencerminkan posisi angka dalam model koordinat.

$$\begin{aligned} & \forall x, y, z, a, b, c \in [1, 125] \\ & (x, y, z) \neq (a, b, c) \\ & Swap(A_{x,y,z}, B_{a,b,c}) = A_{a,b,c}, B_{x,y,z} \end{aligned}$$

Penentuan dua angka yang akan ditukar dalam aksi ini didasari pada evaluasi *state* berdasarkan seberapa dekat total penjumlahan elemen pada baris, kolom, pilar, dan diagonal terhadap *magic constant* atau pelanggaran penjumlahan yang tidak sesuai dengan *magic constant*. Struktur solusi ini juga mencakup implementasi beberapa kelas atau fungsi yang mendukung proses pencarian solusi optimal dengan algoritma *local search*, memberikan

gambaran kerangka kerja kode yang digunakan untuk menyelesaikan permasalahan *diagonal magic cube* berukuran $5 \times 5 \times 5$ ini.

BAB II

PEMBAHASAN

2.1 Pemilihan *Objective Function*

2.1.1 *Objective Function* Pertama

Objective function dalam AI pada permasalahan ini digunakan untuk mengevaluasi hasil algoritma atau mengukur seberapa baik kondisi *state* dibandingkan dengan kondisi sempurna atau *global maximum*. Definisi *state* sempurna di sini adalah ketika *state* mencapai *perfect diagonal magic cube* dimana semua hasil penjumlahan pada setiap baris, kolom, pilar, diagonal bidang, dan diagonal ruang memiliki hasil 315. Model AI akan melakukan aksinya terus-menerus sesuai dengan algoritma yang digunakan dengan melakukan *swap* dua angka berbeda hingga mencapai nilai maksimum dari *objective function*.

Pada kondisi permasalahan *diagonal magic cube* berorde lima ini, terdapat 25 baris, 25 kolom, 25 tiang, 30 diagonal bidang, dan 4 diagonal ruang yang akan dihitung hasil penjumlahahannya. Sehingga, akan dilakukan 109 kali perhitungan menggunakan *objective function*. Pendekatan yang akan digunakan pada persoalan ini adalah *cost-based value*, dimana semakin tinggi *value*-nya maka semakin buruk atau jauh dari kondisi *perfect diagonal magic cube*, sedangkan kondisi *global maximum* ditunjukkan dengan nilai *value* = 0.

Objective function akan menerima parameter *initialState* dan akan mengembalikan *Value*. Perhitungan diawali dengan menghitung hasil penjumlahan semua angka pada setiap baris. Setiap baris yang sudah dijumlahkan ini akan dibandingkan dengan 315, selisih angkanya menjadi *cost-value* dan ditampung di variabel *Value*. Penjumlahan dan perbandingan ini dilakukan pada masing-masing 25 baris, 25 kolom, 25 tiang, 30 diagonal bidang, dan 4 diagonal ruang lainnya. Sehingga, semakin tinggi nilai *Value* dari *state*, maka semakin jauh *state* dari *global maximum state*.

Berikut adalah *simplified pseudocode* dari *objective function* yang digunakan.

```

function objectiveCheck(State) returns Value
    Cube ← State
    Value ← 0

    {state or cube itself consists of 3D Array of Integer [1, 125] with
    predetermined rows(), cols(), pillars(), diagonals(), trigonals()}

    {check all rows}
    for i = 1 to 25 do
        Value ← Value + abs(315 - sum(Cube.rows[i]))

    {check all columns}
    for j = 1 to 25 do
        Value ← Value + abs(315 - sum(Cube.cols[j]))

    {check all pillars}
    for k = 1 to 25 do
        Value ← Value + abs(315 - sum(Cube.pillars[k]))

    {check all diagonals}
    for l = 1 to 30 do
        Value ← Value + abs(315 - sum(Cube.diagonals[l]))

    {check all space diagonals}
    for m = 1 to 4 do
        Value ← Value + abs(315 - sum(Cube.trigonals[m]))

return Value

```

Berikut adalah notasi matematika sederhana dari *objective function* yang digunakan. Diasumsikan R_i adalah hasil penjumlahan semua angka pada baris ke- i dan berlaku dengan pola yang sama untuk C_i sebagai kolom, P_i sebagai tiang, D_i sebagai diagonal bidang, dan S_i sebagai diagonal ruang.

$$V = \sum_{i=1}^{25} |315 - R_i| + \sum_{i=1}^{25} |315 - C_i| + \sum_{i=1}^{25} |315 - P_i| + \sum_{i=1}^{30} |315 - D_i| + \sum_{i=1}^4 |315 - S_i|$$

2.1.2 *Objective Function* Kedua

Pada permasalahan *magic cube*, masalah terpecahkan ketika semua penjumlahan angka pada baris, kolom, tiang, diagonal sisi, dan diagonal ruang dalam sebuah kubus sama dengan *magic number*.

Fungsi kedua ini menghitung banyak penjumlahan angka pada, baris, kolom, tiang, diagonal sisi, dan diagonal ruang yang tidak sama dengan *magic number*. Banyak penjumlahan angka pada baris, kolom, tiang, diagonal sisi, dan diagonal ruang yang mungkin pada kubus 5x5x5 adalah

$$jumlah\ baris = 5\ kolom \times 5\ tiang = 25$$

$$jumlah\ kolom = 5\ baris \times 5\ tiang = 25$$

$$jumlah\ tiang = 5\ baris \times 5\ kolom = 25$$

$$jumlah\ diagonal\ sisi = 3\ sisi \times 5\ tiang \times 2\ diagonal\ per\ 1\ sisi = 30$$

$$jumlah\ diagonal\ ruang = 8\ sudut \times 1\ diagonal\ ruang\ per\ 2\ sudut = 4$$

$$total\ jumlah\ penjumlahan = 25 + 25 + 25 + 30 + 4 = 109$$

Fungsi objektif akan meretur nilai 0 bila kubus telah mencapai kondisi *perfect cube*. Fungsi objektif tersebut dapat direpresentasikan dalam *pseudocode* sebagai berikut:

```
function objective_function(cube_state: 5x5x5 Matrix of int)
    → objective_value: int
    n ← 5 # Cube Order
    # Sum of Row, Column, Depth
    row_sum ← [sum([cube_state[i][j][k] for k in range(n)])
               for j in range(n) for i in range(n)]
    column_sum ← [sum([cube_state[i][j][k] for j in range(n)])
                  for k in range(n) for i in range(n)]
    depth_sum ← [sum([cube_state[i][j][k] for i in range(n)])
                  for k in range(n) for j in range(n)]

    # Sum of Diagonals
    shift_x_cube ← [[[cube_state[i][j][k] for i in
                      range(n-1,-1,-1)] for j in range(n)] for k in range(n)]
    shift_y_cube ← [[cube_state[i][j] for i in
                     range(n-1,-1,-1)] for j in range(n)]
    diagonal_sum ← []
    for cube in (cube_state, shift_x_cube, shift_y_cube):
        for face in cube:
```

```

first_diagonal_sum = sum([face[j][j] for j in
                         range(n)])
second_diagonal_sum = sum([face[j][k] for j, k in
                           zip(range(n), range(n-1,-1,-1))])
diagonal_sum += [first_diagonal_sum,
                  second_diagonal_sum]

# Sum of Space Diagonals
space_diagonal_sum ← []
space_diagonal_sum += [sum([cube_state[i][i][i] for i in
                           range(n)])]
space_diagonal_sum += [sum([cube_state[i][j][j] for i, j in
                           zip(range(n), range(n-1,-1,-1))])]
space_diagonal_sum += [sum([cube_state[i][i][j] for i, j in
                           zip(range(n), range(n-1,-1,-1))])]
space_diagonal_sum += [sum([cube_state[i][j][i] for i, j in
                           zip(range(n), range(n-1,-1,-1))])]

→ 109 - (row_sum + column_sum + depth_sum + diagonal_sum +
           space_diagonal_sum).count(magic_number)

```

2.1.3 Kesimpulan Pemilihan *Objective Function*

Berikut adalah implementasi kedua *objective function* pada bahasa pemrograman Go.

```

func sum_of_magic_sum_differences(cube Cube) int {
    sum := 0
    all_cube_sum := get_all_cube_sum(cube)
    max_sum := len(all_cube_sum)
    for i := range max_sum {
        sum += int(math.Abs(float64(magic_constant -
all_cube_sum[i])))
    }
    return sum
}

func violated_magic_sum_count(cube Cube) int {
    count := 0
    all_cube_sum := get_all_cube_sum(cube)
    max_sum := len(all_cube_sum)
    for i := range max_sum {
        if all_cube_sum[i] == magic_constant {
            count += 1
        }
    }
    return max_sum - count
}

```

Kedua fungsi tersebut dapat digunakan pada algoritma *searching* yang akan dijabarkan di bagian berikutnya karena setiap fungsi algoritma menerima parameter *objective function* yang dapat digunakan. Fungsi objektif `func violated_magic_sum_count(cube Cube) int` dipilih karena perbedaan satu selisih saja dapat mempengaruhi nilai fungsi objektif yang dihasilkan dengan perhitungan yang lebih sedikit dan tidak harus menyimpan angka yang besar seperti melakukan mutlak pada selisih *value* dengan *magic number*, sehingga fungsi objektif ini lebih optimal.

2.2 Penjelasan Implementasi Algoritma *Local Search*

2.2.1 *Steepest Ascent Hill-Climbing*

a. Deskripsi *Steepest Ascent Hill-Climbing*

Algoritma *Steepest Ascent Hill-Climbing Search* berawal dari kubus dengan semua posisi angka tersusun secara acak sebagai *initial state*, lalu membangkitkan dan mengevaluasi semua *successor* dengan *objective function* dan kemudian memilih *successor* dengan *value* terbaik yang mendekati nol sebagai *neighbor*. Ketika *neighbor* memiliki *value* lebih baik dari *current state*, maka *current state* di-*assign* oleh *neighbor*. Jika pada pencarian tidak ditemukan *neighbor* dengan *value* yang lebih baik daripada atau sama dengan *current state*, maka algoritma berhenti.

b. *Source code Steepest Ascent Hill-Climbing dalam bahasa pemrograman Go*

```
func steepest_ascent_hill_climbing(cube Cube,
objective_function ObjectiveFunction)
LocalSearchResultSteepestAscent {
    timeStart := time.Now()
    current_state := copy_cube(cube)
    iteration := 0 // iteration counter
    var local_search_result LocalSearchResultSteepestAscent
    objective_function_logs := []int{}
    var swap_pair SwapPair
    swap_logs := []SwapPair{}

    current_objective_function :=
objective_function(current_state)
    objective_function_logs = append(objective_function_logs,
```

```

current_objective_function)

    improved := true
    for improved && current_objective_function > 0 {
        neighbor_states :=
generate_neighbor_states(current_state, objective_function,
7750, false)
        best_neighbor_value :=
neighbor_states.min_objective_value
        best_neighbor_index :=
neighbor_states.min_neighbor_index
            if best_neighbor_value >=
current_objective_function {
                improved = false
            } else {
                current_state =
neighbor_states.neighbor_states[best_neighbor_index].swapped_c
ube_state
                current_objective_function =
best_neighbor_value
                objective_function_logs =
append(objective_function_logs, current_objective_function)

                swap_pair.initial_coordinate =
neighbor_states.neighbor_states[best_neighbor_index].initial_c
oordinate
                swap_pair.target_coordinate =
neighbor_states.neighbor_states[best_neighbor_index].target_co
ordinate
                swap_logs = append(swap_logs, swap_pair)
            }
        if improved { iteration += 1}
    }

    local_search_result.objective_function_logs =
objective_function_logs
    local_search_result.swap_logs = swap_logs
    local_search_result.initial_state = copy_cube(cube)
    local_search_result.final_state = current_state
    local_search_result.iteration = iteration

    timeElapsed := time.Since(timeStart)
    local_search_result.time =
int(timeElapsedMilliseconds())
        return local_search_result
}

```

\

c. Penjelasan *source code Steepest Ascent Hill-Climbing*

- **Fungsi `steepest_ascent_hill_climbing`**

```
func steepest_ascent_hill_climbing(cube Cube,
                                     objective_function ObjectiveFunction)
    LocalSearchResultSteepestAscent{
```

Fungsi ini menerima **dua** parameter,

- a. `cube` → *struct* yang merepresentasikan kondisi awal dari suatu *state*.
- b. `objective_function` → fungsi objektif yang akan digunakan untuk mengevaluasi seberapa baik solusi saat ini.

Fungsi ini akan mengembalikan *struct* `LocalSearchResultSteepestAscent` yang mencakup hasil pencarian lokal, termasuk *log* dari nilai fungsi objektif selama proses, urutan *swap* yang terjadi, kondisi awal dan akhir dari *state*, banyak iterasi, dan durasi waktu yang dibutuhkan.

- **Deklarasi Variabel Utama**

```
timeStart := time.Now()
current_state := copy_cube(cube)
iteration := 0 // iteration counter
var local_search_result LocalSearchResultSteepestAscent
objective_function_logs := []int{}
var swap_pair SwapPair
swap_logs := []SwapPair{}
```

- a. `timeStart` → Menyimpan waktu mulai untuk mengukur waktu eksekusi algoritma.
- b. `current_state` → Menyimpan kondisi awal dari *cube* yang telah disalin. *State* ini akan terus diperbarui seiring algoritma menemukan state yang lebih baik.
- c. `local_search_result` → *struct* yang akan menyimpan hasil akhir pencarian.
- d. `objective_function_logs` → Daftar yang mencatat nilai fungsi objektif dari setiap iterasi. *Log* ini berguna untuk melacak kemajuan solusi dan membantu

visualisasi data.

- e. **swap_pair** → Struktur data yang menyimpan informasi pasangan koordinat yang ditukar pada setiap iterasi.
- f. **swap_logs** → Daftar yang mencatat setiap pasangan *swap* yang terjadi selama proses.

- **Inisialisasi Nilai *Objective Function***

```
current_objective_function :=  
objective_function(current_state)  
objective_function_logs = append(objective_function_logs,  
current_objective_function)
```

- a. **current_objective_function** → Variabel yang menyimpan nilai fungsi objektif dari state saat ini, digunakan untuk mengevaluasi kualitas solusi pada iterasi tersebut.
- b. **objective_function_logs** → Array yang mencatat semua nilai fungsi objektif yang dihasilkan selama iterasi, berfungsi untuk memantau perubahan dan perkembangan nilai objektif sepanjang proses algoritma.
- c. **append(objective_function_logs, current_objective_function)**
→ Baris ini menambahkan nilai fungsi objektif saat ini ke dalam log, membantu analisis dan visualisasi hasil setelah algoritma selesai berjalan.

- **Proses Perulangan (*Loop*) Pencarian**

```
improved := true  
for improved && current_objective_function > 0 {
```

- a. **improved** → Boolean yang menunjukkan apakah terdapat perbaikan pada iterasi terakhir. Jika *true*, maka algoritma akan terus mencari tetangga yang lebih baik, namun jika tidak, algoritma akan berhenti.
- b. **current_objective_function > 0** → Syarat lain untuk keluar dari *loop* adalah jika nilai fungsi objektif mencapai 0 (optimal).

- **Membangkitkan Tetangga dan Mencari Tetangga Terbaik**

```
neighbor_states := generate_neighbor_states(current_state,
objective_function, 7750, false)
best_neighbor_value := neighbor_states.min_objective_value
best_neighbor_index := neighbor_states.min_neighbor_index
```

- a. **generate_neighbor_states** → Fungsi ini membangkitkan 7750 *successors* dari **current_state**. Fungsi ini akan mengembalikan struktur dan disimpan di **neighbor_states** yang berisi daftar *state* tetangga beserta nilai fungsi objektif masing-masing dan menyimpan *neighbor* terbaik.
- b. **best_neighbor_value** → Nilai fungsi objektif terbaik dari semua tetangga.
- c. **best_neighbor_index** → Indeks dari *state* tetangga yang memiliki nilai fungsi objektif terbaik.

- **Memperbarui State Saat Ini Jika Ada Peningkatan Value**

```
if best_neighbor_value >= current_objective_function {
    improved = false
} else {
    current_state =
neighbor_states.neighbor_states[best_neighbor_index].swapped_c
ube_state
    current_objective_function = best_neighbor_value
    objective_function_logs = append(objective_function_logs,
current_objective_function)

    swap_pair.initial_coordinate =
neighbor_states.neighbor_states[best_neighbor_index].initial_c
oordinate
    swap_pair.target_coordinate =
neighbor_states.neighbor_states[best_neighbor_index].target_co
ordinate
    swap_logs = append(swap_logs, swap_pair)
}
```

Pada blok ini, algoritma memutuskan apakah state saat ini diperbarui,

- a. Jika **best_neighbor_value** tidak lebih baik dari

- `current_objective_function`, algoritma berhenti karena tidak ada peningkatan lebih lanjut.
- Jika `best_neighbor_value` lebih baik, `current_state` diperbarui dengan state tetangga terbaik dan nilai fungsi objektif diperbarui dengan `best_neighbor_value` serta disimpan ke `objective_function_logs`.
 - `swap_pair` berisi pasangan koordinat yang ditukar pada iterasi ini dan disimpan ke `swap_logs`.

- Menyimpan Hasil Akhir

```

local_search_result.objective_function_logs =
    objective_function_logs
local_search_result.swap_logs = swap_logs
local_search_result.initial_state = copy_cube(cube)
local_search_result.final_state = current_state
local_search_result.iteration = iteration

timeElapsed := time.Since(startTime)
local_search_result.time =
int(timeElapsed.Milliseconds())
return local_search_result

```

Setelah perulangan selesai, hasil akhir disimpan di *struct local_search_result*, yang meliputi,

- `objective_function_logs` → Mencatat nilai fungsi objektif pada setiap iterasi untuk melacak perkembangan kualitas solusi secara bertahap dan bagaimana nilai objektif membaik seiring iterasi berlangsung.
- `swap_logs` → Menyimpan catatan koordinat elemen yang ditukar pada setiap langkah iterasi untuk memahami perubahan struktur solusi serta memungkinkan rekonstruksi urutan perubahan yang dilakukan.
- `initial_state` → Menyimpan kondisi awal *cube* sebelum algoritma memulai pencarian solusi.
- `final_state` → Menyimpan kondisi akhir *cube* setelah algoritma selesai mencari solusi.
- `iteration` → Mencatat jumlah iterasi yang telah dilakukan selama pencarian

solusi.

- f. **timeElapsed** → Menghitung dan menyimpan durasi eksekusi algoritma untuk evaluasi kinerja.

2.2.2 *Hill-Climbing with Sideways Move*

a. Deskripsi *Hill-Climbing with Sideways Move*

Hill-Climbing with Sideways Move adalah varian dari metode *Hill Climbing* yang dirancang untuk mengatasi masalah saat algoritma terjebak dalam *plateau* atau *local optimum*. Dalam teknik ini, jika tidak ada peningkatan pada fungsi objektif dari keadaan saat ini (*current state*), algoritma diizinkan untuk melakukan perpindahan ke keadaan dengan nilai yang sama, atau biasa disebut dengan *sideways move*. Ini memungkinkan eksplorasi ruang solusi lebih luas untuk keluar dari area datar (*plateau*), dengan batasan jumlah perpindahan berturut-turut yang dapat dilakukan untuk menghindari stagnasi.

b. *Source code Hill-Climbing with Sideways Move* dalam bahasa pemrograman Go

```
func hill_climbing_with_sideways_move(cube Cube,
objective function ObjectiveFunction, max_sideways_allowed
int) LocalSearchResultSidewaysMove {
    timeStart := time.Now()
    max_sideways := max_sideways_allowed
    current_state := copy_cube(cube)
    var local_search_result LocalSearchResultSidewaysMove
    objective_function_logs := []int{}
    var swap_pair SwapPair
    swap_logs := []SwapPair{}
    iteration := 0
    max_sideways_counter := 0

    current_objective_function :=
    objective_function(current_state)
    objective_function_logs = append(objective_function_logs,
    current_objective_function)

    improved := true
    for improved && current_objective_function > 0 {
        neighbor_states :=
```

```

generate_neighbor_states(current_state, objective_function,
7750, false)
    best_neighbor_value :=
neighbor_states.min_objective_value
    best_neighbor_index :=
neighbor_states.min_neighbor_index
        if best_neighbor_value > current_objective_function
{
    improved = false
} else {
    if max_sideways_counter < max_sideways {
        current_state =
neighbor_states.neighbor_states[best_neighbor_index].swapped_cube_state
            if current_objective_function ==
best_neighbor_value {
                max_sideways_counter += 1
} else {
                max_sideways_counter = 0
}
        current_objective_function =
best_neighbor_value

        objective_function_logs =
append(objective_function_logs, current_objective_function)

        swap_pair.initial_coordinate =
neighbor_states.neighbor_states[best_neighbor_index].initial_coordinate
            swap_pair.target_coordinate =
neighbor_states.neighbor_states[best_neighbor_index].target_coordinate
                swap_logs = append(swap_logs, swap_pair)
} else {
    improved = false
}
}
if improved { iteration += 1}
}

local_search_result.objective_function_logs =
objective_function_logs
local_search_result.swap_logs = swap_logs
local_search_result.initial_state = copy_cube(cube)
local_search_result.final_state = current_state
local_search_result.max_sideways = max_sideways
local_search_result.iteration = iteration

timeElapsed := time.Since(timeStart)
local_search_result.time =
int(timeElapsedMilliseconds())

```

```
        return local_search_result  
    }
```

c. Penjelasan source code *Hill-Climbing with Sideways Move*

- **Fungsi *hill_climbing_with_sideways_move***

```
func hill_climbing_with_sideways_move(cube Cube,  
objective_function ObjectiveFunction, max_sideways_allowed int)  
LocalSearchResultSidewaysMove {
```

Fungsi ini menerima **tiga** parameter yang mendukung proses pencarian dengan *sideways moves*,

- a. **cube** → *struct* yang mewakili keadaan awal dari suatu *state* atau permasalahan yang akan dipecahkan.
- b. **objective_function** → fungsi yang digunakan untuk menilai kualitas solusi saat ini berdasarkan nilai objektif.
- c. **max_sideways_allowed** → batas maksimal *sideways moves* berturut-turut yang diizinkan, yaitu langkah dengan nilai fungsi objektif yang sama.

Fungsi ini mengembalikan sebuah *struct* **LocalSearchResultSidewaysMove**, yang berisi hasil dari proses pencarian lokal, termasuk catatan perubahan nilai fungsi objektif selama iterasi, urutan *swap* yang dilakukan, serta kondisi akhir dari *state* setelah pencarian selesai.

- **Deklarasi Variabel Utama**

```
timeStart := time.Now()  
max_sideways := max_sideways_allowed  
current_state := copy_cube(cube)  
var local_search_result LocalSearchResultSidewaysMove  
objective_function_logs := []int{}  
var swap_pair SwapPair  
swap_logs := []SwapPair{}  
iteration := 0  
max_sideways_counter := 0
```

Bagian ini menginisialisasi variabel-variabel yang akan digunakan untuk menyimpan informasi penting selama proses pencarian lokal, seperti *state* awal, *log* nilai fungsi

objektif, dan catatan *swap* yang dilakukan. Berdasarkan algoritma di atas,

- a. **timeStart** → Menyimpan waktu mulai untuk mengukur waktu eksekusi algoritma.
- b. **max_sideways** → Menyimpan nilai maksimum *sideways moves* yang diperbolehkan, dari parameter **max_sideways_allowed**.
- c. **current_state** → Menyimpan kondisi awal dari *cube* yang telah disalin. *State* ini akan terus diperbarui seiring algoritma menemukan *state* yang lebih baik.
- d. **local_search_result** → *struct* yang akan menyimpan hasil akhir pencarian.
- e. **objective_function_logs** → Daftar yang mencatat nilai fungsi objektif dari setiap iterasi. *Log* ini berguna untuk melacak kemajuan solusi dan membantu visualisasi data.
- f. **swap_pair** → Struktur data yang menyimpan informasi pasangan koordinat yang ditukar pada setiap iterasi.
- g. **swap_logs** → Daftar yang mencatat setiap pasangan *swap* yang terjadi selama proses.
- h. **iteration** → Variabel untuk mencatat jumlah iterasi selama proses pencarian.
- i. **max_sideways_counter** → Menghitung jumlah *sideways moves* yang telah dilakukan, guna membatasi langkah-langkah yang setara.

- Inisialisasi Nilai *Objective Function*

```
    current_objective_function :=
objective_function(current_state)
    objective_function_logs = append(objective_function_logs,
current_objective_function)
```

Bagian ini menghitung dan mencatat nilai fungsi objektif awal untuk **current_state**. Berdasarkan algoritma di atas,

- a. **current_objective_function :=**
objective_function(current_state) → Menghitung nilai fungsi objektif dari **current_state** dan menyimpannya di **current_objective_function**.

- b. `objective_function_logs = append(objective_function_logs, current_objective_function)` → Menambahkan nilai fungsi objektif awal ke dalam `objective_function_logs` sebagai catatan pertama.

- **Proses Perulangan (*Loop*) Pencarian**

```
improved := true
for improved && current_objective_function > 0 {
```

Bagian ini menetapkan kondisi yang akan menjaga proses iterasi tetap berjalan selama masih ada perbaikan yang ditemukan atau nilai fungsi objektif belum mencapai optimal. Berdasarkan algoritma di atas,

- `improved` → Variabel *boolean* yang menunjukkan apakah ada perbaikan pada iterasi terakhir. Jika `true`, maka algoritma akan terus mencari tetangga yang lebih baik; jika `false`, algoritma akan berhenti.
- `current_objective_function > 0` → Syarat tambahan untuk keluar dari *loop*. Artinya, algoritma akan terus berjalan sampai nilai fungsi objektif mencapai 0, yang mengindikasikan kondisi optimal.

- **Membangkitkan Tetangga dan Mencari Tetangga Terbaik**

```
neighbor_states :=
generate_neighbor_states(current_state, objective_function,
7750, false)
    best_neighbor_value :=
neighbor_states.min_objective_value
        best_neighbor_index :=
neighbor_states.min_neighbor_index
```

Bagian ini bertanggung jawab untuk menghasilkan tetangga dari *state* saat ini dan mencari tetangga dengan nilai fungsi objektif terbaik. Berdasarkan algoritma di atas,

- `neighbor_states := generate_neighbor_states(current_state, objective_function, 7750, false)` → Menghasilkan tetangga dari `current_state` dengan maksimum 7750 tetangga, menggunakan `objective_function` untuk mengevaluasi.

- b. `best_neighbor_value := neighbor_states.min_objective_value`
→ Mengambil nilai fungsi objektif terendah di antara tetangga yang dihasilkan.
- c. `best_neighbor_index := neighbor_states.min_neighbor_index` →
Mengambil indeks dari tetangga dengan nilai fungsi objektif terbaik.

- Memperbarui *State* Saat Ini Jika Ada Peningkatan *Value*

```

        if best_neighbor_value > current_objective_function
    {
        improved = false
    } else {
        if max_sideways_counter < max_sideways {
            current_state =
neighbor_states.neighbor_states[best_neighbor_index].swapped_c
ube_state
            if current_objective_function ==
best_neighbor_value {
                max_sideways_counter += 1
            } else {
                max_sideways_counter = 0
            }
            current_objective_function =
best_neighbor_value

            objective_function_logs =
append(objective_function_logs, current_objective_function)

            swap_pair.initial_coordinate =
neighbor_states.neighbor_states[best_neighbor_index].initial_c
oordinate
            swap_pair.target_coordinate =
neighbor_states.neighbor_states[best_neighbor_index].target_co
ordinate
            swap_logs = append(swap_logs, swap_pair)
        } else {
            improved = false
        }
    }
    if improved { iteration += 1}
}

```

Bagian ini memeriksa apakah tetangga terbaik memberikan peningkatan pada nilai fungsi objektif atau melakukan *sideways move* jika diperlukan. Berdasarkan algoritma di atas,

- a. `if best_neighbor_value > current_objective_function` → Jika nilai tetangga terbaik lebih buruk, `improved` di-set ke `false` untuk mengakhiri *loop*.
- b. `if max_sideways_counter < max_sideways` → Memeriksa apakah *sideways move* yang dilakukan belum mencapai batas maksimal. Jika iya, *state* diperbarui dengan tetangga terbaik.
- c. `current_state = neighbor_states.neighbor_states[best_neighbor_index].swapped_cube_state` → Memperbarui `current_state` ke tetangga terbaik.
- d. `max_sideways_counter += 1` → Meningkatkan *sideways counter* jika nilai objektif tetap sama.
- e. `current_objective_function = best_neighbor_value` → Memperbarui nilai fungsi objektif saat ini.
- f. `swap_logs` → Menyimpan informasi pertukaran koordinat yang dilakukan pada setiap iterasi.

- **Menyimpan Hasil Akhir**

```

local_search_result.objective_function_logs =
objective_function_logs
local_search_result.swap_logs = swap_logs
local_search_result.initial_state = copy_cube(cube)
local_search_result.final_state = current_state
local_search_result.max_sideways = max_sideways
local_search_result.iteration = iteration

timeElapsed := time.Since(startTime)
local_search_result.time =
int(timeElapsed.Milliseconds())
return local_search_result
}

```

Bagian ini bertujuan untuk menyimpan hasil akhir dari pencarian lokal, termasuk *log* perubahan nilai fungsi objektif, catatan *swap*, dan kondisi akhir *state*, ke dalam variabel `local_search_result`, yang akan dikembalikan sebagai *output* fungsi. Berdasarkan algoritma di atas,

- a. `local_search_result.objective_function_logs = objective_function_logs` → Menyimpan *log* perubahan fungsi objektif selama proses pencarian.
- b. `local_search_result.swap_logs = swap_logs` → Menyimpan *log* setiap pasangan *swap* yang dilakukan.
- c. `local_search_result.initial_state = copy_cube(cube)` → Menyimpan *state* awal dari `cube` sebagai referensi.
- d. `local_search_result.final_state = current_state` → Menyimpan *state* terakhir sebagai hasil pencarian.
- e. `local_search_result.max_sideways = max_sideways` → Menyimpan jumlah maksimal *sideways* yang diizinkan.
- f. `local_search_result.iteration = iteration` → Menyimpan jumlah iterasi yang dilakukan selama pencarian.
- g. `timeElapsed := time.Since(timeStart)` → Menghitung waktu eksekusi total algoritma.
- h. `local_search_result.time = int(timeElapsed.Milliseconds())`
→ Menyimpan waktu eksekusi dalam milidetik.

2.2.3 Random Restart Hill-Climbing

a. Deskripsi Random Restart Hill-Climbing

Random Restart Hill Climbing adalah algoritma yang dibangun di atas metode *hill climbing* untuk mengatasi masalah terjebak pada puncak lokal (*local maxima*). Metode ini juga dikenal sebagai *Shotgun Hill Climbing*, karena melakukan beberapa iterasi dari titik awal acak. Setiap iterasi menjalankan algoritma *hill climbing* dari kondisi awal baru yang dihasilkan secara acak, dan jika solusi yang ditemukan pada iterasi baru lebih baik daripada solusi sebelumnya, solusi tersebut disimpan.

Algoritma ini bekerja dengan cara memulai pencarian dari berbagai kondisi awal acak, memungkinkan eksplorasi ruang pencarian yang lebih luas

daripada hanya mengoptimalkan dari satu titik awal. Setelah setiap iterasi, solusi yang ditemukan dibandingkan dengan solusi terbaik yang telah disimpan. Jika iterasi menghasilkan solusi yang lebih baik, maka solusi tersebut menggantikan solusi terbaik sebelumnya. Proses ini terus berlanjut sampai mencapai batas waktu atau iterasi yang telah ditentukan.

b. *Source code Random Restart Hill-Climbing dalam bahasa pemrograman Go*

```
func random_restart_hill_climbing(cube Cube, objective_function
ObjectiveFunction, max_restart_allowed_input int)
LocalSearchResultRestart {
    timeStart := time.Now()
    current_state := copy_cube(cube)
    max_restart_allowed := max_restart_allowed_input
    restart_iteration := 0
    iteration_per_every_restart := []int{}
    var local_search_result LocalSearchResultRestart
    objective_function_logs := []int{}
    var swap_pair SwapPair
    swap_logs := []SwapPair{}
    current_objective_function :=
objective_function(current_state)
    objective_function_logs = append(objective_function_logs,
current_objective_function)

    iteration_per_restart_counter := 0
    for current_objective_function > 0 && restart_iteration <
max_restart_allowed + 1{
        neighbor_states :=
generate_neighbor_states(current_state, objective_function,
7750, false)
        best_neighbor_value :=
neighbor_states.min_objective_value
        best_neighbor_index :=
neighbor_states.min_neighbor_index

        if best_neighbor_value >=
current_objective_function && restart_iteration ==
max_restart_allowed {
            iteration_per_every_restart =
append(iteration_per_every_restart,
iteration_per_restart_counter)
            break
        }
        if best_neighbor_value >=
current_objective_function && restart_iteration <
max_restart_allowed {
```

```

        iteration_per_every_restart =
append(iteration_per_every_restart,
iteration_per_restart_counter)
        iteration_per_restart_counter = 0
        restart_iteration += 1
        current_state = randomize_cube(current_state)
        current_objective_function =
objective_function(current_state)
    } else {
        current_state =
neighbor_states.neighbor_states[best_neighbor_index].swapped_c
ube_state
        current_objective_function =
best_neighbor_value

        swap_pair.initial_coordinate =
neighbor_states.neighbor_states[best_neighbor_index].initial_c
oordinate
        swap_pair.target_coordinate =
neighbor_states.neighbor_states[best_neighbor_index].target_co
ordinate
        swap_logs = append(swap_logs, swap_pair)
        iteration_per_restart_counter += 1

    }
    objective_function_logs =
append(objective_function_logs, current_objective_function)
}

total_iteration := 0
for i := range iteration_per_every_restart {
    total_iteration += iteration_per_every_restart[i]
}

local_search_result.objective_function_logs =
objective_function_logs
local_search_result.swap_logs = swap_logs
local_search_result.initial_state = copy_cube(cube)
local_search_result.final_state = current_state
local_search_result.max_restart = max_restart_allowed
local_search_result.iteration_per_restart =
iteration_per_every_restart
local_search_result.restart_iteration = restart_iteration
local_search_result.total_iteration = total_iteration
timeElapsed := time.Since(timeStart)
local_search_result.time =
int(timeElapsed.Milliseconds())
return local_search_result
}

```

c. Penjelasan source code Random Restart Hill-Climbing

- **Fungsi random_restart_hill_climbing**

```
func random_restart_hill_climbing(cube Cube, objective_function  
ObjectiveFunction, max_restart_allowed_input int)  
LocalSearchResultRestart {
```

Fungsi ini menerima **tiga** parameter utama yang mendukung proses pencarian dengan *random restart*,

- a. **cube** → *struct* yang merepresentasikan kondisi awal dari sebuah *state* atau masalah yang ingin diselesaikan.
- b. **objective_function** → fungsi yang digunakan untuk mengevaluasi kualitas solusi saat ini berdasarkan nilai objektif yang diberikan.
- c. **max_restart_allowed_input** → jumlah maksimum *restart* yang diperbolehkan jika tidak ada peningkatan signifikan dalam iterasi pencarian.

Fungsi ini mengembalikan **LocalSearchResultRestart**, yang berisi hasil dari proses pencarian lokal dengan *restart*, termasuk *log* perubahan nilai fungsi objektif, catatan *swap*, kondisi awal dan akhir *state*, serta jumlah *restart* dan iterasi total yang dilakukan.

- **Deklarasi Variabel Utama**

```
timeStart := time.Now()  
current_state := copy_cube(cube)  
max_restart_allowed := max_restart_allowed_input  
restart_iteration := 0  
iteration_per_every_restart := []int{}  
var local_search_result LocalSearchResultRestart  
objective_function_logs := []int{}  
var swap_pair SwapPair  
swap_logs := []SwapPair{}
```

Bagian ini menginisialisasi variabel-variabel utama yang diperlukan untuk menyimpan informasi penting selama proses pencarian lokal, termasuk *state* awal, *log* nilai fungsi objektif, catatan *swap*, dan batas *restart* yang diperbolehkan. Berdasarkan algoritma di atas,

- a. `timeStart` → Menyimpan waktu mulai untuk mengukur waktu eksekusi algoritma.
- b. `current_state` → Menyimpan kondisi awal dari *cube* yang telah disalin. *State* ini akan diperbarui setiap kali ditemukan *state* yang lebih baik atau ketika terjadi restart.
- c. `max_restart_allowed` → Menyimpan jumlah maksimum *restart* yang diperbolehkan, dari parameter `max_restart_allowed_input`.
- d. `restart_iteration` → Menghitung jumlah *restart* yang telah dilakukan selama pencarian.
- e. `iteration_per_every_restart` → Menyimpan jumlah iterasi yang dilakukan untuk setiap *restart* sebagai catatan.
- f. `local_search_result` → *struct* yang akan menyimpan hasil akhir pencarian.
- g. `objective_function_logs` → Daftar yang mencatat nilai fungsi objektif dari setiap iterasi. Berguna untuk melacak perubahan nilai fungsi objektif.
- h. `swap_pair` → Struktur data yang menyimpan informasi koordinat pasangan yang ditukar pada setiap iterasi.
- i. `swap_logs` → Daftar yang mencatat semua *swap* yang terjadi selama proses pencarian.
- j. `current_objective_function` → Menyimpan nilai fungsi objektif dari `current_state`, dihitung pada awal proses.

- Inisialisasi Nilai *Objective Function* Awal

```

    current_objective_function :=
objective_function(current_state)
    objective_function_logs = append(objective_function_logs,
current_objective_function)

```

Bagian ini menghitung dan mencatat nilai fungsi objektif awal untuk `current_state`. Berdasarkan algoritma di atas,

- a. `current_objective_function :=`

- `objective_function(current_state)` → Menghitung nilai fungsi objektif dari `current_state` dan menyimpannya di `current_objective_function`.
- b. `objective_function_logs = append(objective_function_logs, current_objective_function)` → Menambahkan nilai fungsi objektif awal ke dalam `objective_function_logs` sebagai catatan pertama.

- **Loop Pencarian dan Random Restart**

```

iteration_per_restart_counter := 0
for current_objective_function > 0 && restart_iteration <
max_restart_allowed + 1{
    neighbor_states :=
generate_neighbor_states(current_state, objective_function,
7750, false)
    best_neighbor_value :=
neighbor_states.min_objective_value
    best_neighbor_index :=
neighbor_states.min_neighbor_index

```

Bagian ini mengatur proses *loop* utama pencarian dan *random restart*. Pencarian terus dilakukan selama nilai fungsi objektif belum mencapai optimal (lebih dari 0) dan jumlah *restart* belum melebihi batas.

- `iteration_per_restart_counter := 0` → Variabel yang menghitung jumlah iterasi dalam satu sesi *restart*.
- `for current_objective_function > 0 && restart_iteration < max_restart_allowed + 1` → *Loop* berjalan selama nilai fungsi objektif lebih besar dari 0 dan jumlah *restart* belum mencapai batas maksimum.
- `neighbor_states := generate_neighbor_states(current_state, objective_function, 7750, false)` → Menghasilkan tetangga dari `current_state` dengan maksimum 7750 *state* tetangga.
- `best_neighbor_value := neighbor_states.min_objective_value` → Mengambil nilai fungsi objektif terbaik di antara tetangga yang dihasilkan.
- `best_neighbor_index := neighbor_states.min_neighbor_index` → Mengambil indeks dari *state* tetangga yang memiliki nilai fungsi objektif terbaik.

- Memeriksa dan Melakukan *Random Restart* Jika Tidak Ada Perbaikan

```

        if best_neighbor_value >=
current_objective_function && restart_iteration == max_restart_allowed {
            iteration_per_every_restart =
append(iteration_per_every_restart,
iteration_per_restart_counter)
            break
        }
        if best_neighbor_value >=
current_objective_function && restart_iteration <
max_restart_allowed {
            iteration_per_every_restart =
append(iteration_per_every_restart,
iteration_per_restart_counter)
            iteration_per_restart_counter = 0
            restart_iteration += 1
            current_state = randomize_cube(current_state)
            current_objective_function =
objective_function(current_state)
        } else {
            current_state =
neighbor_states.neighbor_states[best_neighbor_index].swapped_c
ube_state
            current_objective_function =
best_neighbor_value

            swap_pair.initial_coordinate =
neighbor_states.neighbor_states[best_neighbor_index].initial_c
oordinate
            swap_pair.target_coordinate =
neighbor_states.neighbor_states[best_neighbor_index].target_co
ordinate
            swap_logs = append(swap_logs, swap_pair)
            iteration_per_restart_counter += 1
        }
    }
}

```

Bagian ini memeriksa apakah nilai fungsi objektif tetangga lebih baik dari *state* saat ini; jika tidak, algoritma melakukan *restart* acak pada *state*.

- a. `if best_neighbor_value >= current_objective_function && restart_iteration == max_restart_allowed` → Jika nilai terbaik dari tetangga tidak lebih baik dan batas *restart* telah tercapai, *loop* akan berhenti.
- b. `if best_neighbor_value >= current_objective_function &&`

```
restart_iteration < max_restart_allowed → Jika tidak ada perbaikan dan restart masih diizinkan, algoritma melakukan random restart.
```

- c. `current_state = randomize_cube(current_state)` → Mengacak ulang state untuk memulai dari titik baru.
- d. `current_objective_function = objective_function(current_state)` → Menghitung nilai fungsi objektif dari `current_state` setelah restart.
- e. `swap_logs` → Menyimpan swap yang dilakukan saat mencapai tetangga terbaik.
- f. `iteration_per_restart_counter += 1` → Menghitung jumlah iterasi dalam sesi restart ini.

- **Mencatat Nilai Objective Function**

```
objective_function_logs =  
append(objective_function_logs, current_objective_function)  
}
```

Menambahkan nilai `current_objective_function` ke dalam `objective_function_logs` pada setiap iterasi atau *restart*. Ini memungkinkan analisis perubahan nilai fungsi objektif di setiap langkah pencarian.

- **Menghitung Total Iterasi**

```
total_iteration := 0  
for i := range iteration_per_every_restart {  
    total_iteration += iteration_per_every_restart[i]  
}
```

Bagian ini menghitung total iterasi dari semua sesi *restart* untuk mendapatkan jumlah keseluruhan langkah pencarian.

- a. `total_iteration := 0` → Menginisialisasi variabel untuk menghitung jumlah iterasi total.
- b. `for i := range iteration_per_every_restart` → Loop melalui setiap iterasi per *restart* untuk menjumlahkan semuanya.

- **Menyimpan Hasil Akhir**

```

    local_search_result.objective_function_logs =
objective_function_logs
    local_search_result.swap_logs = swap_logs
    local_search_result.initial_state = copy_cube(cube)
    local_search_result.final_state = current_state
    local_search_result.max_restart = max_restart_allowed
    local_search_result.iteration_per_restart =
iteration_per_every_restart
    local_search_result.restart_iteration = restart_iteration
    local_search_result.total_iteration = total_iteration
    timeElapsed := time.Since(timeStart)
    local_search_result.time =
int(timeElapsed.Milliseconds())
    return local_search_result
}

```

Bagian ini menyimpan hasil akhir dari proses pencarian ke dalam `local_search_result` dan mengembalikannya sebagai *output* fungsi.

- a. `local_search_result.objective_function_logs = objective_function_logs` → Menyimpan *log* nilai fungsi objektif selama proses pencarian.
- b. `local_search_result.swap_logs = swap_logs` → Menyimpan catatan *swap* yang dilakukan selama pencarian.
- c. `local_search_result.initial_state = copy_cube(cube)` → Menyimpan *state* awal sebagai referensi awal pencarian.
- d. `local_search_result.final_state = current_state` → Menyimpan *state* terakhir setelah pencarian selesai.
- e. `local_search_result.max_restart = max_restart_allowed` → Menyimpan jumlah maksimum *restart* yang diperbolehkan.
- f. `local_search_result.iteration_per_restart = iteration_per_every_restart` → Menyimpan *log* jumlah iterasi untuk setiap sesi *restart*.
- g. `local_search_result.restart_iteration = restart_iteration` → Menyimpan jumlah *restart* yang dilakukan.
- h. `local_search_result.total_iteration = total_iteration` → Menyimpan jumlah total iterasi yang dilakukan.
- i. `timeElapsed := time.Since(timeStart)` → Menghitung waktu eksekusi

keseluruhan algoritma.

j. `local_search_result.time = int(timeElapsed.Milliseconds())`

→ Menyimpan waktu eksekusi dalam milidetik.

2.2.4 *Stochastic Hill-Climbing*

a. Deskripsi *Stochastic Hill-Climbing*

Stochastic Hill Climbing adalah algoritma optimasi yang menggunakan elemen acak dalam proses pencarinya. Algoritma ini adalah varian dari *hill climbing* standar yang tetap berfokus pada pencarian solusi lokal, namun perbedaannya terletak pada cara memilih langkah selanjutnya. Jika *hill climbing* konvensional selalu memilih langkah terbaik (tercuram), *stochastic hill climbing* memilih langkah secara acak di antara pilihan *uphill moves*. Pemilihan ini bisa tergantung pada curamnya kenaikan, sehingga memberikan fleksibilitas dalam pencarian solusi.

Proses dimulai dengan membuat solusi awal secara acak. Kemudian, algoritma membuat salinan solusi terbaik sejauh ini dan memodifikasinya untuk mencari solusi baru. Jika solusi baru lebih baik dari yang sebelumnya, algoritma menggantinya sebagai solusi terbaik saat ini. Jika tidak, solusi tersebut dibuang, dan proses dilanjut. Hal ini berlanjut sampai waktu atau iterasi yang ditentukan berakhir.

b. *Source code Stochastic Hill-Climbing* dalam bahasa pemrograman Go

```
func stochastic_hill_climbing(cube Cube, objective_function  
ObjectiveFunction, max_iteration_allowed int)  
LocalSearchResultStochastic{  
    timeStart := time.Now()  
    max_iteration := max_iteration_allowed  
    current_state := copy_cube(cube)  
    current_iteration := 0  
    var local_search_result LocalSearchResultStochastic  
    objective_function_logs := []int{}  
    var swap_pair SwapPair  
    swap_logs := []SwapPair{}  
  
    current_objective_function :=
```

```

objective_function(current_state)
    objective_function_logs = append(objective_function_logs,
current_objective_function)

        for current_iteration < max_iteration &&
current_objective_function > 0 {
            neighbor_states :=
generate_neighbor_states(current_state, objective_function, 1,
true)
            random_neighbor_index :=
neighbor_states.min_neighbor_index
            random_neighbor :=
neighbor_states.neighbor_states[random_neighbor_index]
            random_neighbor_value :=
random_neighbor.objective_function_value

            if random_neighbor_value <
current_objective_function {
                current_state =
random_neighbor.swapped_cube_state
                current_objective_function =
random_neighbor_value
                objective_function_logs =
append(objective_function_logs, current_objective_function)

                swap_pair.initial_coordinate =
random_neighbor.initial_coordinate
                swap_pair.target_coordinate =
random_neighbor.target_coordinate
                swap_logs = append(swap_logs, swap_pair)
            }
            current_iteration += 1
        }

        local_search_result.objective_function_logs =
objective_function_logs
        local_search_result.swap_logs = swap_logs
        local_search_result.initial_state = copy_cube(cube)
        local_search_result.final_state = current_state
        local_search_result.max_iteration = max_iteration
        local_search_result.iteration = current_iteration
        timeElapsed := time.Since(startTime)
        local_search_result.time =
int(timeElapsed.Milliseconds())
        return local_search_result
}

```

c. Penjelasan source code Stochastic Hill-Climbing

```
func stochastic_hill_climbing(cube Cube, objective_function  
ObjectiveFunction, max_iteration_allowed int)  
LocalSearchResultStochastic{
```

Fungsi ini menerima **tiga** parameter yang mendukung proses pencarian secara acak (*stochastic hill climbing*),

- **cube** → *struct* yang merepresentasikan kondisi awal dari sebuah *state* atau permasalahan yang akan dicari solusinya.
- **objective_function** → fungsi yang digunakan untuk menilai seberapa baik solusi saat ini berdasarkan nilai objektif yang diberikan.
- **max_iteration_allowed** → jumlah maksimum iterasi yang diperbolehkan dalam proses pencarian.

Fungsi ini mengembalikan **LocalSearchResultStochastic**, yang berisi hasil dari proses pencarian lokal secara acak, termasuk *log* perubahan nilai fungsi objektif, catatan *swap*, kondisi awal dan akhir *state*, serta jumlah iterasi yang dilakukan.

- **Deklarasi Variabel Utama**

```
timeStart := time.Now()  
max_iteration := max_iteration_allowed  
current_state := copy_cube(cube)  
current_iteration := 0  
var local_search_result LocalSearchResultStochastic  
objective_function_logs := []int{}  
var swap_pair SwapPair  
swap_logs := []SwapPair{}
```

Bagian ini menginisialisasi variabel-variabel utama yang diperlukan untuk menyimpan informasi penting selama proses pencarian, seperti *state* awal, *log* nilai fungsi objektif, dan catatan *swap* yang dilakukan..

- a. **timeStart** → Menyimpan waktu mulai untuk mengukur waktu eksekusi algoritma.
- b. **max_iteration** → Menyimpan nilai maksimum iterasi yang diizinkan, diambil dari parameter **max_iteration_allowed**.

- c. **current_state** → Menyimpan kondisi awal dari *cube* yang telah disalin. *State* ini akan diperbarui seiring algoritma menemukan *state* yang lebih baik.
- d. **current_iteration** → Variabel untuk menghitung jumlah iterasi yang dilakukan.
- e. **local_search_result** → *struct* yang akan menyimpan hasil akhir pencarian.
- f. **objective_function_logs** → Daftar yang mencatat nilai fungsi objektif dari setiap iterasi. Berguna untuk melacak kemajuan solusi dan membantu visualisasi data.
- g. **swap_pair** → Struktur data yang menyimpan informasi koordinat pasangan yang ditukar pada setiap iterasi.
- h. **swap_logs** → Daftar yang mencatat setiap pasangan *swap* yang terjadi selama proses.

- Inisialisasi Nilai *Objective Function* Awal

```
current_objective_function :=
objective_function(current_state)
objective_function_logs = append(objective_function_logs,
current_objective_function)
```

Bagian ini menghitung dan mencatat nilai fungsi objektif awal untuk *state* yang sedang diproses.

- a. **current_objective_function :=**
objective_function(current_state) → Menghitung nilai fungsi objektif dari **current_state** dan menyimpannya di **current_objective_function**.
- b. **objective_function_logs = append(objective_function_logs,**
current_objective_function) → Menambahkan nilai fungsi objektif awal ke dalam **objective_function_logs** sebagai catatan pertama.

- Proses Perulangan (*Loop*) Pencarian

```
for current_iteration < max_iteration &&
current_objective_function > 0 {
```

Bagian ini menetapkan kondisi untuk menjaga proses iterasi tetap berjalan selama jumlah iterasi belum mencapai batas dan nilai fungsi objektif belum mencapai optimal.

```
for      current_iteration      <      max_iteration      &&  
current_objective_function  >  0  → Loop berjalan selama iterasi belum  
mencapai batas maksimum (max_iteration) dan nilai fungsi objektif lebih besar dari 0  
(belum optimal).
```

- **Mebangkitkan Tetangga Acak dan Mengevaluasi**

```
neighbor_states :=  
generate_neighbor_states(current_state, objective_function, 1,  
true)  
random_neighbor_index :=  
neighbor_states.min_neighbor_index  
random_neighbor :=  
neighbor_states.neighbor_states[random_neighbor_index]  
random_neighbor_value :=  
random_neighbor.objective_function_value
```

Bagian ini bertugas untuk menghasilkan satu tetangga secara acak dari *state* saat ini dan mengevaluasi nilai fungsi objektifnya.

- a. `neighbor_states := generate_neighbor_states(current_state,
objective_function, 1, true)` → Menghasilkan satu tetangga acak dari `current_state` dengan menggunakan `objective_function` sebagai penilaian.
- b. `random_neighbor_index := neighbor_states.min_neighbor_index`
→ Mengambil indeks pertama dari tetangga yang terpilih secara acak.
- c. `random_neighbor := neighbor_states.neighbor_states[random_neighbor_index]` →
Mengambil informasi lengkap dari tetangga acak tersebut.
- d. `random_neighbor_value := random_neighbor.objective_function_value` → Mengambil nilai fungsi objektif dari tetangga acak untuk dibandingkan.

- **Memperbarui State Jika Tetangga Memiliki Nilai Lebih Baik**

```

        if random_neighbor_value <
current_objective_function {
            current_state =
random_neighbor.swapped_cube_state
            current_objective_function =
random_neighbor_value
            objective_function_logs =
append(objective_function_logs, current_objective_function)

            swap_pair.initial_coordinate =
random_neighbor.initial_coordinate
            swap_pair.target_coordinate =
random_neighbor.target_coordinate
            swap_logs = append(swap_logs, swap_pair)
}
        current_iteration += 1
}

```

Bagian ini memeriksa apakah tetangga acak memiliki nilai fungsi objektif yang lebih rendah (lebih baik); jika ya, maka algoritma memperbarui `current_state` dengan tetangga tersebut.

- a. `if random_neighbor_value < current_objective_function` → Memeriksa apakah tetangga acak memberikan perbaikan pada nilai fungsi objektif.
- b. `current_state = random_neighbor.swapped_cube_state` → Memperbarui `current_state` dengan *state* tetangga yang lebih baik.
- c. `current_objective_function = random_neighbor_value` → Memperbarui nilai fungsi objektif dengan nilai tetangga yang lebih baik.
- d. `objective_function_logs = append(objective_function_logs, current_objective_function)` → Menambahkan nilai fungsi objektif terbaru ke dalam `objective_function_logs`.
- e. `swap_pair.initial_coordinate` dan `swap_pair.target_coordinate` → Menyimpan koordinat *swap* yang dilakukan untuk mencapai state tetangga.
- f. `swap_logs = append(swap_logs, swap_pair)` → Menyimpan informasi pasangan *swap* ke dalam `swap_logs`.
- g. `current_iteration += 1` → Meningkatkan iterasi pencarian.

- Menyimpan Hasil Akhir

```
    local_search_result.objective_function_logs =
objective_function_logs
    local_search_result.swap_logs = swap_logs
    local_search_result.initial_state = copy_cube(cube)
    local_search_result.final_state = current_state
    local_search_result.max_iteration = max_iteration
    local_search_result.iteration = current_iteration
    timeElapsed := time.Since(startTime)
    local_search_result.time =
int(timeElapsed.Milliseconds())
    return local_search_result
}
```

Bagian ini menyimpan hasil akhir dari proses pencarian ke dalam `local_search_result` dan mengembalikannya sebagai *output* fungsi.

- a. `local_search_result.objective_function_logs = objective_function_logs` → Menyimpan *log* nilai fungsi objektif dari setiap iterasi dalam `local_search_result`.
- b. `local_search_result.swap_logs = swap_logs` → Menyimpan *log* dari setiap pasangan *swap* yang dilakukan.
- c. `local_search_result.initial_state = copy_cube(cube)` → Menyimpan *state* awal sebagai referensi dari kondisi awal pencarian.
- d. `local_search_result.final_state = current_state` → Menyimpan *state* terakhir sebagai hasil pencarian.
- e. `local_search_result.max_iteration = max_iteration` → Menyimpan batas maksimum iterasi yang diizinkan.
- f. `local_search_result.iteration = current_iteration` → Menyimpan jumlah iterasi yang dilakukan selama pencarian.
- g. `timeElapsed := time.Since(startTime)` → Menghitung waktu eksekusi total dari algoritma.
- h. `local_search_result.time = int(timeElapsed.Milliseconds())` → Menyimpan waktu eksekusi dalam milidetik.

2.2.5 Simulated Annealing

a. Deskripsi Simulated Annealing

Simulated Annealing adalah sebuah algoritma yang menggabungkan *Hill-Climbing* dengan *Complete Random Walk*. Algoritma ini juga dapat disebut sebagai versi algoritma *Stochastic Hill-Climbing* dengan membolehkan kembali ke *state* yang lebih buruk (*downhill*). Sehingga, hal yang spesial pada algoritma *Simulated Annealing* adalah tidak hanya *neighbor* dengan *value objective function* lebih baik yang dapat menjadi *current state*, tetapi *neighbor* dengan nilai *objective function* lebih buruk-pun dapat menjadi *current state* dengan probabilitas $e^{\Delta E/T}$, dengan T adalah “*temperatur*” sebagai fungsi waktu dari t dan ΔE adalah selisih dari *value* pada *neighbor* dengan nilai *value* pada *current state*.

Jika *value* pada *neighbor* memiliki nilai lebih baik daripada *value* pada *current state* ($\Delta E > 0$), maka *neighbor* akan menjadi *current state*. Namun, jika *neighbor* memiliki *value* lebih buruk daripada *current state* ($\Delta E < 0$), maka tetangga dapat menjadi *current state* jika *random number* atau angka pembanding yang didapatkan berada di dalam *range* nilai probabilitas $e^{\Delta E/T}$.

b. Source code Simulated Annealing dalam bahasa pemrograman Go

```
func simulated_annealing(cube Cube, objective_function
ObjectiveFunction, initial_temperature float64, cooling_rate
float64) SimulatedAnnealingResult {
    timeStart := time.Now() // for time elapsed purpose
    temperature := []float64{} // for visualization purpose
    probability_plot := []float64{} // for visualization
purpose
    current_state := copy_cube(cube)
    stuck_iteration := 0
    var simulated_local_result SimulatedAnnealingResult
    objective_function_logs := []int{}
    var swap_pair SwapPair
    swap_logs := []SwapPair{}
    var current_iteration float64 = 1
```

```

        current_temperature := initial_temperature
        temperature = append(temperature, current_temperature)
// for visualization purpose
        current_objective_function :=
objective_function(current_state)
        objective_function_logs =
append(objective_function_logs, current_objective_function)

        tidakBerubah := 0 // cek deltaE < 0 tetapi tidak
berpindah
        // for current_iteration < max_iteration &&
current_objective_function > 0 {
            for current_temperature > 0.0 &&
current_objective_function > 0 {
                current_temperature =
schedule_temperature(current_iteration, current_temperature,
cooling_rate)
                temperature = append(temperature,
math.Round(current_temperature * 100) / 100) // for
visualization purpose

                if current_temperature == 0.0 {
                    break
                }
                neighbor_states :=
generate_neighbor_states(current_state, objective_function, 1,
true)
                random_neighbor_index :=
neighbor_states.min_neighbor_index
                random_neighbor :=
neighbor_states.neighbor_states[random_neighbor_index]
                random_neighbor_value :=
random_neighbor.objective_function_value

                delta_E := float64(current_objective_function -
random_neighbor_value)
                if delta_E > 0 {
                    probability_plot = append(probability_plot, 1)
                    current_state =
random_neighbor.swapped_cube_state
                    current_objective_function =
random_neighbor_value
                    objective_function_logs =
append(objective_function_logs, current_objective_function)

                    swap_pair.initial_coordinate =
random_neighbor.initial_coordinate
                    swap_pair.target_coordinate =
random_neighbor.target_coordinate
                    swap_logs = append(swap_logs, swap_pair)
                } else {

```

```

        probability := math.Exp(delta_E /
current_temperature)
        probability_plot = append(probability_plot,
probability)
        if probability > rand.Float64() {
            current_state =
random_neighbor.swapped_cube_state
            current_objective_function =
random_neighbor_value
            objective_function_logs =
append(objective_function_logs, current_objective_function)

            swap_pair.initial_coordinate =
random_neighbor.initial_coordinate
            swap_pair.target_coordinate =
random_neighbor.target_coordinate
            swap_logs = append(swap_logs, swap_pair)
            stuck_iteration += 1
        } else { tidakBerubah += 1 } // karena lebih
buruk dan tidak masuk threshold
    }
    current_iteration += 1
}

simulated_local_result.objective_function_logs =
objective_function_logs
simulated_local_result.swap_logs = swap_logs
simulated_local_result.final_state = current_state
simulated_local_result.stuck_iteration = stuck_iteration
simulated_local_result.temperatures = temperature
simulated_local_result.not_changed = tidakBerubah
simulated_local_result.probability_plot =
probability_plot
timeElapsed := time.Since(timeStart)
simulated_local_result.time =
int(timeElapsed.Milliseconds())
return simulated_local_result
}

```

c. Penjelasan source code Simulated Annealing

- Fungsi simulated_annealing

```

func simulated_annealing(cube Cube, objective_function
ObjectiveFunction, initial_temperature float64, cooling_rate
float64) SimulatedAnnealingResult {

```

Fungsi ini menerima beberapa parameter,

- a. **cube** → *struct* yang merepresentasikan kondisi awal dari state yang sedang dievaluasi.
- b. **objective_function** → fungsi objektif yang digunakan untuk mengevaluasi solusi. Semakin rendah nilai fungsi ini, semakin optimal solusinya.
- c. **initial_temperature** → suhu awal yang mengontrol probabilitas penerimaan solusi yang lebih buruk pada awal algoritma.
- d. **cooling_rate** → parameter untuk *scheduling* temperatur.

Fungsi ini mengembalikan *struct* **SimulatedAnnealingResult** , yang mencakup hasil akhir pencarian lokal, seperti *log* dari nilai fungsi objektif selama proses, daftar koordinat yang ditukar, dan kondisi akhir dari *state*.

- **Deklarasi Variabel Utama**

```
current_state := copy_cube(cube)
var simulated_local_result SimulatedAnnealingResult
objective_function_logs := []int{}
var swap_pair SwapPair
swap_logs := []SwapPair{}
var current_iteration float64 = 1
current_temperature := initial_temperature
```

- a. **current_state** → Menyimpan kondisi awal dari *cube*, yang akan terus diperbarui saat algoritma menemukan state yang lebih baik.
- b. **simulated_local_result** → Struktur data yang akan menyimpan hasil akhir pencarian, termasuk *log* dan kondisi *final*.
- c. **objective_function_logs** → Daftar yang mencatat nilai fungsi objektif dari setiap iterasi, yang berfungsi untuk melacak progres dari pencarian solusi.
- d. **swap_pair** → Struktur data yang menyimpan informasi pasangan koordinat yang ditukar pada setiap iterasi.
- e. **swap_logs** → Daftar yang mencatat setiap pasangan *swap* yang terjadi selama proses.
- f. **current_iteration** → Menyimpan iterasi saat ini, dimulai dari 1.

- g. **current_temperature** → Variabel yang menyimpan suhu saat ini, dimulai dari suhu awal.

- Inisialisasi Nilai Awal *Objective Function*

```
current_objective_function :=  
objective_function(current_state)  
objective_function_logs = append(objective_function_logs,  
current_objective_function)  
fmt.Printf("Objective Function Value: %d, Current Iteration:  
%d, Current Temperature: %f\n", current_objective_function,  
int(current_iteration), current_temperature)
```

- a. **current_objective_function = append(objective_function_logs,**
current_objective_function) → Mencatat nilai
current_objective_function ke dalam daftar
objective_function_logs. Log ini memantau perubahan nilai fungsi objektif sepanjang iterasi dan memvisualisasi perkembangan solusi dan analisis performa algoritma.

- Looping Utama untuk *Simulated Annealing*

```
for current_iteration < max_iteration &&  
current_objective_function > 0 {
```

Perulangan ini akan terus berjalan selama,

- a. **current_iteration < max_iteration** → Perulangan akan terus berjalan selama **current_iteration** belum mencapai batas maksimal iterasi yang telah ditentukan (**max_iteration**). Kondisi ini memastikan algoritma tidak berjalan tanpa batas dan menghentikan proses jika sudah mencapai jumlah iterasi yang diinginkan.
- b. **current_objective_function > 0** → Perulangan akan berlanjut selama nilai fungsi objektif (**current_objective_function**) masih lebih besar dari 0. Kondisi ini mengindikasikan algoritma belum menemukan solusi optimal yang memiliki nilai fungsi objektif 0.

- **Pembaruan Suhu Menggunakan `schedule_temperature`**

```
current_temperature = schedule_temperature(current_iteration,
current_temperature, max_iteration, cooling_rate)
if current_temperature == 0.0 {
    break
}
```

- a. `schedule_temperature` → Fungsi yang mengatur penurunan suhu secara bertahap berdasarkan iterasi saat ini, suhu awal, batas iterasi, dan laju pendinginan.
- b. Jika `current_temperature` mencapai 0, perulangan akan dihentikan karena suhu yang terlalu rendah mengindikasikan tidak ada kemungkinan perubahan lagi.

- **Membangkitkan Tetangga dan Memilih Tetangga Secara Acak**

```
neighbor_states := generate_neighbor_states(current_state,
objective_function, 1, true)
random_neighbor_index := neighbor_states.min_neighbor_index
random_neighbor :=
neighbor_states.neighbor_states[random_neighbor_index]
random_neighbor_value :=
random_neighbor.objective_function_value
```

- a. `generate_neighbor_states` → Fungsi ini menghasilkan tetangga `current_state`. Di sini, kita hanya memilih satu tetangga secara acak dengan pengaturan `true` agar variasi solusi lebih besar.
 - b. `random_neighbor` → Tetangga acak yang dipilih dari daftar `neighbor states`.
 - c. `random_neighbor_value` → Nilai fungsi objektif dari `random_neighbor`.
- **Menghitung ΔE , Memutuskan untuk Menerima atau Menolak Solusi Baru**

```
delta_E := float64(current_objective_function -
random_neighbor_value)
```

- a. **delta_E** → Selisih antara nilai fungsi objektif `current_state` dengan `random_neighbor`. Jika `delta_E` positif, artinya `random_neighbor` lebih baik. Sebaliknya, jika negatif, berarti lebih buruk.

- **Kondisi Penerimaan Solusi Baru**

```

if delta_E > 0 {
    probability_plot = append(probability_plot,
1)
    current_state =
random_neighbor.swapped_cube_state
    current_objective_function =
random_neighbor_value
    objective_function_logs =
append(objective_function_logs, current_objective_function)

    swap_pair.initial_coordinate =
random_neighbor.initial_coordinate
    swap_pair.target_coordinate =
random_neighbor.target_coordinate
    swap_logs = append(swap_logs, swap_pair)
} else {
    probability := math.Exp(delta_E /
current_temperature)
    probability_plot = append(probability_plot,
probability)
    if probability > rand.Float64() {
        current_state =
random_neighbor.swapped_cube_state
        current_objective_function =
random_neighbor_value
        objective_function_logs =
append(objective_function_logs, current_objective_function)

        swap_pair.initial_coordinate =
random_neighbor.initial_coordinate
        swap_pair.target_coordinate =
random_neighbor.target_coordinate
        swap_logs = append(swap_logs, swap_pair)
        stuck_iteration += 1
    }
}

```

Pada bagian ini,

- a. Jika `delta_E` positif (solusi `neighbor` lebih baik daripada `current`), solusi langsung diperbarui ke `random_neighbor.swapped_cube_state`. Fungsi

- objektif (`current_objective_function`) juga diperbarui ke nilai yang lebih baik, yaitu `random_neighbor_value`. *Log swap* dan *log fungsi objektif dicatat*.
- b. Jika `delta_E` negatif (solusi *neighbor* lebih buruk daripada *current*), algoritma akan menghitung probabilitas dengan menggunakan rumus eksponensial. Semakin tinggi suhu (`current_temperature`), semakin besar kemungkinan untuk menerima solusi lebih buruk. Algoritma kemudian membandingkan probabilitas ini dengan *random number* dari `rand.Float64()` yang menghasilkan angka antara 0 dan 1. Jika probabilitas lebih besar dari *random number*, solusi lebih buruk diterima. Jika tidak, solusi lebih buruk ditolak dan algoritma tetap pada kondisi saat ini. Fungsi objektif (`current_objective_function`) diperbarui ke nilai yang lebih buruk, *log swap* dan *log fungsi objektif dicatat*.

- **Log dan Pembaruan Iterasi**

```
probability_plot = append(probability_plot, 1)
objective_function_logs = append(objective_function_logs,
current_objective_function)
swap_logs = append(swap_logs, swap_pair)
```

- Setiap kali solusi diterima (baik lebih baik atau lebih buruk), logs diperbarui.
- `probability_plot` mencatat nilai 1 sebagai indikasi bahwa solusi diterima.
- `objective_function_logs` mencatat nilai terbaru dari fungsi objektif setelah pembaruan.
- Swap logs mencatat koordinat pasangan yang ditukar (`swap_pair`), berisi `initial_coordinate` dan `target_coordinate` dari solusi tetangga.

- **Menyimpan Hasil Akhir**

```
local_search_result.objective_function_logs =
objective_function_logs
local_search_result.swap_logs = swap_logs
local_search_result.final_state = current_state
return local_search_result
```

Setelah loop selesai, hasil akhir disimpan dalam `local_search_result`, mencakup,

- a. `Objective_function_logs` → Catatan perubahan nilai fungsi objektif selama proses.
- b. `swap_logs` → Daftar pasangan koordinat yang ditukar.
- c. `final_state` → Kondisi akhir dari `current_state`, yang merupakan hasil terbaik yang ditemukan.

2.2.6 *Genetic Algorithm*

a. Deskripsi *Genetic Algorithm*

Genetic Algorithm adalah sebuah algoritma yang menggunakan pendekatan proses seleksi alamiah atau genetik dan biasa disebut dengan *stochastic beam search* yang bekerja secara paralel. Algoritma dimulai dengan me-generate *initial states* sebanyak k lalu *successor* akan ditentukan dengan cara rekombinasi yang variatif pada dua *parent states*. *States* akan direpresentasikan sebagai *string* agar mudah untuk dikombinasikan. Proses rekombinasi pada *parent states* yang dilakukan biasanya *selection*, *crossover*, dan *mutation*.

Selection dapat dilakukan secara *random* dengan memanfaatkan *fitness function* dalam menentukan *value* semua *parent states*. Semua *value* dapat dibuat dalam model *roulette wheel* dengan area probabilitas berdasarkan perbandingan antar *value*-nya, lalu dua *parent states* akan dipilih secara *random*.

Crossover dapat dilakukan pada kedua *parent states* yang terpilih dengan memotong *string* pada kedua *parent states* lalu menyilangkannya untuk membentuk kombinasi *string* baru sebagai “anaknya”.

Mutation dapat dilakukan dengan cara mengubah isi pada *string*. Hal yang diubah dapat berupa mengganti karakter, menukar posisi, melakukan *inverse* pada kumpulan karakter, dan mengacak kumpulan karakter pada *string*. Sehingga hasil akhir dari beberapa proses rekombinasi berupa *string* yang merepresentasikan *state*.

b. *Source code Genetic Algorithm* dalam bahasa pemrograman Go

```

func genetic_algorithm(objective_function ObjectiveFunction,
n_population int, n_iteration int) GeneticAlgorithmResult {

    // Essential constants
    var genetic_algorithm_result GeneticAlgorithmResult
    timeStart := time.Now()
    population := n_population
    iteration := n_iteration

    // initial state (constant) -> consists of n_population
    cubes
    initial_cube := []Cube{}
    initial_best_cube := Cube{}
    initial_best_value := 999

    // final state (always treat the last) -> consists of
    n_population cubes
    final_cube := []Cube{}
    final_best_cube := Cube{}
    final_best_value := 999

    objective_value_plot := []int{} // for visualization:
    plot of the best value of every iteration
    avg_objective_value := []int{} // for visualization:
    average of the value of every iteration
    obj_value_sum := 0

    // Used for iterations
    current_cube := []Cube{}
    _ = current_cube

    // ===== generate n random cubes to initial_cube, copy
    to current_cube =====

    // get initial_best_cube, initial_best_value
    // generate random n_population initial cubes
    // NOT yet iteration

    for i := range population {
        _ = i
        initial_cube = append(initial_cube,
        generate_random_cube()) // generate n random cubes
        obj_value_check :=
        objective_function(initial_cube[i]) // check objective value
        obj_value_sum += obj_value_check
        if obj_value_check < initial_best_value { //get the
        best initial value
            initial_best_value = obj_value_check
            initial_best_cube = initial_cube[i]
        }
    }
}

```

```

    // count initial avg_objective_value
    avg_objective_value = append(avg_objective_value,
obj_value_sum / population)

    // deep copy initial_cube into current_cube
current_cube = slices.Clone(initial_cube)

    // ===== initial max objective value plot =====
objective_value_plot = append(objective_value_plot,
initial_best_value)

    // ===== Start of Iterations =====
for i := range iteration {
    = i
    obj_value_sum = 0
    // initial preparation for iterations
    current_best_value := 999
    fitness_sum := 0.0
    // best_objective_value = 999
    n_objective_value := []int{}

    // ===== Count all fitness value =====
    fitness_value := []float64{}
    for j := range population { // check fitness value
        = j
        n_objective_value = append(n_objective_value,
objective_function(current_cube[j]))
        fitness_value = append(fitness_value, 1.0 /
float64(n_objective_value[j] + 1))
    }

    // Normalize fitness values
    for _, value := range fitness_value {
        fitness_sum += value
    }
    for j := range fitness_value {
        fitness_value[j] /= fitness_sum
    }

    final_cube = []Cube{}
    // ===== Selection using weightedRandomChoiceFloat
=====

    for j := range population { // spin wheel (weighted
probs)
        = j
        x := weightedRandomChoiceFloat(fitness_value)
        // fmt.Printf("x: %d\n", x)
        final_cube = append(final_cube,
current_cube[x]) // from chosen x index using weighted func
    }

```

```

current_cube = []Cube{ }

// ===== Crossover =====
if isEven(population) {
    for j := 0; j < population; j += 2 {
        _ = j
        child1, child2 :=
crossoverTwoSlices(final_cube[j], final_cube[j+1])
        current_cube = append(current_cube,
child1)
        current_cube = append(current_cube,
child2)
    }
} else {
    for j := 0; j < population-1; j += 2 {
        _ = j
        child1, child2 :=
crossoverTwoSlices(final_cube[j], final_cube[j+1])
        current_cube = append(current_cube,
child1)
        current_cube = append(current_cube,
child2)
    }
    current_cube = append(current_cube,
final_cube[population-1])

    child1, child2 :=
crossoverTwoSlices(final_cube[population-1], final_cube[0])
    _ = child2
    current_cube = append(current_cube, child1)
}

// ===== Mutation =====
// mutation used: swap between 2 cube of 5*5*5
cubes
for j := range population {
    _ = j
    x1 := rand.IntN(5)
    y1 := rand.IntN(5)
    z1 := rand.IntN(5)

    x2 := rand.IntN(5)
    y2 := rand.IntN(5)
    z2 := rand.IntN(5)
    swapCubeOfCubes(current_cube[j], x1, x2, y1,
y2, z1, z2)
}

// Find the best value of final state before moving
to the next iteration

```

```

        for j := range population {
            _ = j
            obj_value_check :=
objective_function(current_cube[j])
            obj_value_sum += obj_value_check
            if obj_value_check < current_best_value {
                current_best_value = obj_value_check
            }
        }

        avg_objective_value = append(avg_objective_value,
obj_value_sum / population)

        objective_value_plot = append(objective_value_plot,
current_best_value)

        // Preparation to next iteration
        // Copy final into current
        // Free up final state
        final_cube = []Cube{}
        _ = final_cube
        final_cube = slices.Clone(current_cube)
    }

    // Put final best cube after all iterations finished
    for i := range population {
        _ = i

        obj_value_check :=
objective_function(final_cube[i])
        if obj_value_check < final_best_value {
            final_best_value = obj_value_check
            final_best_cube = final_cube[i]
        }
    }

    genetic_algorithm_result.initial_cube = initial_cube
    genetic_algorithm_result.initial_best_cube =
initial_best_cube
    genetic_algorithm_result.initial_best_value =
initial_best_value
    genetic_algorithm_result.final_cube = final_cube
    genetic_algorithm_result.final_best_cube =
final_best_cube
    genetic_algorithm_result.final_best_value =
final_best_value
    genetic_algorithm_result.objective_value_plot =
objective_value_plot
    genetic_algorithm_result.avg_objective_value =
avg_objective_value
    genetic_algorithm_result.population = population

```

```

genetic_algorithm_result.iteration = iteration
timeElapsed := time.Since(startTime)
genetic_algorithm_result.time =
int(timeElapsed.Milliseconds())
    return genetic_algorithm_result
}

```

c. Penjelasan *source code Genetic Algorithm*

- Inisialisasi

```

initial_cube := []Cube{}
initial_best_cube := Cube{}
initial_best_value := 999

```

- a. Membuat populasi awal (**initial_cube**).
- b. **initial_best_cube** dan **initial_best_value** digunakan untuk menyimpan individu dan nilai terbaik dari populasi awal.

- Generasi Awal Populasi

```

for i := range population {
    initial_cube = append(initial_cube,
generate_random_cube())
    obj_value_check := objective_function(initial_cube[i])
    obj_value_sum += obj_value_check
    if obj_value_check < initial_best_value {
        initial_best_value = obj_value_check
        initial_best_cube = initial_cube[i]
    }
}

```

- a. Menghasilkan populasi awal dengan **generate_random_cube()**.
- b. Mengevaluasi nilai objektif setiap individu dengan **objective_function()**.
- c. Menyimpan individu terbaik dari populasi awal.

- Persiapan Iterasi

```

current_cube = slices.Clone(initial_cube)

```

```
objective_value_plot = append(objective_value_plot,  
initial_best_value)
```

- a. Melakukan *deep copy* populasi awal ke **current_cube**.
- b. Menyimpan nilai terbaik dari populasi awal ke dalam **objective_value_plot**.

- Menghitung *Fitness Value*

```
fitness_value := []float64{ }  
for j := range population {  
    _ = j  
    n_objective_value = append(n_objective_value,  
objective_function(current_cube[j]))  
    fitness_value = append(fitness_value, 1.0 /  
float64(n_objective_value[j] + 1))  
}
```

- a. Menghitung nilai *fitness* dari setiap individu.
- b. *Fitness* dihitung sebagai $1 / (\text{nilai objektif} + 1)$ untuk menghindari nilai nol.

- Normalisasi *Fitness Value*

```
for _, value := range fitness_value {  
    fitness_sum += value  
}  
for j := range fitness_value {  
    fitness_value[j] /= fitness_sum  
}
```

Pada *loop* pertama, semua nilai *fitness* dari individu dalam populasi dijumlahkan,

- a. **fitness_value** → *array* yang berisi nilai *fitness* dari setiap individu dalam populasi.
- b. **fitness_sum** → total dari semua nilai *fitness*. Ini digunakan sebagai faktor pembagi dalam normalisasi.

Pada *loop* kedua, setiap nilai *fitness* dinormalisasi dengan membagi nilai *fitness* individu dengan **fitness_sum**,

- a. Normalisasi ini memastikan jumlah total dari semua nilai *fitness* menjadi 1.0 (100%). Hal ini diperlukan untuk metode seleksi berbasis probabilitas seperti *roulette wheel selection*.

- **Seleksi dengan `weightedRandomChoiceFloat`**

```
for j := range population {
    _ = j
    x := weightedRandomChoiceFloat(fitness_value)
    final_cube = append(final_cube, current_cube[x])
}
```

- Loop `for j := range population`* → iterasi sebanyak jumlah individu dalam populasi. Untuk setiap iterasi, satu individu akan dipilih menggunakan metode probabilistik.
- `x := weightedRandomChoiceFloat(fitness_value)`* → memilih indeks individu berdasarkan probabilitas terpilih.
- Fungsi *`weightedRandomChoiceFloat`* → menggunakan nilai *fitness* yang telah dinormalisasi sebagai *weight* untuk memilih individu. Indeks yang dipilih (*x*) memiliki peluang lebih besar jika nilai *fitness*-nya lebih tinggi.
- `final_cube = append(final_cube, current_cube[x])`* → menambahkan individu yang terpilih (*current_cube[x]*) ke populasi baru (*final_cube*).

- **Crossover**

```
if isEven(population) {
    for j := 0; j < population; j += 2 {
        child1, child2 := crossoverTwoSlices(final_cube[j],
final_cube[j+1])
        current_cube = append(current_cube, child1)
        current_cube = append(current_cube, child2)
    }
} else {
    for j := 0; j < population-1; j += 2 {
        _ = j
        child1, child2 := crossoverTwoSlices(final_cube[j],
final_cube[j+1])
        current_cube = append(current_cube, child1)
```

```

        current_cube = append(current_cube, child2)
    }
    current_cube = append(current_cube,
final_cube[population-1])

    child1, child2 :=
crossoverTwoSlices(final_cube[population-1], final_cube[0])
    _ = child2
    current_cube = append(current_cube, child1)
}

```

- a. Jika jumlah populasi genap, dilakukan *crossover* secara berpasangan menggunakan fungsi **crossoverTwoSlices**. Hasil *crossover* menghasilkan dua individu baru (**child1** dan **child2**) yang ditambahkan ke dalam **current_cube**.
- b. Jika jumlah populasi ganjil, pasangan individu berurutan di-*crossover* seperti pada kondisi genap hingga individu terakhir. Individu terakhir di-*crossover* dengan individu pertama (**final_cube[population-1]** dengan **final_cube[0]**) untuk memastikan tidak ada individu yang tertinggal.
- c. Setelah *crossover* dilakukan, **current_cube** berisi populasi baru yang siap digunakan untuk generasi berikutnya dalam algoritma.
- d. Proses ini memastikan individu dengan nilai *fitness* tinggi memiliki peluang lebih besar untuk terpilih, tetapi individu dengan nilai *fitness* rendah tetap memiliki kesempatan untuk dipilih, menjaga keragaman populasi.

- ***Mutation***

```

for j := range population {
    x1 := rand.IntN(5)
    y1 := rand.IntN(5)
    z1 := rand.IntN(5)

    x2 := rand.IntN(5)
    y2 := rand.IntN(5)
    z2 := rand.IntN(5)
    swapCubeOfCubes(current_cube[j], x1, x2, y1, y2, z1, z2)
}

```

- a. Loop `for j := range population` digunakan untuk melakukan mutasi pada setiap individu dalam populasi (`current_cube[j]`).
- b. x_1, y_1, z_1 dan x_2, y_2, z_2 dipilih secara acak menggunakan `rand.IntN(5)`, yang menghasilkan angka acak antara 0 hingga 4 (sesuai dengan ukuran $5 \times 5 \times 5$ dari *cube*).
- c. `swapCubeOfCubes(current_cube[j], x1, x2, y1, y2, z1, z2) →` melakukan pertukaran dua elemen dalam *cube* pada posisi (x_1, y_1, z_1) dan (x_2, y_2, z_2) .
- d. Proses ini memperkenalkan perubahan kecil (mutasi) pada individu, yang bertujuan untuk mencegah algoritma terjebak pada *local optima*.

- **Evaluasi Nilai Terbaik dalam Iterasi**

```

for j := range population {
    _ = j
    obj_value_check := objective_function(current_cube[j])
    obj_value_sum += obj_value_check
    if obj_value_check < current_best_value {
        current_best_value = obj_value_check
    }
}

avg_objective_value = append(avg_objective_value,
obj_value_sum / population)

objective_value_plot = append(objective_value_plot,
current_best_value)

```

- a. Loop melalui setiap individu dalam populasi dan hitung nilai fungsi objektifnya dengan `objective_function()`. Jumlahkan nilai *objective function* untuk menghitung rata-rata populasi.
- b. Bandingkan nilai fungsi objektif setiap individu dengan nilai terbaik saat ini (`current_best_value`). Jika nilai individu lebih kecil, perbarui `current_best_value`.
- c. Hitung rata-rata nilai *objective function* populasi dan simpan pada `avg_objective_value`.
- d. Simpan nilai `current_best_value` ke dalam `objective_value_plot`

- Persiapan Iterasi Berikutnya

```
final_cube = slices.Clone(current_cube)
```

Melakukan *deep copy* dari `current_cube` ke `final_cube` untuk persiapan iterasi berikutnya.

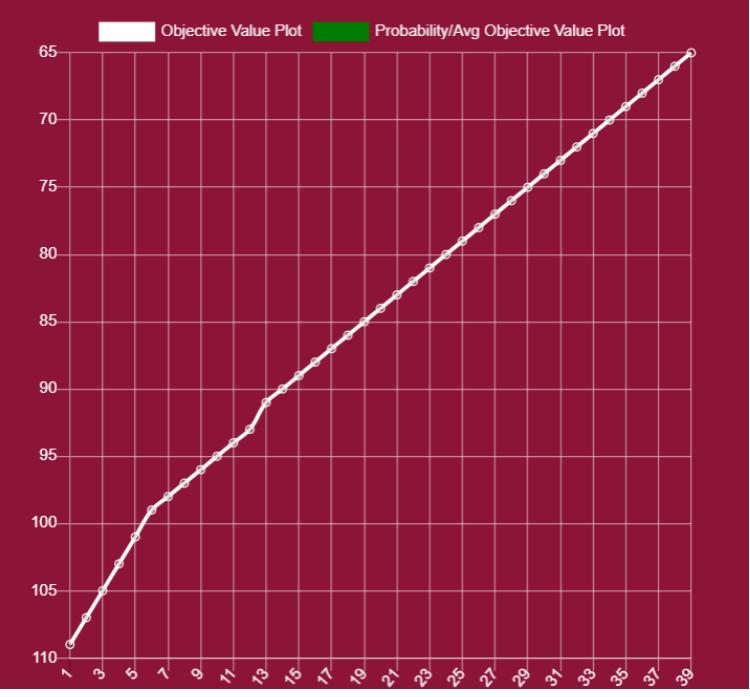
- Hasil Akhir

```
for i := range population {
    obj_value_check := objective_function(final_cube[i])
    if obj_value_check < final_best_value {
        final_best_value = obj_value_check
        final_best_cube = final_cube[i]
    }
}
```

- a. `for i := range population` → Loop untuk mengevaluasi setiap individu `final_cube[i]` dalam populasi akhir.
- b. `obj_value_check := objective_function(final_cube[i])` → menghitung nilai objektif dari individu saat ini menggunakan *objective function*.
- c. Jika nilai objektif dari individu saat ini (`obj_value_check`) lebih kecil dari `final_best_value`, maka `final_best_value` diperbarui dengan nilai baru (`obj_value_check`) dan `final_best_cube` diperbarui dengan individu saat ini (`final_cube[i]`).
- d. Setelah loop selesai, `final_best_value` dan `final_best_cube` mengandung nilai objektif terbaik dan individu terbaik dari populasi akhir.

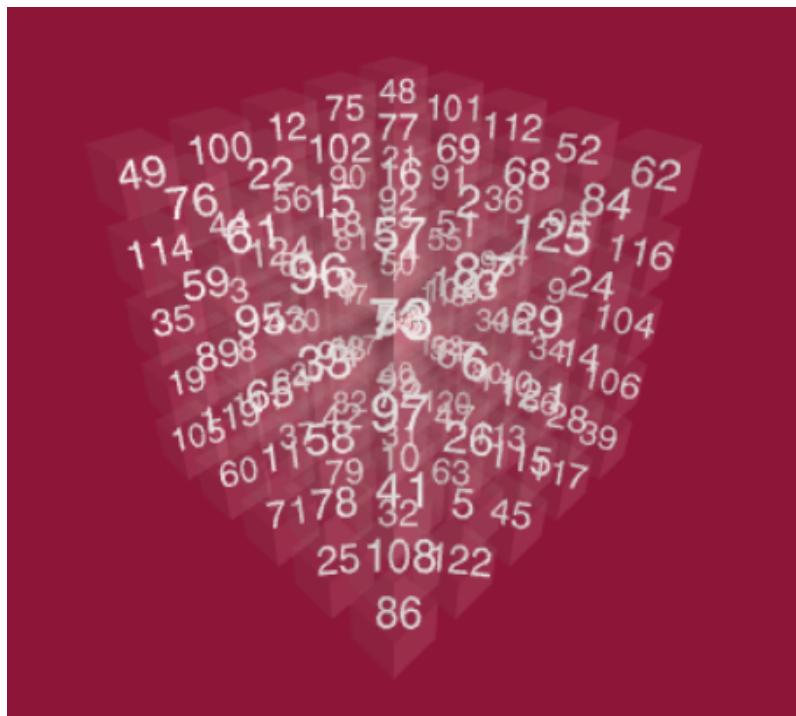
2.3 Hasil Eksperimen dan Analisis

2.3.1 Steepest Ascent Hill-Climbing

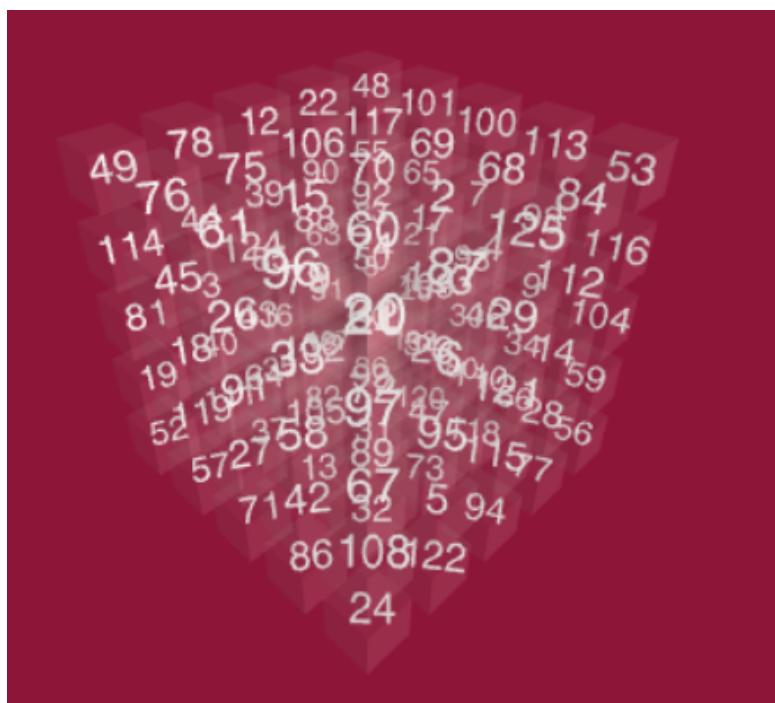
| Run | Final Objective Value | Iterasi | Durasi (ms) | Plot Objective Value |
|-----|-----------------------|---------|-------------|---|
| 1 | 109 → 65 | 38 | 2128 |  <p>Gambar 2.1 Testing 1 Algoritma Steepest Ascent Hill-Climbing</p> |

| | | | | |
|---|----------------------|----|------|---|
| 2 | $109 \rightarrow 74$ | 30 | 1701 | <p>Gambar 2.2 Testing 2 Algoritma Steepest Ascent Hill-Climbing</p> |
| 3 | $109 \rightarrow 69$ | 35 | 1968 | <p>Gambar 2.3 Testing 3 Algoritma Steepest Ascent Hill-Climbing</p> |

Berikut adalah tampilan visualisasi *initial state* dan *final state* untuk salah satu *run*, yaitu *run 1*:



Gambar 2.4 Visualisasi *Initial State* Steepest Ascent Hill-Climbing



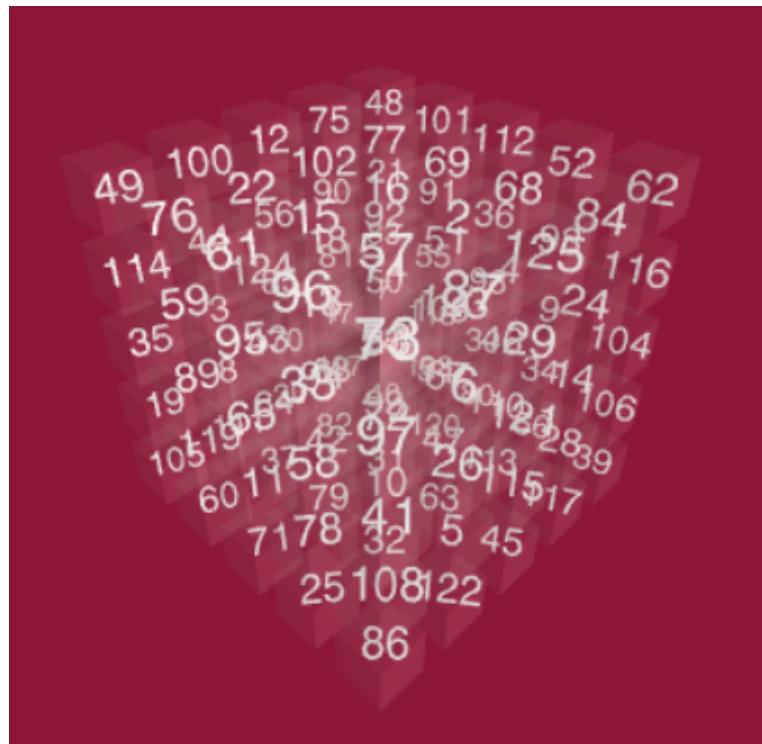
Gambar 2.5 Visualisasi *Final State* Steepest Ascent Hill-Climbing

2.3.2 Hill-Climbing with Sideways Move

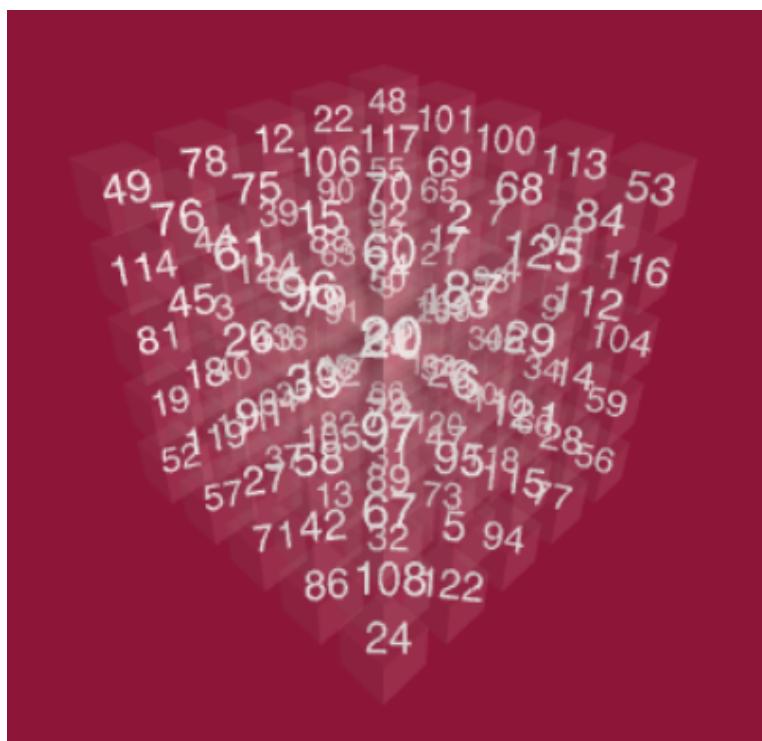
| <i>Run</i> | <i>Final Objective Value</i> | Iterasi | Durasi (ms) | <i>Max Sideways</i> | <i>Plot Objective Value</i> |
|------------|------------------------------|---------|-------------|---------------------|--|
| 1 | 109 → 65 | 48 | 2675 | 10 | <p>Gambar 2.6 Testing 1 Algoritma Hill-Climbing with Sideways Move</p> |

| | | | | | |
|---|----------------------|----|------|----|--|
| 2 | $108 \rightarrow 64$ | 50 | 2772 | 10 | <p>Gambar 2.7 Testing 2 Algoritma Hill-Climbing with Sideways Move</p> |
| 3 | $108 \rightarrow 68$ | 45 | 2559 | 10 | <p>Gambar 2.8 Testing 3 Algoritma Hill-Climbing with Sideways Move</p> |

Berikut adalah tampilan visualisasi *initial state* dan *final state* untuk salah satu *run*, yaitu *run 1*:

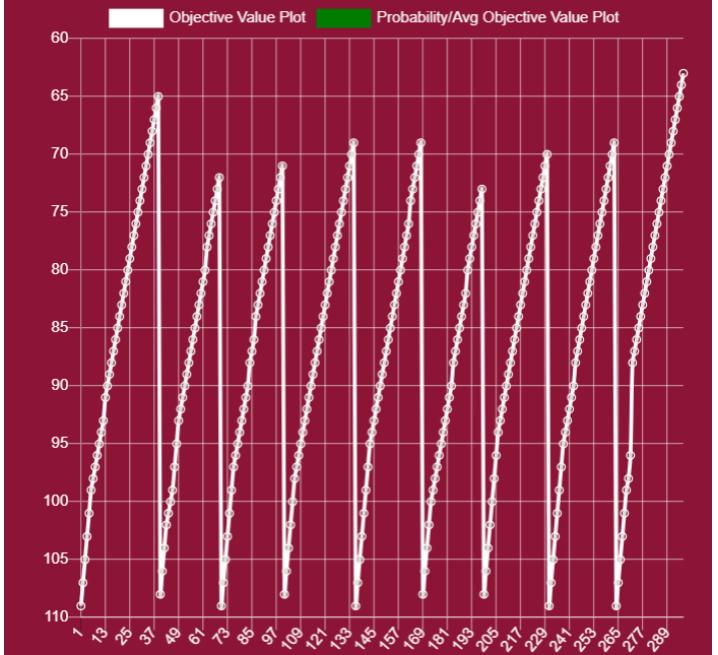


Gambar 2.9 Visualisasi *Initial State* Hill-Climbing with Sideways Move



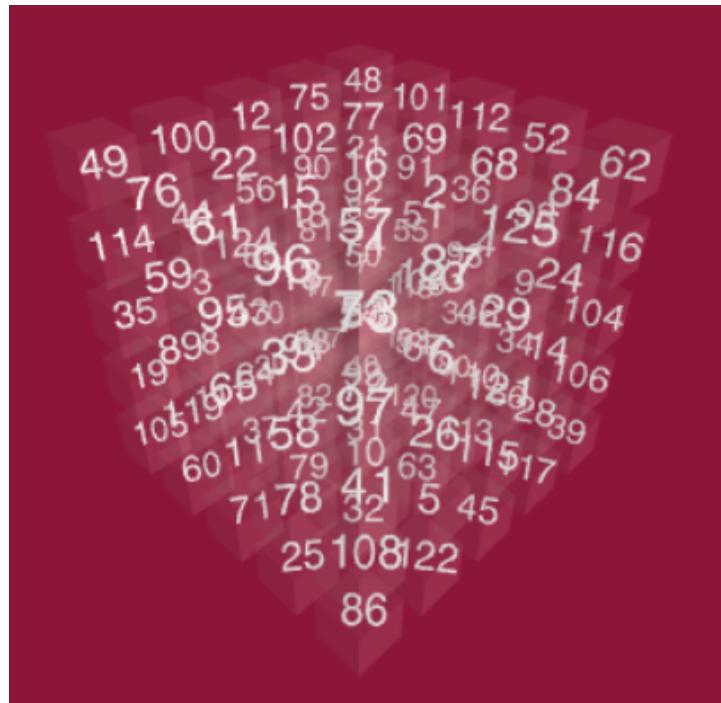
Gambar 2.10 Visualisasi *Final State* Hill-Climbing with Sideways Move

2.3.3 Random Restart Hill-Climbing

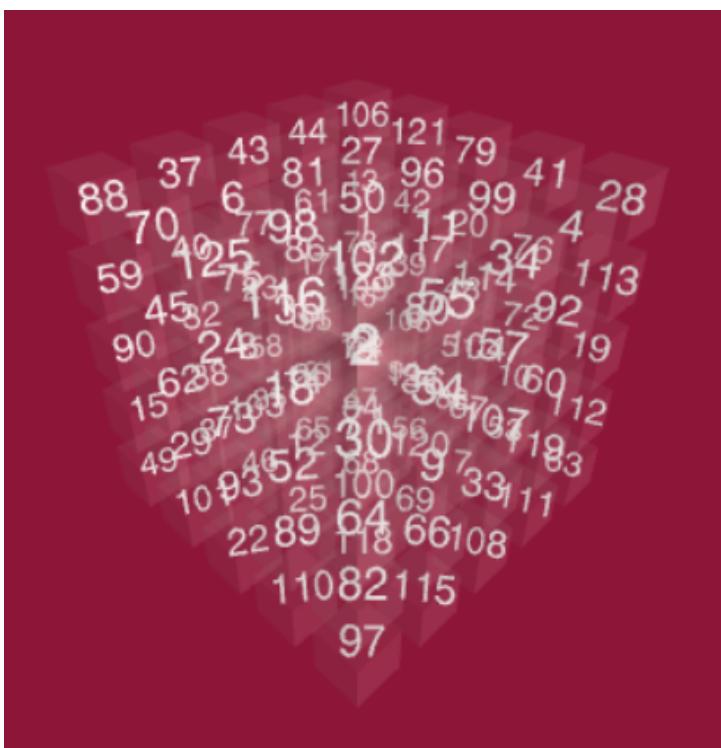
| Run | <i>Final Objective Value</i> | Durasi (ms) | <i>Max Restarts</i> | <i>Restart</i> | <i>Plot Objective Value</i> |
|-----|------------------------------|-------------|---------------------|----------------|--|
| 1 | 63 | 20000 | 10 | 10 |  <p>Iterasi: 38, 29, 30, 34, 32, 29, 31, 32, 30, 30, 39</p> <p>Gambar 2.11 Testing 1 Algoritma Random Restart Hill-Climbing</p> |

| | | | | | | |
|---|----|-------|----|----|---|--|
| 2 | 70 | 19839 | 10 | 10 | <p>Iterasi: 34, 29, 32, 31, 35, 33, 30, 37, 34, 28, 32</p> | |
| 3 | 71 | 18418 | 10 | 10 | <p>Iterasi: 30, 33, 27, 33, 31, 23, 24, 31, 28, 32, 28</p> | |

Berikut adalah tampilan visualisasi *initial state* dan *final state* untuk salah satu *run*, yaitu *run 1*:

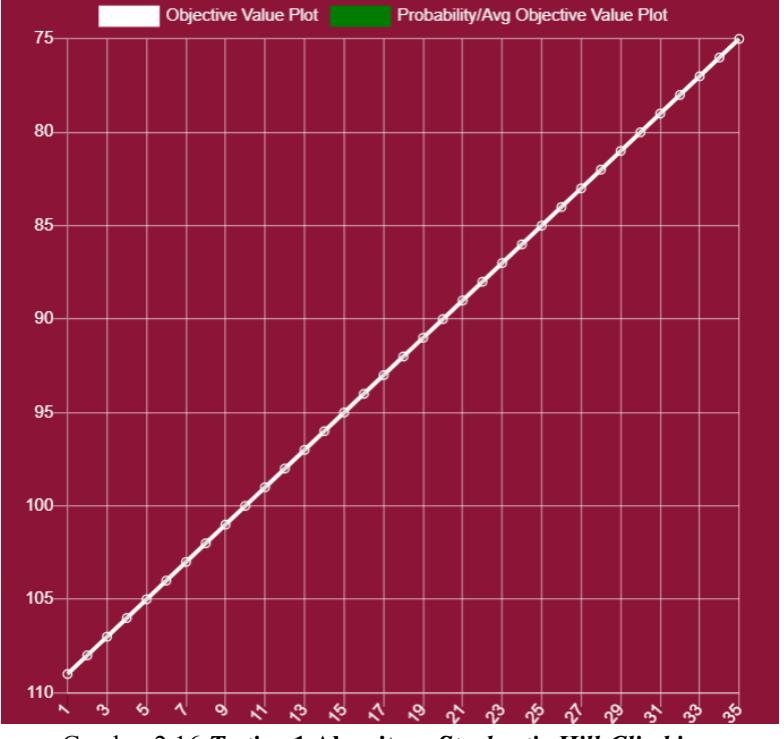


Gambar 2.14 Visualisasi *Initial State* Random Restart Hill-Climbing



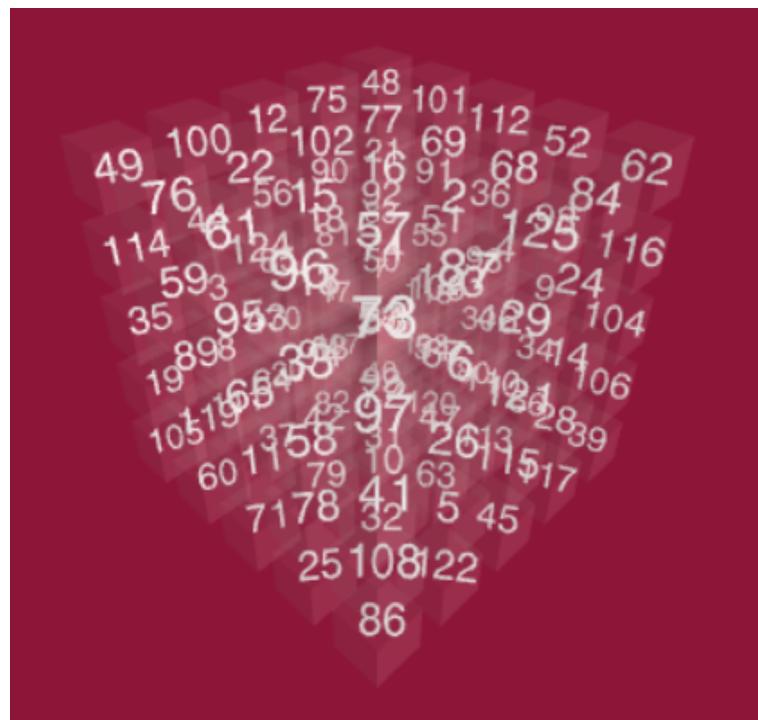
Gambar 2.15 Visualisasi *Final State* Random Restart Hill-Climbing

2.3.4 Stochastic Hill-Climbing

| <i>Run</i> | <i>Final Objective Value</i> | <i>Iterasi</i> | <i>Durasi (ms)</i> | <i>Max Iterations</i> | <i>Plot Objective Value</i> |
|------------|------------------------------|----------------|--------------------|-----------------------|--|
| 1 | 109 → 75 | 10000 | 68 | 10000 |  <p>Gambar 2.16 <i>Testing 1 Algoritma Stochastic Hill-Climbing</i></p> |

| | | | | | |
|---|----------------------|-------|----|-------|---|
| 2 | $109 \rightarrow 72$ | 10000 | 70 | 10000 | <p>Gambar 2.17 Testing 2 Algoritma Stochastic Hill-Climbing</p> |
| 3 | $109 \rightarrow 77$ | 10000 | 68 | 10000 | <p>Gambar 2.18 Testing 3 Algoritma Stochastic Hill-Climbing</p> |

Berikut adalah tampilan visualisasi *initial state* dan *final state* untuk salah satu *run*, yaitu *run 1*:

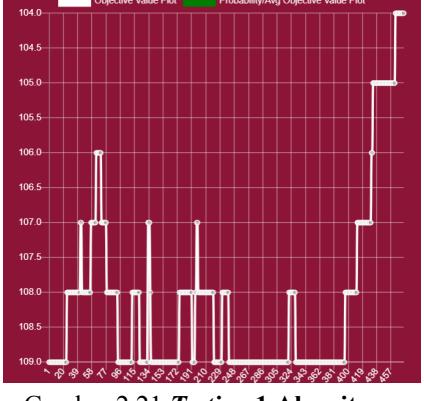
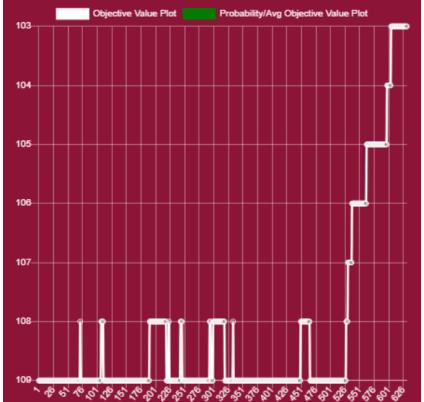
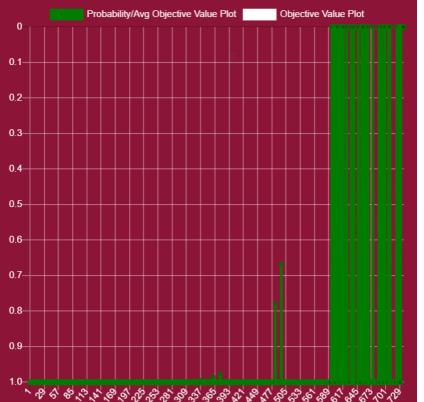


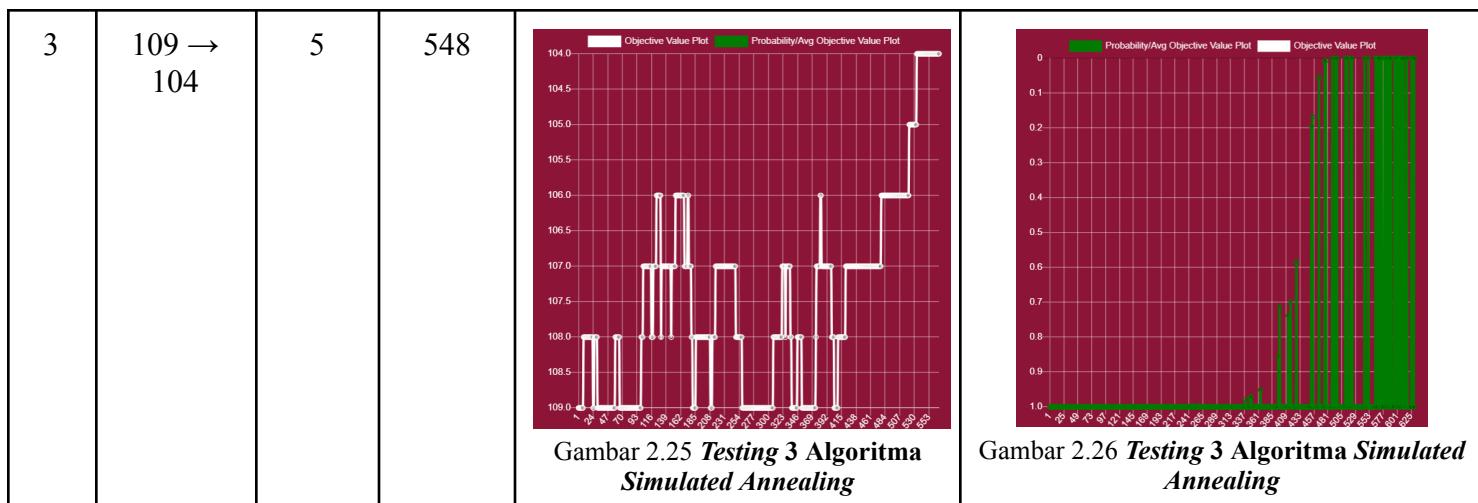
Gambar 2.19 Visualisasi *Initial State* Stochastic Hill-Climbing



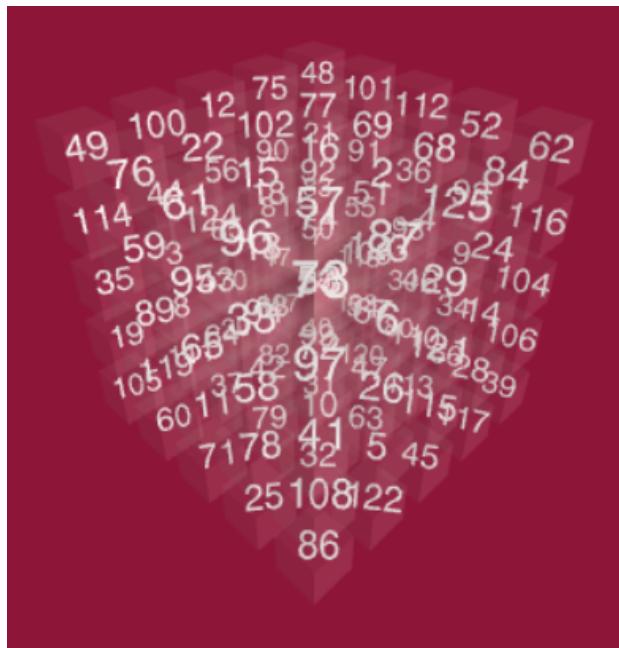
Gambar 2.20 Visualisasi *Final State* Stochastic Hill-Climbing

2.3.5 Simulated Annealing

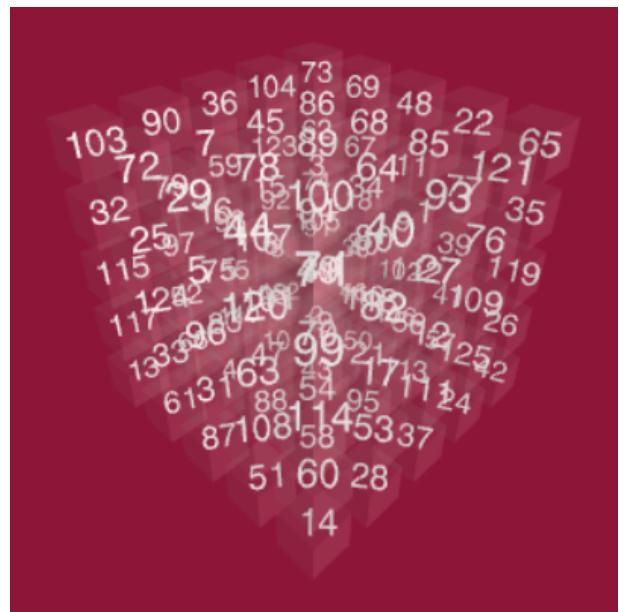
| Run | Final Objective Value | Durasi (ms) | Jumlah Stuck | Plot Objective Value | Plot Probability |
|-----|-----------------------|-------------|--------------|--|---|
| 1 | 109 → 104 | 5 | 568 |  <p>Gambar 2.21 Testing 1 Algoritma Simulated Annealing</p> |  <p>Gambar 2.22 Testing 1 Algoritma Simulated Annealing</p> |
| 2 | 109 → 103 | 5 | 558 |  <p>Gambar 2.23 Testing 2 Algoritma Simulated Annealing</p> |  <p>Gambar 2.24 Testing 2 Algoritma Simulated Annealing</p> |



Berikut adalah tampilan visualisasi *initial state* dan *final state* untuk salah satu *run*, yaitu *run 1*:

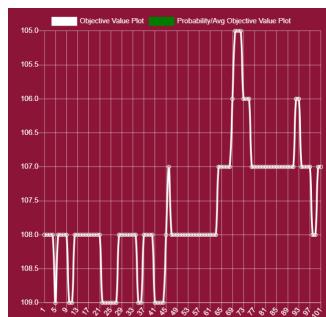
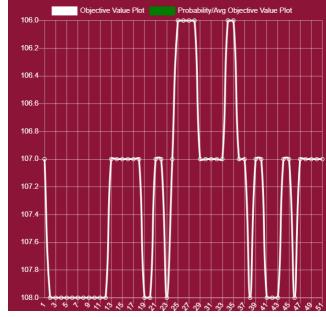
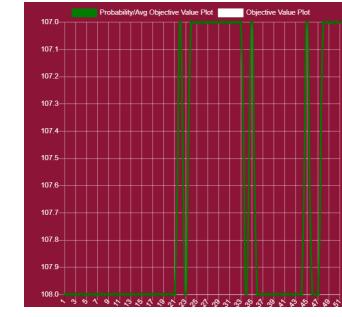


Gambar 2.27 Visualisasi *Initial State Simulated Annealing*



Gambar 2.28 Visualisasi *Final State Simulated Annealing*

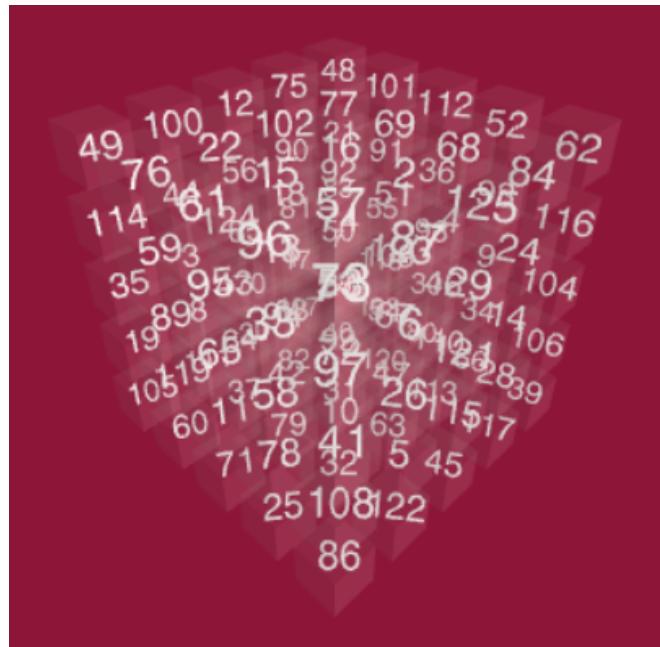
2.3.6 Genetic Algorithm

| Run | <i>Final Objective Value</i> | Durasi (ms) | Populasi | Iterasi | Plot <i>Objective Value</i> | Plot <i>Avg Objective Value</i> |
|-----|------------------------------|-------------|----------|---------|--|---|
| 1 | 108 → 107 | 11 | 10 | 100 |  <p>Gambar 2.29 Testing 1 Plot Objective Value Genetic Algorithm</p> |  <p>Gambar 2.30 Testing 1 Plot Avg Objective Value Genetic Algorithm</p> |
| 2 | 107 → 107 | 6 | 10 | 50 |  <p>Gambar 2.31 Testing 2 Plot Objective Value Genetic Algorithm</p> |  <p>Gambar 2.32 Testing 2 Plot Avg Objective Value Genetic Algorithm</p> |

| | | | | | | |
|---|-----------------------|---|-----|----|---|---|
| 3 | $107 \rightarrow 109$ | 4 | 10 | 25 | <p>Gambar 2.33 Testing 3 Plot Objective Value Genetic Algorithm</p> | <p>Gambar 2.34 Testing 3 Plot Avg Objective Value Genetic Algorithm</p> |
| 4 | $106 \rightarrow 104$ | 8 | 100 | 10 | <p>Gambar 2.35 Testing 4 Plot Objective Value Genetic Algorithm</p> | <p>Gambar 2.36 Testing 4 Plot Avg Objective Value Genetic Algorithm</p> |
| 5 | $106 \rightarrow 106$ | 8 | 50 | 10 | <p>Gambar 2.37 Testing 5 Plot Objective Value Genetic Algorithm</p> | <p>Gambar 2.38 Testing 5 Plot Avg Objective Value Genetic Algorithm</p> |

| | | | | | | |
|---|-----------------------|---|----|----|--|--|
| 6 | $106 \rightarrow 108$ | 4 | 25 | 10 | <p>Gambar 2.39 Testing 6 Plot Objective Value Genetic Algorithm</p> | <p>Gambar 2.40 Testing 6 Plot Avg Objective Value Genetic Algorithm</p> |
|---|-----------------------|---|----|----|--|--|

Berikut adalah tampilan visualisasi *initial state* dan *final state* untuk salah satu *run*, yaitu *run* 1:



Gambar 2.41 Visualisasi *Initial State Genetic Algorithm*



Gambar 2.42 Visualisasi *Final State Genetic Algorithm*

2.3.7 Analisis Perbandingan Algoritma dalam Mendekati *Global Optima*

Pendekatan algoritma *local search* dan metaheuristik, seperti *Steepest Ascent Hill-Climbing*, *Hill-Climbing with Sideways Move*, *Random Restart Hill-Climbing*, *Stochastic Hill-Climbing*, *Simulated Annealing*, dan *Genetic Algorithm*, menunjukkan hasil yang berbeda dalam upaya mendekati *global optima* pada permasalahan *Magic Cube*.

Steepest Ascent Hill-Climbing dan *Hill-Climbing with Sideways Move* cenderung **mencapai *local optima* dengan cepat**, namun memiliki keterbatasan dalam kemampuan eksplorasi sehingga mudah terjebak pada *local optima*, terutama tanpa adanya *restart* atau *sideways move* yang memadai. Algoritma *Hill-Climbing* yang dilengkapi dengan mekanisme *Random Restart* menunjukkan peningkatan performa dalam eksplorasi, memungkinkan algoritma untuk meninggalkan *local optima* dan mencoba solusi baru dari titik awal yang berbeda. Hal ini membuat *Random Restart Hill-Climbing* lebih efektif dalam **menjangkau solusi yang lebih baik dibandingkan metode *hill-climbing* tanpa *restart***. *Stochastic Hill-Climbing* juga menawarkan **fleksibilitas lebih tinggi** dalam eksplorasi, dengan memilih tetangga acak di setiap iterasi. Namun, metode ini sangat bergantung pada keberuntungan pemilihan tetangga, yang dapat menyebabkan hasilnya tidak konsisten dalam mendekati *global optima*.

Di sisi lain, algoritma metaheuristik seperti *Simulated Annealing* dan *Genetic Algorithm* menunjukkan **fleksibilitas dan kemampuan eksplorasi** yang lebih tinggi. *Simulated Annealing* secara khusus dirancang untuk menghindari jebakan *local optima* dengan menurunkan probabilitas penerimaan solusi yang kurang optimal seiring waktu, yang memungkinkan eksplorasi ruang solusi yang lebih luas pada tahap awal. Pada percobaan, memang *final state* menggunakan algoritma *Simulated Annealing* menunjukkan *objective value* yang lebih rendah dibandingkan yang lainnya, hal ini terjadi karena parameter *initial temperature* yang digunakan sangat kecil. **Jika *initial temperature* dinaikkan, maka *simulated annealing* dapat mengalahkan semua algoritma lainnya.** *Genetic Algorithm*, yang menggunakan seleksi berbasis populasi dan mekanisme seperti *crossover* dan mutasi, menunjukkan hasil yang lebih bervariasi. Tetapi, pada

kasus magic cube 5x5x5 yang memiliki aturan dimana setiap angka harus *distinct*, *Genetic Algorithm* menjadi algoritma yang paling buruk karena *crossover*-nya yang sangat rentan akan menghasilkan anakan dengan angka terduplicat. Secara keseluruhan, meskipun algoritma-algoritma ini memiliki kelebihan dan kekurangan masing-masing, kombinasi atau hibridisasi antara metode-metode tersebut dapat menjadi solusi yang lebih efektif dalam mencapai *global optima* pada permasalahan optimasi yang kompleks seperti *Magic Cube*.

2.3.8 Analisis Perbandingan Hasil Pencarian Algoritma

Hasil pencarian dari algoritma *Steepest Ascent Hill-Climbing*, *Hill-Climbing with Sideways Move*, *Random Restart Hill-Climbing*, *Stochastic Hill-Climbing*, *Simulated Annealing*, dan *Genetic Algorithm* menunjukkan variasi kinerja dalam mendekati solusi optimal. Algoritma ***Steepest Ascent Hill-Climbing*** dan ***Hill-Climbing with Sideways Move*** cenderung cepat mencapai *local optima*, namun performanya terbatas pada kemampuan eksplorasi yang minim, yang mengakibatkan terjebak pada solusi suboptimal. ***Hill-Climbing dengan tambahan mekanisme Random Restart*** memperlihatkan peningkatan kinerja yang signifikan, karena dapat melepaskan algoritma dari jebakan *local optima* dengan memulai pencarian dari titik awal baru. Sementara itu, ***Stochastic Hill-Climbing***, meskipun lebih eksploratif, memiliki ketergantungan tinggi pada pemilihan tetangga acak, yang terkadang menghambat kemampuan algoritma untuk secara konsisten mendekati *global optima*.

Di sisi lain, algoritma metaheuristik seperti *Simulated Annealing* dan *Genetic Algorithm* menunjukkan hasil yang lebih variatif dalam menjangkau ruang solusi yang lebih luas. *Simulated Annealing* mampu menghindari jebakan *local optima* dengan mekanisme penerimaan solusi yang kurang optimal pada tahap awal, dan secara bertahap menurunkan toleransi terhadap solusi tersebut seiring berjalannya waktu. Hal ini memungkinkan algoritma untuk lebih fleksibel dalam eksplorasi solusi pada tahap awal, sehingga mendekati *global optima* dengan lebih efektif. ***Genetic Algorithm***, dengan seleksi berbasis populasi serta operasi *crossover* dan mutasi, memperlihatkan pencarian solusi yang lebih

bervariasi. Secara keseluruhan, meskipun setiap algoritma memiliki kekuatan dan kelemahan, algoritma metaheuristik menunjukkan kinerja yang lebih konsisten dan mendekati *global optima* dibandingkan dengan algoritma local search konvensional, terutama pada permasalahan optimasi yang kompleks seperti *Magic Cube*.

2.3.9 Analisis Perbandingan Durasi Pencarian Algoritma

Analisis durasi pencarian menunjukkan variasi yang signifikan di antara keenam algoritma yang digunakan dalam eksperimen. **Steepest Ascent Hill Climbing (SAHC)** memiliki durasi eksekusi paling cepat karena memilih tetangga terbaik secara langsung di setiap iterasi. Meskipun cepat, SAHC sering terjebak dalam *local optimum* dan kurang efektif ketika menghadapi masalah dengan banyak puncak lokal. **Hill Climbing with Sideways Move (HCSM)** memiliki durasi yang lebih lama dibandingkan SAHC, karena terdapat perpindahan lateral di area datar (*plateau*) yang menambah waktu pencarian. **Random Restart Hill Climbing (RRHC)** memiliki durasi pencarian lebih lama karena terdapat fase *restart* yang dilakukan ketika solusi tidak meningkat. **Stochastic Hill Climbing (SHC)** menawarkan durasi pencarian yang lebih bervariasi. Algoritma ini tidak mengevaluasi semua *neighbor* secara sistematis, melainkan memilih tetangga secara acak, sehingga dapat mengurangi durasi pada kasus tertentu, tetapi kurang efisien pada masalah yang lebih kompleks. **Simulated Annealing (SA)** memiliki durasi yang relatif panjang karena mengizinkan perpindahan ke solusi yang lebih buruk di awal pencarian, terutama ketika suhu masih tinggi. Strategi ini membantu algoritma menghindari jebakan *local optimum*, tetapi memerlukan waktu lebih lama karena adanya fase penurunan temperatur yang bertahap, walaupun dapat menjadi sangat cepat jika fungsi suhu mempercepat penurunan suhu. **Genetic Algorithm (GA)** memiliki durasi yang bergantung pada parameter iterasi dan populasi, jika parameter disamakan dengan algoritma yang lain maka algoritma ini menjadi yang paling lama di antara semua algoritma yang diuji karena melibatkan proses evolusi melalui seleksi, *crossover*, dan mutasi dalam populasi yang besar.

Oleh karena itu, **SAHC** merupakan algoritma dengan durasi pencarian tercepat namun kurang optimal karena banyaknya probabilitas terjebak dalam *local optimum*. Sebaliknya, **SA** dengan parameter yang tinggi dan menawarkan solusi yang lebih mendekati global optimum meskipun dengan waktu pencarian yang lebih lama. Untuk permasalahan kompleks, algoritma **SA** lebih disarankan meskipun memerlukan durasi lebih lama dibandingkan algoritma *Hill Climbing* yang rentan terjebak di *plateau*.

2.3.10 Analisis Perbandingan Konsistensi Algoritma

Analisis konsistensi dari keenam algoritma memperlihatkan beberapa algoritma menunjukkan performa yang stabil dalam menemukan solusi yang mendekati optimal, sementara yang lain cenderung lebih bervariasi dalam hasil akhirnya. **Steepest Ascent Hill Climbing** (**SAHC**) dan **Hill Climbing with Sideways Move** (**HCSM**) menunjukkan tingkat konsistensi tertinggi karena aturannya yang tidak fleksibel. Dengan *initial state* yang sama, kedua algoritma tersebut akan menghasilkan hasil yang selalu sama. **Random Restart Hill Climbing** (**RRHC**) menunjukkan konsistensi yang sedikit menurun karena bergantung dengan parameter banyaknya restart yang diperbolehkan. Ketiga algoritma (**SAHC**, **HSCM**, **RRHC**) sama-sama mencari *neighbor* terbaik dari seluruh suksesor yang dibangkitkan, sehingga ketiga algoritma cenderung menghasilkan hasil yang selalu sama untuk *initial state* yang sama. **Stochastic Hill Climbing** (**SHC**) memiliki tingkat konsistensi hasil yang lebih rendah dibandingkan metode *Hill Climbing* lainnya. Hal ini karena pemilihan acak *neighbor* menimbulkan ketidakpastian dalam jalur pencarian solusi. **Simulated Annealing** (**SA**) memberikan konsistensi yang lebih baik daripada SHC, karena mekanisme penerimaan solusi yang lebih buruk di awal pencarian memungkinkan SA untuk mencapai *optima* yang lebih baik daripada SHC. **Genetic Algorithm** (**GA**) memiliki konsistensi yang bergantung terhadap implementasinya, biasanya memiliki konsistensi yang rendah karena adanya proses seleksi, *crossover*, dan mutasi yang berkelanjutan.

Oleh karena itu, **SAHC**, **HSCM**, **RRHC** merupakan algoritma yang paling konsisten dalam menghasilkan solusi yang stabil diantara percobaan yang berbeda.

2.3.11 Analisis Variasi Parameter pada *Genetic Algorithm*

Analisis variasi parameter pada *Genetic Algorithm* (GA) dilakukan dengan mempelajari dampak perubahan beberapa parameter kunci, yaitu ukuran populasi, jumlah iterasi, tingkat *crossover*, dan tingkat mutasi, terhadap performa algoritma dalam mencapai solusi yang optimal. Parameter ini sangat mempengaruhi efektivitas dan efisiensi pencarian solusi oleh GA. Pertama, ukuran populasi menentukan jumlah individu dalam satu generasi. Hasil eksperimen menunjukkan ukuran populasi yang lebih besar memberikan keragaman genetik yang lebih baik, sehingga meningkatkan peluang algoritma untuk menemukan solusi yang lebih optimal. Namun, populasi yang terlalu besar memperlambat waktu eksekusi karena waktu yang dibutuhkan untuk melakukan seleksi, *crossover*, dan mutasi meningkat secara signifikan. Selanjutnya, jumlah iterasi yang lebih tinggi memberikan lebih banyak kesempatan bagi algoritma untuk berevolusi, sehingga dapat menghasilkan solusi yang lebih baik. Namun, peningkatan iterasi di luar batas tertentu hanya memberikan sedikit perbaikan terhadap kualitas solusi, sementara durasi eksekusi meningkat secara linear. Oleh karena itu, penting untuk menetapkan jumlah iterasi yang seimbang antara kualitas solusi dan durasi eksekusi. Tingkat *crossover* juga memainkan peran penting dalam kombinasi genetik dari individu-individu terpilih. Dalam eksperimen, tingkat *crossover* yang tinggi (misalnya, di atas 80%) menunjukkan hasil yang baik karena memungkinkan pencampuran informasi genetik yang lebih banyak, meningkatkan eksplorasi solusi baru. Namun, tingkat crossover yang terlalu tinggi dapat menyebabkan hilangnya solusi yang baik karena memutus rantai gen yang sudah optimal.

Tingkat mutasi mempengaruhi eksplorasi algoritma dengan memperkenalkan perubahan acak pada individu. Hasil eksperimen menunjukkan mutasi yang rendah cenderung mengurangi kemampuan algoritma untuk keluar dari local optimum, sedangkan mutasi yang terlalu tinggi mengakibatkan

hilangnya solusi yang sudah baik akibat terlalu banyak perubahan acak. Tingkat mutasi optimal berkisar antara 1% hingga 5%, menciptakan keseimbangan antara eksplorasi dan eksploitasi. Secara keseluruhan, performa GA sangat dipengaruhi oleh penentuan parameter seperti ukuran populasi, jumlah iterasi, tingkat *crossover*, dan mutasi. Ukuran populasi yang cukup besar dan iterasi yang moderat memberikan hasil yang optimal tanpa memperlambat eksekusi, sedangkan tingkat *crossover* yang tinggi memperluas eksplorasi solusi baru. Tingkat mutasi yang rendah hingga moderat efektif dalam menghindari jebakan local optimum tanpa mengorbankan stabilitas solusi. Oleh karena itu, penyesuaian parameter melalui uji coba berulang sangat penting agar algoritma dapat mencapai hasil yang optimal sesuai dengan karakteristik masalah yang dihadapi.

BAB III

KESIMPULAN DAN SARAN

3.1 Kesimpulan

Diagonal magic cube berukuran $5 \times 5 \times 5$ adalah permasalahan optimasi yang bertujuan menyusun angka-angka dari 1 hingga 125 sehingga total penjumlahan elemen pada setiap baris, kolom, pilar, dan diagonal sama dengan nilai *magic constant* 315. Dalam menyelesaikan permasalahan ini, beberapa algoritma *local search* telah diterapkan, termasuk *Steepest Ascent Hill-Climbing*, *Hill-Climbing with Sideways Move*, *Stochastic Hill-Climbing*, *Random Restart Hill-Climbing*, *Simulated Annealing*, dan *Genetic Algorithm*. Setiap algoritma memiliki pendekatan unik dalam memperbaiki solusi secara bertahap dan mampu mencapai *local optima*. Meskipun algoritma *local search* seringkali berpotensi terjebak di *local optima*, teknik seperti *random restart* dan eksplorasi berbasis probabilitas pada algoritma metaheuristik seperti *simulated annealing* membantu memperluas jangkauan pencarian solusi untuk mendekati konfigurasi yang optimal.

Hasil percobaan menunjukkan bahwa algoritma-algoritma ini secara konsisten berhasil mendekati atau mencapai *local optima*, dengan kinerja terbaik yang bervariasi tergantung pada karakteristik dan mekanisme masing-masing algoritma. Algoritma metaheuristik seperti *Simulated Annealing* dan *Genetic Algorithm* menunjukkan keunggulan dalam mencapai solusi yang lebih mendekati *global optima*, berkat kemampuan algoritma-algoritma untuk menghindari jebakan *local optima* melalui mekanisme penerimaan solusi yang lebih adaptif dan eksplorasi berbasis populasi. Secara keseluruhan, kombinasi antara algoritma *local search* dan metaheuristik memberikan solusi yang lebih efektif dalam menyelesaikan permasalahan *diagonal magic cube* dan menunjukkan potensi untuk diterapkan pada permasalahan optimasi kompleks lainnya.

3.2 Saran

Peningkatan hasil pencarian solusi pada permasalahan *diagonal magic cube* dapat dicapai dengan memperbaiki metode evaluasi pada *objective function*. Hal ini bertujuan agar algoritma tidak mudah terjebak pada *local optima* dan lebih mampu mendekati *global optima*. Selain itu, penggunaan pendekatan *hybrid algorithm*, seperti kombinasi antara *Simulated Annealing* dan *Genetic Algorithm*, berpotensi memperkuat kemampuan eksplorasi solusi sehingga algoritma dapat menjangkau ruang solusi yang lebih luas. Eksperimen lebih lanjut dengan variasi ukuran *magic cube*, misalnya 3x3x3 atau 7x7x7, juga disarankan untuk menguji performa dan adaptabilitas algoritma terhadap skala permasalahan yang berbeda. Di samping itu, optimasi kecepatan eksekusi, khususnya pada proses *swap* dan evaluasi fungsi objektif, perlu dilakukan guna mempercepat waktu pencarian solusi, terutama ketika algoritma diterapkan pada masalah dengan kompleksitas yang lebih tinggi atau ukuran yang lebih besar.

Selain itu, kami juga menyarankan untuk melakukan *tuning* parameter algoritma lebih lanjut, seperti tingkat *crossover* dan mutasi pada *Genetic Algorithm*. Proses *tuning* ini dilakukan secara adaptif sehingga parameter dapat berubah sesuai dengan kondisi pencarian solusi. Implementasi metode *logging* dan visualisasi yang lebih mendetail pada setiap iterasi juga diharapkan dapat memberikan wawasan lebih mengenai performa algoritma, sehingga penyesuaian parameter dapat dilakukan dengan lebih tepat. Penggunaan teknik *parallel processing* juga disarankan untuk mempercepat waktu eksekusi, terutama pada ukuran *magic cube* yang lebih besar. Terakhir, penerapan metode *stopping criterion* yang lebih fleksibel, seperti *adaptive threshold* dapat membantu menghindari iterasi yang berlebihan dan mengurangi waktu komputasi tanpa mengorbankan kualitas solusi.

BAB IV

PEMBAGIAN TUGAS TIAP ANGGOTA KELOMPOK

Berikut adalah pembagian kerja penggerjaan Tugas Besar IF3070 Dasar Intelegensi Artifisial 1, yaitu:

| NIM | Nama | Bagian Pengerjaan |
|----------|----------------------|---|
| 18222023 | Thalita Zahra Sutejo | <ol style="list-style-type: none">1. Penggerjaan Laporan Bagian Deskripsi Persoalan2. Penggerjaan Laporan Bagian Pemilihan <i>Objective Function</i>3. Penggerjaan Laporan Bagian Penjelasan Implementasi 3 Algoritma <i>Local Search</i> (<i>Hill-Climbing with Sideways Move</i>, <i>Random Restart Hill-Climbing</i>, <i>Stochastic Hill-Climbing</i>)4. Penggerjaan Laporan Bagian Hasil Eksperimen dan Analisis (Bagian Analisis Perbandingan Algoritma dalam Mendekati <i>Global Optima</i> dan Bagian Analisis Perbandingan Hasil Pencarian Algoritma)5. Penggerjaan Bagian 3.1 Kesimpulan dan 3.2 Saran6. Penggerjaan README7. Finalisasi Laporan Tugas Besar |
| 18222056 | Irfan Musthofa | <ol style="list-style-type: none">1. Pembuatan <i>Source Code Genetic Algorithm</i>2. Finalisasi <i>Source Code Steepest Ascent Hill-Climbing</i>3. Finalisasi <i>Source Code Hill-Climbing with Sideways Move</i>4. Finalisasi <i>Source Code Random Restart Hill-Climbing</i>5. Finalisasi <i>Source Code Stochastic Hill-Climbing</i>6. Finalisasi <i>Source Code Simulated Annealing</i>7. Pembuatan <i>Front-End</i>8. Melakukan semua <i>Testing</i> dan <i>Debugging</i> |

| | | |
|-----------------|---------------------------|---|
| | | <p>9. Finalisasi Laporan Tugas Besar</p> |
| 18222059 | Eleanor Cordelia | <ol style="list-style-type: none"> 1. Pengeraaan Laporan Bagian Deskripsi Persoalan 2. Pengeraaan Laporan Bagian Pemilihan <i>Objective Function</i> 3. Pengeraaan Laporan Bagian Penjelasan Implementasi 3 Algoritma <i>Local Search (Steepest Ascent Hill-Climbing, Simulated Annealing, Genetic Algorithm)</i> 4. Pengeraaan Laporan Bagian Hasil Eksperimen dan Analisis (Analisis Perbandingan Durasi Pencarian Algoritma, Analisis Perbandingan Konsistensi Algoritma, Analisis Variasi Parameter pada <i>Genetic Algorithm</i>) 5. Pengeraaan Bagian 3.2 Saran 6. Pengeraaan README 7. Finalisasi Laporan Tugas Besar |
| 18222063 | Muhammad Faiz Atharrahman | <ol style="list-style-type: none"> 1. Pembuatan <i>Source Code Steepest Ascent Hill-Climbing</i> 2. Pembuatan <i>Source Code Hill-Climbing with Sideways Move</i> 3. Pembuatan <i>Source Code Random Restart Hill-Climbing</i> 4. Pembuatan <i>Source Code Stochastic Hill-Climbing</i> 5. Pembuatan <i>Source Code Simulated Annealing</i> 6. Pembuatan Visualisasi 3D dan <i>Chart</i> 7. Pembuatan <i>Front-End</i> 8. Pembuatan Plot Grafik dan Pengisian Plot <i>Objective Value</i> pada Laporan Tugas Besar |

REFERENSI

1. Russell, S., & Norvig, P. (2020). *Artificial intelligence: A modern approach* (4th ed.). Pearson.
2. GeeksforGeeks. (n.d.). *Introduction to hill climbing in artificial intelligence*. Retrieved October 1, 2024. Tersedia di <https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/>
3. Baeldung. (n.d.). *Simulated annealing*. Retrieved October 1, 2024. Tersedia di <https://www.baeldung.com/cs/simulated-annealing>
4. Ananda, M. P. (2019). *A new algorithm of hill climbing for solving the travelling salesman problem*. *Informatika*, 12(1), 19-25. Retrieved October 1, 2024. Tersedia di <https://ti.ukdw.ac.id/ojs/index.php/informatika/article/download/88/50>
5. Sari, D. K., & Syamsul, A. (2021). *Optimasi pencarian solusi menggunakan metode hill climbing untuk penjadwalan dosen*. *Pelita*, 10(2), 1-6. Retrieved October 1, 2024. Tersedia di <https://ejurnal.stmik-budidarma.ac.id/index.php/pelita/article/download/1843/1428>
6. Prasetyo, H., & Hapsari, D. (2021). *Pengembangan metode hill climbing dalam optimasi sistem informasi manajemen*. *Journal of Information Systems and Informatics Engineering*, 2(1), 75-81. Retrieved October 1, 2024. Tersedia di <https://ejurnal.pelitaindonesia.ac.id/ojs32/index.php/JOISIE/article/download/2122/905>
7. TutorialsPoint. (n.d.). *Genetic algorithms - Mutation*. TutorialsPoint. Retrieved October 2, 2024. Tersedia di https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm
8. Weisstein, E. W. (n.d.). Magic Cube. *Wolfram MathWorld*. from <https://mathworld.wolfram.com/MagicCube.html>.
9. Weisstein, E. W. (n.d.). Magic Constant. *Wolfram MathWorld*. from <https://mathworld.wolfram.com/MagicConstant.html>.