

Laporan Tugas Besar 1 IF3070 Dasar Inteligensi Artifisial

Pencarian Solusi Diagonal Magic Cube dengan Local Search

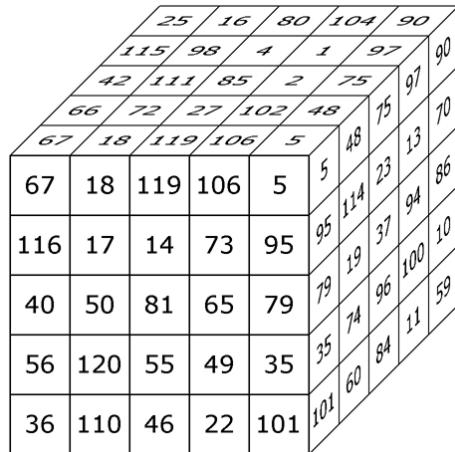
Disusun Oleh:

Matthew Lim / 18222005

Alessandro Jusack Hasian / 18222025

Satria Wisnu Wibowo / 18222087

Wisyendra Lunarmalam / 18222095



Program Studi Sistem dan Teknologi Informasi

Sekolah Teknik Elektro dan Informatika - Institut Teknologi Bandung

Jl. Ganesha 10, Bandung 40132

	Program Studi Sistem dan Teknologi Informasi STEI – ITB	Nomor Dokumen	Jumlah Halaman
		01	77

1. Deskripsi Persoalan

Magic Cube

Diagonal *magic cube* merupakan kubus yang tersusun dari angka 1 hingga n^3 tanpa pengulangan dengan n adalah panjang sisi pada kubus tersebut. Kubus baru dikatakan valid jika:

1. Penjumlahan *value* tiap unit pada satuan baris sama
2. Penjumlahan *value* tiap unit pada satuan kolom sama
3. Penjumlahan *value* tiap unit pada satuan tiang sama
4. Penjumlahan *value* tiap unit pada satuan diagonal baris sama
5. Penjumlahan *value* tiap unit pada satuan diagonal kolom sama
6. Penjumlahan *value* tiap unit pada satuan diagonal tiang sama
7. Penjumlahan *value* tiap unit pada satuan diagonal ruang sama

Jumlah yang sama disini disebut *magic number* dan terdapat total 109 rusuk kubus yang jika *value* tiap unit dalam rusuk tersebut dijumlahkan memiliki hasil yang sama dengan *magic number* tersebut.

2. Pembahasan

2.1 Pemilihan Objective Function

Objective function yang digunakan berupa jumlah banyaknya rusuk kubus yang mencapai *value* *magic number*. Kami menemukan bahwa *magic number* memiliki nilai 315 dengan alasan berikut.

Oleh karena semua bilangan pada kubus tersebut berurutan (1 - 125) dan tidak ada yang berulang, maka bisa diberi kesimpulan bahwa rata-rata bilangan sama dengan median dengan rumus:

$$\text{Rata-rata} = (n^3 + 1)/2$$

* $n = 5$ (jumlah sisi *magic cube*).

Untuk mendapatkan *magic number*, cukup menjumlahkan rata-rata bilangan sebanyak 5x sesuai (jumlah sisi *magic cube*) sehingga diperoleh:

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

. *Objective function* membuat 109 rusuk memiliki nilai penjumlahan sebesar 315. Target 109 rusuk diperoleh dari 25 kolom, 25 baris, 25 pilar, 30 diagonal, dan 4 triagonal. Jika ingin diformulasikan menjadi sebuah ekuasi:

$$\mathbf{h = \text{Number of edges where edge} \sum = 315}$$

Dengan target program untuk akan memaksimalkan angka tersebut menjadi 109 (*global maximum*).

2.2 Penjelasan Algoritma Local Search

Struct *Magic Cube*

Pada kode ini, terdapat dua struct yang digunakan yaitu *Block* dan *Cube* yang bersama-sama membentuk representasi sebuah kubus berukuran 5x5x5. Struktur ini, secara keseluruhan memungkinkan representasi sebuah kubus yang bisa digunakan dalam berbagai operasi seperti pengecekan konfigurasi blok, manipulasi posisi, nilai blok, atau evaluasi berdasarkan nilai *magic_number* dan *total_edges* yang telah ditentukan.

```
#define SIZE 5
#define TOTAL_VALUES (SIZE * SIZE * SIZE)
#define MAGIC_NUMBER 315
#define TOTAL_EDGES 109

typedef struct {
    uint8_t value;
    uint8_t pos;
} Block;

typedef struct {
    Block blocks[SIZE][SIZE][SIZE];
} Cube;
```

Fungsi Objective Function

Berikut merupakan fungsi *objective function* untuk menghitung nilai *heuristic* pada *cube*. Nilai *heuristic* yang dihitung berdasarkan jumlah deret angka di berbagai dimensi dan arah dalam kubus yang mencapai *magic number*. Setiap kali deret nilai pada baris, kolom, atau diagonal mencapai angka ini, nilai *heuristic* bertambah 1.

```
int calculate_heuristics(Cube *cube) {

    int heuristics = 0;

    // Check rows
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            int row_sum = 0;
            for (int k = 0; k < SIZE; k++) {
                row_sum += cube->blocks[i][j][k].value;
            }
            if (row_sum == MAGIC_NUMBER) heuristics++;
        }
    }

    // Check columns
    for (int i = 0; i < SIZE; i++) {
```

```

        for (int k = 0; k < SIZE; k++) {
            int col_sum = 0;
            for (int j = 0; j < SIZE; j++) {
                col_sum += cube->blocks[i][j][k].value;
            }
            if (col_sum == MAGIC_NUMBER) heuristics++;
        }
    }

    // Check depths
    for (int j = 0; j < SIZE; j++) {
        for (int k = 0; k < SIZE; k++) {
            int depth_sum = 0;
            for (int i = 0; i < SIZE; i++) {
                depth_sum += cube->blocks[i][j][k].value;
            }
            if (depth_sum == MAGIC_NUMBER) heuristics++;
        }
    }

    // Check main diagonals in each plane
    for (int i = 0; i < SIZE; i++) {
        int diag1_sum = 0, diag2_sum = 0;
        for (int j = 0; j < SIZE; j++) {
            diag1_sum += cube->blocks[i][j][j].value;
            diag2_sum += cube->blocks[i][j][SIZE - j - 1].value;
        }
        if (diag1_sum == MAGIC_NUMBER) heuristics++;
        if (diag2_sum == MAGIC_NUMBER) heuristics++;
    }

    for (int j = 0; j < SIZE; j++) {
        int diag1_sum = 0, diag2_sum = 0;
        for (int i = 0; i < SIZE; i++) {
            diag1_sum += cube->blocks[i][j][i].value;
            diag2_sum += cube->blocks[i][j][SIZE - i - 1].value;
        }
        if (diag1_sum == MAGIC_NUMBER) heuristics++;
        if (diag2_sum == MAGIC_NUMBER) heuristics++;
    }

    for (int k = 0; k < SIZE; k++) {
        int diag1_sum = 0, diag2_sum = 0;
        for (int i = 0; i < SIZE; i++) {
            diag1_sum += cube->blocks[i][i][k].value;
            diag2_sum += cube->blocks[i][SIZE - i - 1][k].value;
        }
        if (diag1_sum == MAGIC_NUMBER) heuristics++;
        if (diag2_sum == MAGIC_NUMBER) heuristics++;
    }
}

```

```
}

// Check space diagonals
int space_diag1_sum = 0, space_diag2_sum = 0, space_diag3_sum = 0, space_diag4_sum
= 0;
for (int i = 0; i < SIZE; i++) {
    space_diag1_sum += cube->blocks[i][i][i].value;
    space_diag2_sum += cube->blocks[i][i][SIZE - i - 1].value;
    space_diag3_sum += cube->blocks[i][SIZE - i - 1][i].value;
    space_diag4_sum += cube->blocks[SIZE - i - 1][i][i].value;
}
if (space_diag1_sum == MAGIC_NUMBER) heuristics++;
if (space_diag2_sum == MAGIC_NUMBER) heuristics++;
if (space_diag3_sum == MAGIC_NUMBER) heuristics++;
if (space_diag4_sum == MAGIC_NUMBER) heuristics++;

return heuristics;
}
```

Primitive Function / Procedure

Fungsi atau prosedur di cube.c atau fungsi-fungsi lain untuk keperluan generik.

1. Init_cube, menginisialisasi *cube* dengan nilai-nilai berurutan lalu mengacak posisi nilai-nilai tersebut.

```
function init_cube() returns Cube {  
    cube ← new Cube  
    for each block in cube do  
        block ← initialize block with default values  
    end for  
    return cube  
}
```

```
void init_cube(Cube *cube) {  
    uint8_t value = 1;  
    for (int i = 0; i < SIZE; i++) {  
        for (int j = 0; j < SIZE; j++) {  
            for (int k = 0; k < SIZE; k++) {  
                cube->blocks[i][j][k].pos = value - 1;  
                cube->blocks[i][j][k].value = value++;  
            }  
        }  
    }  
    shuffle_cube(cube);  
}
```

2. Copy_Cube, mengkopi cube dari argumen pertama ke cube argumen ke dua yang diterima oleh prosedur.

```
Procedure copy_cube(cube1,cube2) {  
    for each block in cube do  
        elemen  
    end for  
}
```

```
void copy_cube(Cube *og, Cube *fake) {  
    for (int i = 0; i < SIZE; i++) {  
        for (int j = 0; j < SIZE; j++) {  
            for (int k = 0; k < SIZE; k++) {  
                fake->blocks[i][j][k] = og->blocks[i][j][k];  
            }  
        }  
    }  
}
```

3. Display_cube, menampilkan keadaan *cube* ke layar dengan setiap blok atau sisi *cube* ditampilkan dengan atributnya untuk memungkinkan pengguna melihat susunan/konfigurasi saat ini.

```
function display_cube(cube) {
    for each block in cube do
        display block
    end for
}

for (int i = 0; i < SIZE; i++) {
    printf("Layer %d:\n", i + 1);
    for (int j = 0; j < SIZE; j++) {
        for (int k = 0; k < SIZE; k++) {
            printf("%2d ", cube->blocks[i][j][k].value);
        }
        printf("\n");
    }
    printf("\n");
}
```

4. Shuffle, mengacak posisi nilai-nilai dalam *array* blok yang merepresentasikan blok-blok *cube*.

```
function shuffle(array) {
    for i ← 0 to length of array - 1 do
        j ← random index between 0 and length of array - 1
        swap(array[i], array[j])
    end for
}
```

```
void shuffle(Block *array) {
    size_t n = TOTAL_VALUES;
    for (size_t i = 0; i < n - 1; i++) {
        size_t j = i + rand() / (RAND_MAX / (n - i) + 1);
        uint8_t t = array[i].value;
        uint8_t p = array[i].pos;

        array[i].value = array[j].value;
        array[i].pos = array[j].pos;

        array[j].value = t;
        array[j].pos = p;
```

```
    }
}
```

5. Shuffle_cube, mengacak posisi blok di dalam *cube* untuk membuat konfigurasi awal secara acak.

```
function shuffle_cube(cube) {
    shuffle(cube.blocks)
}
```

```
void shuffle_cube(Cube *cube) {
    srand(time(NULL));

    Block *flat_array = flatten_cube(cube);

    shuffle(flat_array);

    uint8_t index = 0;
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            for (int k = 0; k < SIZE; k++) {
                cube->blocks[i][j][k].value = flat_array[index++].value;
            }
        }
    }
    free(flat_array);
}
```

6. Swap, menukar nilai antara dua variabel bertipe *uint8_t*. Fungsi ini menggunakan pointer untuk mengakses variabel yang ingin ditukar secara langsung. Fungsi ini berguna untuk melakukan pertukaran nilai tanpa mengembalikan nilai secara eksplisit.

```
function swap(u1, u2)
    temp ← u1
```

```
    u1 ← u2
```

```
    u2 ← temp
```

```
end function
```

```

void swap(uint8_t *u1, uint8_t *u2) {
    uint8_t temp = *u1;
    *u1 = *u2;
    *u2 = temp;
}

```

7. Flatten,
function flatten_cube(cube) returns array of Blocks:

```

flat_array ← new array of size number of blocks in cube
index ← 0

for i from 0 to SIZE - 1 do
    for j from 0 to SIZE - 1 do
        for k from 0 to SIZE - 1 do
            flat_array[index].pos ← cube.blocks[i][j][k].pos
            flat_array[index].value ← cube.blocks[i][j][k].value
            index ← index + 1

return flat_array

```

```

Block *flatten_cube(Cube *cube) {
    Block *flat_array = (Block *)malloc(sizeof(Block) * (TOTAL_VALUES));

    size_t index = 0;
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            for (int k = 0; k < SIZE; k++) {
                flat_array[index].pos = cube->blocks[i][j][k].pos;
                flat_array[index].value = cube->blocks[i][j][k].value;
                index++;
            }
        }
    }
    return flat_array;
}

```

```
}
```

8. Unflatten 1,
Function unflatten_cube(flat_array):

```
cube ← new Cube
```

```
Set index to 0
```

```
for i from 0 to SIZE - 1:
```

```
    for j from 0 to SIZE - 1:
```

```
        for k from 0 to SIZE - 1:
```

```
            set cube.blocks[i][j][k].pos ← flat_array[index].pos
```

```
            set cube.blocks[i][j][k].value ← flat_array[index].value
```

```
index ← index + 1
```

```
return cube
```

```
Cube *unflatten_cube(Block *flat_array) {
    Cube *cube = (Cube *)malloc(sizeof(Cube));

    size_t index = 0;
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            for (int k = 0; k < SIZE; k++) {
                cube->blocks[i][j][k].pos = flat_array[index].pos;
                cube->blocks[i][j][k].value = flat_array[index].value;
                index++;
            }
        }
    }
}
```

```
    return cube;
}
```

9. Unflatten 2,
function unflatten_cube2(flat_array) returns Cube:

```
cube ← new Cube
```

```
index ← 0
```

```
for i from 0 to SIZE - 1 do
    for j from 0 to SIZE - 1 do
        for k from 0 to SIZE - 1 do
            cube.blocks[i][j][k].value ← flat_array[index]
            index ← index + 1
```

```
return cube
```

```
Cube *cube = (Cube *)malloc(sizeof(Cube));

size_t index = 0;
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
        for (int k = 0; k < SIZE; k++) {
            cube->blocks[i][j][k].value = *flat_array[index];
            index++;
        }
    }
}
return cube;
```

1. Steepest Ascent Hill-Climbing

Pada **steepest ascent hill-climbing** (sach), algoritma mencari solusi terbaik dengan mengevaluasi semua kemungkinan tetangga (*neighbors*) dari kondisi saat ini dan memilih tetangga dengan peningkatan terbesar dalam nilai heuristik (pada konteks *objective function* kami, kubus dengan rusuk *magic number* terbanyak).

Jika tidak ada peningkatan yang ditemukan, algoritma berhenti di *local maximum*, karena tidak ada solusi yang lebih baik di sekitar tetangga tersebut.

Kebutuhan komputasi akan relatif tinggi karena algoritma ini harus mengevaluasi semua tetangga dalam setiap langkahnya. Pada kasus seperti *magic cube*, di mana ada banyak kombinasi pasangan nilai yang dapat ditukar, hal ini bisa menjadi sangat mahal dari segi komputasi.

Berikut adalah deskripsi fungsi/kelas dari algoritma,

function sahc (*initial state of cube*) returns state with highest value {

current \leftarrow *initial state of cube*

while *current.value* < **highest possible h** do
 best \leftarrow *current*

block 1, block 2 $\leftarrow 0$

while *block 1* < **number of blocks in cube** do
 block 2 \leftarrow *block 1* + 1
 while *block 2* < **number of blocks in cube** do

neighbor \leftarrow *current.swap(block1, block2)*

if *neighbor.value* > *best.value* then
 best \leftarrow *neighbor*

if *best.value* > *current.value* then
 current \leftarrow *best*

if *current* == **highest possible h** then
 return *current*

else
 return *current*

```
}
```

Fungsi sahc di atas akan menerima sebuah initial state dari cube dan akan melakukan inisialisasi properties seperti current. Kemudian fungsi akan melakukan generasi neighbor dengan melakukan swap antar dua block di kubus kemudian akan dilakukan perbandingan antara nilai heuristic antar current dan neighbor, jika neighbor lebih bagus maka current akan menjadi neighbor, looping akan dilakukan sampai semua neighbor telusuri. Kemudian, algoritma akan mencapai local maksimum dimana algoritma akan berhenti dan mengembalikan nilai current. Pada implementasi algoritma ini akan diterapkan prosedur untuk performansi dan keperluan untuk menampilkan beberapa nilai yang lokal .

Procedure:

sahc(Cube* cube): Prosedur ini mengoptimalkan konfigurasi *cube* untuk mencapai nilai heuristik terbaik. Berikut adalah Source Code implementasi algoritma dan akan dijelaskan menjadi beberapa bagian,

Bagian 1: Inisialisasi *linear cube* untuk mengakses elemen secara linear

```
#include "sahc.h"
#include <stdbool.h>

void sahc(Cube *cube) {
    FILE *file = fopen("src/results/sahc_result.csv", "w");

    if (file == NULL) {
        perror("Unable to open file");
        return;
    }

    fprintf(file, "Iteration,Heuristic Value,Time\n");

    int h_current, h_best, h_new;
    uint8_t u1, u2; // Menandakan swap
    uint8_t best_u1, best_u2; // Menyimpan swap terbaik

    int improved = true; // Flag untuk mengecek ada peningkatan atau tidak
    int iterations = 0;

    // Menglinearisasi 3D array untuk memudahkan swap
    uint8_t *linear_cube[TOTAL_VALUES];
    int index = 0;
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            for (int k = 0; k < SIZE; k++) {
                linear_cube[index++] = cube->blocks[i][j][k].value;
            }
        }
    }
}
```

```
    }
}
```

Bagian 2: Hitung heuristik awal dari konfigurasi *cube* saat ini

```
h_current = calculate_heuristics(cube);
```

Bagian 3: Loop utama untuk algoritma SAHC

```
while (improved && h_current < TOTAL_EDGES) {
    improved = false;
    h_best = h_current;
```

Bagian 4: Mencoba semua tetangga swap untuk mencari peningkatan terbaik

```
for (u1 = 0; u1 < TOTAL_VALUES - 1; u1++) {
    for (u2 = u1 + 1; u2 < TOTAL_VALUES; u2++) {
        // Swap
        swap(linear_cube[u1], linear_cube[u2]);

        // Hitung nilai heuristik baru setelah swap
        h_new = calculate_heuristics(cube);

        // Jika heuristik baru lebih baik, simpan sebagai terbaik
        if (h_new > h_best) {
            h_best = h_new;
            best_u1 = u1;
            best_u2 = u2;
            improved = true;
        }

        // Swap ke kondisi awal untuk mencoba tetangga lain
        swap(linear_cube[u1], linear_cube[u2]);
    }
}
```

Bagian 5: Jika ada peningkatan, lakukan swap ke kondisi terbaik dan berhenti jika tidak ada

```
if (improved) {
    // Swap ke kondisi terbaik
    swap(linear_cube[best_u1], linear_cube[best_u2]);
    h_current = h_best;
    iterations++;
    unflatten_cube2(linear_cube);
    printf("Iteration %d: Improved heuristic to %d\n", iterations,
h_current);
    if (h_current == TOTAL_EDGES) {
        printf("Found a magic cube!\n");
    }
}
```

```

        break;
    }
}
else {
    printf("Stopped at iteration %d\n", iterations);
    printf("Local maximum found with h = %d\n", h_current);
}
end_time = clock();
elapsed_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC * 1000;
fprintf(file, "%d,%d,%.4f\n", iterations, h_current, elapsed_time);

}
fclose(file);

}

```

Pada tahap inisialisasi, prosedur dimulai dengan membuat representasi 1D array (*linear_cube*) dari cube untuk memudahkan *swap* nilai antar tetangga. Hal ini menghilangkan kebutuhan untuk *nested loop* dan mempercepat proses *swap*. Pada loop utama, setiap iterasi pada algoritma akan dilakukan pencarian tetangga dengan peningkatan nilai heuristik tertinggi, penyimpanan, dan pembaharuan konfigurasi *cube* jika ditemukan peningkatan. Kemudian, jika tidak ada peningkatan lebih lanjut, algoritma berhenti dan menyimpan hasil saat ini sebagai konfigurasi lokal optimal.

2. Hill-climbing with Sideways Move

Pada **hill-climbing with sideways move** (hcsm), algoritma bekerja mirip dengan **steepest ascent**, tetapi memiliki kemampuan untuk melakukan langkah *sideways*, di mana solusi yang sama baiknya (tidak lebih baik, tetapi juga tidak lebih buruk). Dalam kata lain seluruh skor *objective function* dari *neighbor* sama semua) diterima untuk menghindari plateau. Algoritma ini memungkinkan langkah sideways terbatas untuk menghindari terlalu lama terjebak pada *local flat maximum*.

Kebutuhan komputasi akan hampir sama dengan **steepest ascent hill-climbing** karena tetap mengevaluasi semua tetangga. Penambahan *sideways move* memungkinkan lebih banyak langkah dilakukan tanpa peningkatan, yang meningkatkan jumlah iterasi tetapi memberi algoritma lebih banyak kesempatan untuk keluar dari *plateau*.

Berikut adalah deskripsi fungsi/kelas dari algoritma ini,

function hcsm (*initial state of cube*) returns state with highest value {

current \leftarrow *initial state of cube*
total side ways $\leftarrow 0$

While *current.value* < **highest possible h** do

best \leftarrow *current*

block 1, block 2 $\leftarrow 0$

while *block 1* < **number of blocks in cube** do

block 2 \leftarrow *block 1* + 1

while *block 2* < **number of blocks in cube** do

neighbor \leftarrow *current.swap(block1, block2)*

if *neighbor.value* > *best.value* **then**

best \leftarrow *neighbor*

else if *neighbor.value* == *best.value* and

total side ways < **maximum side ways** **then**

best \leftarrow *neighbor*

total side ways \leftarrow *total sideways* + 1

```

if best.value > current.value then
    current ← best

    if current == highest possible h then
        return current
    else
        return current

}

```

Fungsi hcsm di atas akan menerima sebuah initial state dari cube dan akan melakukan inisialisasi properties seperti *current* dan *total side ways*. Kemudian fungsi akan melakukan generasi neighbor dengan melakukan swap antar dua block di kubus kemudian akan dilakukan perbandingan antara nilai heuristic antar *current* dan *neighbor*, jika *neighbor* lebih bagus maka *current* akan menjadi *neighbor*, looping akan dilakukan sampai semua *neighbor* telusuri. Kemudian, algoritma akan mencapai local maksimum dimana algoritma akan melakukan sideways move dan berhenti ketika global maksimum telah ditemukan atau melebihi maximum sideways move dengan mengembalikan nilai *current*. Pada implementasi algoritma ini akan diterapkan prosedur untuk performansi dan keperluan untuk menampilkan beberapa nilai yang lokal .

Procedure:

hcsm(Cube *cube): Prosedur ini mengoptimalkan konfigurasi *cube* untuk mencapai nilai heuristik terbaik dengan mempertimbangkan langkah *sideways*. Berikut adalah Source Code implementasi algoritma dan akan dijelaskan menjadi beberapa bagian,

Bagian 1: Inisialisasi linear_cube

```

#include "hcsm.h"
#include <stdbool.h>

void hcsm(Cube *cube) {
    FILE *file = fopen("src/results/hcsm_result.csv", "w");

    if (file == NULL) {
        perror("Unable to open file");
        return;
    }

    fprintf(file, "Iteration,Heuristic Value,Total sideways,Time\n");
    clock_t start_time = clock();
}

```

```

clock_t end_time;
double elapsed_time;

int h_current, h_best, h_new;
uint8_t u1, u2; // Menandakan swap
uint8_t best_u1, best_u2; // Menyimpan swap terbaik

int improved = true; // Flag untuk mengecek ada peningkatan atau tidak
int iterations = 0;
// Menglinearisasi 3D array untuk memudahkan swap
uint8_t *linear_cube[TOTAL_VALUES];
int index = 0;
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
        for (int k = 0; k < SIZE; k++) {
            linear_cube[index++] = &cube->blocks[i][j][k].value;
        }
    }
}

```

Bagian 2: Hitung heuristik awal dari konfigurasi *cube* saat ini

```

h_current = calculate_heuristics(cube);

int total_sideways = 0;

```

Bagian 3: Loop utama algoritma HC-SM

```

while (improved && h_current < TOTAL_EDGES) {
    improved = false;
    h_best = h_current;

```

Bagian 4: Mencari pasangan *swap* yang terbaik

```

for (u1 = 0; u1 < TOTAL_VALUES - 1; u1++) {
    for (u2 = u1 + 1; u2 < TOTAL_VALUES; u2++) {
        // Swap
        swap(linear_cube[u1], linear_cube[u2]);

        // Hitung nilai heuristik baru setelah swap
        h_new = calculate_heuristics(cube);

        // Jika heuristik baru lebih baik, perbarui h_best
        if (h_new > h_best) {
            h_best = h_new;
            best_u1 = u1;
            best_u2 = u2;
            improved = true;
        }
    }
}

```

```

        // Jika heuristik baru sama dan total_sideways < limit (i.e.
        100 / bebas), perbarui h_best DAN total_sideways
    else if (h_new == h_best && total_sideways < 2000) {
        total_sideways++;
        h_best = h_new;
        best_u1 = u1;
        best_u2 = u2;
        improved = true;
    }

    // Kembalikan swap jika tidak ada peningkatan
    swap(linear_cube[u1], linear_cube[u2]);
}
}

```

Bagian 5: Jika ada peningkatan, lakukan *swap* ke konfigurasi terbaik

```

iterations++;

if (improved) {
    swap(linear_cube[best_u1], linear_cube[best_u2]); // Lakukan swap
ke terbaik
    h_current = h_best;
    iterations++;
    unflatten_cube2(linear_cube);
    printf("Iteration %d: Improved heuristic to %d\n", iterations,
h_current);

    // Jika heuristik mencapai nilai terbaik (TOTAL_EDGES), solusi
optimal ditemukan
    if (h_current == TOTAL_EDGES) {
        printf("Found a magic cube!\n");
        break;
    }
} else {
    // Jika tidak ada peningkatan, berhenti dan print status
    printf("Stopped at iteration %d\n", iterations);
    printf("Local maximum found with h = %d\n", h_current);
    printf("Total sideways move: %d\n", total_sideways);
}

end_time = clock();
elapsed_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC * 1000;
fprintf(file, "%d,%d,%d,.4f\n", iterations, h_current, total_sideways,
elapsed_time);

}
fclose(file);
}

```

Prosedur dimulai dengan membuat representasi 1D array (*linear_cube*) dari cube untuk memudahkan *swap* nilai antar tetangga. Hal ini menghilangkan kebutuhan untuk *nested loop* dan mempercepat proses *swap*. Pada setiap iterasi, algoritma mengevaluasi semua pasangan elemen dalam cube dan mencoba untuk *swap* elemen-elemen tersebut. Jika nilai heuristik setelah *swap* lebih baik dari nilai heuristik sebelumnya, algoritma memperbarui konfigurasi cube dengan *swap* tersebut. Jika nilai heuristik setelah *swap* tidak lebih baik namun tetap sama dengan nilai terbaik yang ditemukan sebelumnya, dan jumlah langkah *sideways* belum mencapai batas tertentu, algoritma menerima langkah *sideways* tersebut. Namun, jika tidak ada peningkatan nilai heuristik (baik yang lebih baik atau sideways move diterima), maka algoritma berhenti dan menyimpan konfigurasi cube saat itu sebagai *local optimal solution*. Jika solusi optimal ditemukan, algoritma berhenti dan mencetak hasilnya.

3. Random Restart Hill-climbing

Pada **Random Restart Hill-climbing** (rrhc) algoritma melakukan *restart* secara acak jika mencapai *local maximum* (atau *local minimum* tergantung *objective function* yang dipilih) dan memulai ulang dari titik acak lainnya. Dengan cara ini, peluang menemukan solusi global meningkat karena memulai pencarian dari berbagai posisi awal.. Secara teori kubus pada akhirnya akan mendapatkan solusi, hanya saja pada implementasi total random yang diperbolehkan ada 200 karena kemungkinan kubus untuk mencapai global maksimum sangatlah kecil, disini dibatasi untuk sebagai demonstrasi dari algoritma nya saja.

Kebutuhan komputasi akan lebih tinggi dibandingkan dengan **hill-climbing** tanpa pengulangan, sebab melibatkan restart terus menerus menjadi kondisi acak setiap kali menyentuh *local maximum* (dalam kasus ini akan sangat sering). Setiap restart dimulai dengan **hill-climbing** baru, sehingga beban komputasi bertambah seiring bertambahnya restart yang dilakukan. Berikut adalah deskripsi fungsi dari algoritma ini,

function rrhc (*initial state of cube*) return state with highest value {

max restart \leftarrow maximum number of restart

restart \leftarrow 0

current \leftarrow **initial state of cube**

total side ways \leftarrow 0

local maximum \leftarrow false

while *restart* $<$ *max restart* and *current.value* $<$ **highest possible h** do

if *local maximum* then

restart \leftarrow *restart* + 1

if *restart* == *max restart* then

return current

else

current \leftarrow **new state of cube**

total side ways \leftarrow 0

best \leftarrow *current*

block 1, *block 2* \leftarrow 0

while *block 1* $<$ **number of blocks in cube** do

block 2 \leftarrow *block 1* + 1

while *block 2* $<$ **number of blocks in cube** do

```

 $neighbor \leftarrow current.swap( block1, block2)$ 

if  $neighbor.value > best.value$  then
     $best \leftarrow neighbor$ 
else if  $neighbor.value == best.value$  and
     $total\ side\ ways < maximum\ side\ ways$  then
         $best \leftarrow neighbor$ 
         $total\ side\ ways \leftarrow total\ sideways + 1$ 

if  $best.value > current.value$  then
     $current \leftarrow best$ 

if  $current == highest\ possible\ h$  then
    return  $current$ 
else
     $local\ maximum \leftarrow true$ 

}
```

Fungsi rrhc di atas akan menerima sebuah initial state dari cube dan akan melakukan inisialisasi properties seperti jumlah restart maksimal, jumlah sideways, dan seterusnya. Kemudian fungsi akan melakukan generasi neighbor dengan melakukan swap antar dua block di kubus kemudian akan dilakukan perbandingan antara nilai heuristic antar current dan neighbor, jika neighbor lebih bagus maka current akan menjadi neighbor, looping akan dilakukan sampai semua neighbor telusuri. Kemudian, algoritma akan mencapai local maksimum dimana akan dimulai restart dengan cara mengubah initial state kubus menjadi state yang baru dan jumlah restart akan bertambah. Algoritma akan mengulangi tahap sampai jumlah restart mencapai jumlah restart maksimum atau algoritma menemukan global maksimum. Pada implementasi algoritma ini akan diterapkan prosedur untuk performansi dan keperluan untuk menampilkan beberapa nilai yang lokal .

Sesuai penjelasan di atas, implementasi akan dilakukan menggunakan prosedur **rrhc(Cube * cube)** dimana akan menerima sebuah state cube dan mengembalikan cube dengan nilai heuristik terbaik

Berikut adalah Source Code implementasi algoritma dan akan dijelaskan menjadi beberapa bagian,

1. Inisialisasi Properties

```
void rrhc(Cube *cube) {
    FILE *file = fopen("src/results/rrhc_result.csv", "w");

    if (file == NULL) {
        perror("Unable to open file");
        return;
    }

    fprintf(file, "Iteration,Heuristic Value,Restarted,Inner Iteration,Time\n");

    fprintf(file, "Iteration,Heuristic Value,Time\n");

    int h_current, h_best, h_new;
    uint8_t u1, u2;
    uint8_t best_u1, best_u2;

    int improved = true;
    int iterations = 0;
    int inner_iteration = 0;

    int restarted = 0;
    int max_restarted = 200;

    uint8_t *linear_cube[TOTAL_VALUES];
    int index = 0;
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            for (int k = 0; k < SIZE; k++) {
                linear_cube[index++] = &cube->blocks[i][j][k].value;
            }
        }
    }
    clock_t start_time = clock();
    clock_t end_time;
    double elapsed_time;
```

```

    h_current = calculate_heuristics(cube);
    bool reached_local_maximum = false;
    int total_sideways = 0;

    .....
}

```

Pada tahap pertama, algoritma akan melakukan pendefinisian properties seperti variable iterasi maksimum (max_restarted), jumlah restart (restarted), iterasi (iteration), dan seterusnya. Di potongan ini dilakukan juga inisialisasi linear_cube yang akan mengubah cube menjadi satu baris panjang of small cube yang sejumlah 125 atau seperti yang sebelumnya dituliskan, mengubah array yang semulanya 3D menjadi 1D untuk keperluan mempermudah swapping pada tahap generating neighbor, dilakukan juga kalkulasi heuristic value untuk initial cube menggunakan fungsi **calculate_heuristics()**.

2. Looping search (Jika sudah mencapai lokal maksimum)

```

void rrhc(Cube *cube) {
    .....

    while (h_current < TOTAL_EDGES && restarted < max_restarted) {

        if (reached_local_maximum == true) {
            inner_iteration = 0;

            restarted++;
            if (restarted == max_restarted) {
                break;
            }
            shuffle_cube(cube);
            h_current = calculate_heuristics(cube);
            reached_local_maximum = false;
            printf("Restarted %d\n", restarted);
            total_sideways = 0;
        }
        .....
    }
}

```

Kemudian, algoritma akan masuk ke looping search dan akan mengecek apakah sudah mencapai lokal maksimum atau belum, jika sudah, maka algoritma akan melakukan restart dan melakukan perubahan initial state menggunakan prosedur **shuffle_cube()**.

3. Looping search (Generating neighbor dan finding best value from neighbor)

```
void rrhc(Cube *cube) {
    .....
    while (h_current < TOTAL_EDGES && restarted < max_restarted){

        .....

        improved = false;
        h_best = h_current;

        for (u1 = 0; u1 < TOTAL_VALUES - 1; u1++) {

            for (u2 = u1 + 1; u2 < TOTAL_VALUES; u2++) {

                swap(linear_cube[u1], linear_cube[u2]);

                h_new = calculate_heuristics(cube);

                if (h_new > h_best) {
                    h_best = h_new;
                    best_u1 = u1;
                    best_u2 = u2;
                    improved = true;
                }
                else if (h_new == h_best && total_sideways < 2000) {
                    total_sideways++;
                    h_best = h_new;
                    best_u1 = u1;
                    best_u2 = u2;
                    improved = true;
                }

                swap(linear_cube[u1], linear_cube[u2]);
            }
        }

    }
}
```

Pada tahap selanjutnya, algoritma akan swapping dengan menukar posisi ($u1$) dengan posisi 2 ($u2$) dimana $u2$ adalah block selanjutnya dari $u1$ pada *linear_cube* menggunakan prosedur **swap()**. Kemudian akan dilakukan kalkulasi terhadap heuristic value pada semua neighbor dan akan diambil nilai terbaik dan disimpan pada *h_best* demikian juga posisi / neighbor terbaik akan disimpan pada *best_u1* dan *best_u2*. Tidak hanya itu, karena random restart ini juga menggunakan sideways, maka neighbor yang sama dengan current juga disimpan statenya apabila tidak ditemukan neighbor lebih baik dari current.

4. Pemindahan ke neighbor atau restart

```
void rrhc(Cube *cube){
    .....
    while (h_current < TOTAL_EDGES && restarted < max_restarted){

        .....
        iterations++;
        inner_iteration++;

        if (improved) {

            swap(linear_cube[best_u1], linear_cube[best_u2]);
            h_current = h_best;
            iterations++;
            unflatten_cube2(linear_cube);

            printf("Iteration %d: Improved heuristic to %d\n", iterations,
                   h_current);

            if (h_current == TOTAL_EDGES) {
                printf("Found a magic cube!\n");
                break;
            }
        }
        else {
            printf("Stopped at iteration %d\n", iterations);
            printf("Local maximum found with h = %d\n", h_current);
            reached_local_maximum = true;
            printf("Restarting...\n");

        }
    }
}
```

```

        end_time = clock();
        elapsed_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC *
1000;
        fprintf(file, "%d,%d,%d,%d,.4f\n", iterations, h_current, restarted,
inner_iteration, elapsed_time);

    }
printf("Elapsed time %f\n", elapsed_time);
fclose(file);

}

```

Pada tahap ini, algoritma akan mengecek apakah dari neighbor terdapat peningkatan atau tidak terhadap current yang dinilai dari jika nilai *improved* adalah true, jika ada maka algoritma akan mengubah current menjadi neighbor dan algoritma akan mengulang kembali looping. Namun, jika tidak, maka algoritma akan melakukan restart dan menyimpan kondisi *reached_local_maximum* menjadi true dimana algoritma akan menjalankan tahap 2 dan lanjut sampai restart maksimum.

4. Stochastic Hill-climbing

Pada **stochastic hill-climbing** (shc) algoritma akan memilih tetangga secara acak daripada mengevaluasi semua tetangga, dan bergerak ke solusi yang lebih baik jika ditemukan. Jika tetangga yang ditemukan memiliki skor *objective function* yang sama dengan keadaan kubus sekarang, maka pencarian tetangga berikutnya hingga menemukan pertukaran *random* yang memiliki *state* lebih baik. Ini akan lebih efisien dalam hal waktu pencarian karena tidak perlu mengevaluasi semua tetangga.

Kebutuhan komputasi relatif akan cukup rendah dibandingkan **hill-climbing** lainnya karena algoritma ini hanya akan mengevaluasi satu tetangga acak di setiap iterasi, bukan semua tetangga sehingga dapat mengurangi kebutuhan komputasi. Berikut adalah deskripsi dari fungsi pada algoritma ini.

function shc (*initial state of cube*) returns state with highest value {

current \leftarrow ***initial state of cube***

max iterations \leftarrow ***maximum iterations***

repeat *max iterations times* **do**

block 1 \leftarrow *random position in current*

block 2 \leftarrow *random position in current*

while *block 1 == block 2* **do**

Block 2 \leftarrow *random position in a cube*

neighbor \leftarrow *current.swap(block1, block2)*

if *neighbor.value > current.value* **then**

current \leftarrow *neighbor*

if *current == highest possible h* **then**

return *current*

Return *current*

}

Procedure:

shc(Cube *cube): Procedure ini mengoptimalkan konfigurasi *cube* dengan mencari solusi menggunakan algoritma Stochastic Hill Climbing. Berikut adalah Source Code implementasi algoritma dan akan dijelaskan menjadi beberapa bagian,

Bagian 1: Inisialisasi Variabel Utama

```
#include "shc.h"
#include <stdbool.h>

void shc(Cube *cube) {
    FILE *file = fopen("src/results/shc_result.csv", "w");

    if (file == NULL) {
        perror("Unable to open file");
        return;
    }

    fprintf(file, "Iteration,Heuristic Value,Time\n");

    uint8_t u1, u2;
    int iterations = 0;
    int MAX_ITERATIONS = 1000000;
```

Bagian 2: Membuat Representasi 1D dari Cube

```
uint8_t *linear_cube[TOTAL_VALUES];
int index = 0;
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
        for (int k = 0; k < SIZE; k++) {
            linear_cube[index++] = &cube->blocks[i][j][k].value;
        }
    }
}

clock_t start_time = clock();
clock_t end_time;
double elapsed_time;
```

Bagian 3: Menghitung Heuristik Awal

```
h_current = calculate_heuristics(cube);
```

Bagian 4: Loop Utama Stochastic Hill Climbing

```
for (int i = 0; i < MAX_ITERATIONS; i++) {
```

```

// Memilih dua elemen acak dalam linear_cube yang akan di-swap
u1 = rand() % TOTAL_VALUES;
u2 = rand() % TOTAL_VALUES;

while (u1 == u2) {
    u2 = rand() % TOTAL_VALUES;
}

// Swap dua elemen acak, kemudian menghitung nilai heuristik setelah swap
swap(linear_cube[u1], linear_cube[u2]);
h_new = calculate_heuristics(cube);

// Membandingkan nilai heuristik baru dengan nilai heuristik saat ini
if (h_new > h_current) {
    h_current = h_new;
} else {
    swap(linear_cube[u1], linear_cube[u2]);
}

```

Bagian 5: Melakukan *Unflatten Cube* dan Mencetak Status

```

iterations++;
unflatten_cube2(linear_cube);
printf("Iteration %d: Improved heuristic to %d\n", iterations, h_current);

```

Bagian 6: Kondisi Berhenti untuk Solusi Optimal

```

if (h_current == TOTAL_EDGES) {
    printf("Found a magic cube!\n");
    break;
}
end_time = clock();
elapsed_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC * 1000;
fprintf(file, "%d,%d,%f\n", iterations, h_current, elapsed_time);

}
fclose(file);

}

```

Procedure dimulai dengan membuat representasi 1D array (*linear_cube*) dari cube untuk memudahkan *swap* nilai antar tetangga. Hal ini menghilangkan kebutuhan untuk *nested loop* dan mempercepat proses *swap*. Pada setiap iterasi, dua elemen dalam *linear_cube* dipilih secara acak menggunakan indeks *u1* dan *u2*. Kedua elemen ini kemudian di-*swap*, dan algoritma menghitung nilai heuristik baru (*h_new*) setelah *swap* tersebut. Jika heuristik baru (*h_new*) lebih baik dari heuristik saat ini (*h_current*), maka nilai heuristik saat ini akan diperbarui. Namun, jika heuristik baru tidak lebih baik, algoritma akan membatalkan *swap* dan mengembalikan cube ke kondisi

semula. Proses ini terus berulang selama iterasi hingga mencapai batas maksimal iterasi yang ditentukan oleh *max_iterations*, atau hingga solusi optimal ditemukan. Namun, algoritma akan berhenti jika nilai heuristik mencapai nilai terbaik yang mungkin, yaitu *total_edges*, yang menandakan bahwa solusi optimal telah ditemukan. Jika iterasi mencapai batas maksimal dan solusi optimal belum tercapai, algoritma akan berhenti dan mencetak hasil saat itu, menunjukkan bahwa pencarian belum berhasil menemukan solusi terbaik dalam jumlah iterasi yang diberikan.

5. Simulated Annealing

Pada **simulated annealing** (sa), algoritma mengadopsi pendekatan yang memungkinkan penerimaan solusi yang lebih buruk dengan probabilitas tertentu, terutama saat suhu (temperature) tinggi. Seiring waktu (atau dalam implementasi, seiring jumlah iterasi meningkat), suhu akan menurun, sehingga algoritma menjadi lebih selektif dan akhirnya hanya menerima solusi yang lebih baik. Pendekatan ini membantu menghindari terjebak di *local maximum* dengan memberikan kesempatan untuk keluar dari kondisi tersebut di awal atau pertengahan pencarian. Diharapkan dengan membiarkan program “bebas” bergerak, program dapat lebih *flexible* untuk mencari solusi. Pada program ini, untuk rumus **scheduling(t)** atau fungsi untuk menentukan temperatur seiring iterasi atau waktu bergerak merupakan:

$$T_t = T_0 \times \beta^t$$

Dengan t sebagai waktu/jumlah iterasi, T_0 sebagai temperatur sebelumnya, β sebagai *cooling rate* (kami memilih *cooling rate* sebesar 0.999999), dan T_t sebagai temperatur yang akan dihitung. (Referensi rumus di bagian referensi)

Dengan rumus probabilitas pemilihan *state* lebih jelek/sama:

$$p = e^{\Delta h/T}$$

Kebutuhan komputasi relatif sedang hingga sangat tinggi, tergantung pada seberapa lambat algoritma menurunkan temperatur(*cooling schedule*). Evaluasi tetangga dilakukan satu per satu seperti **stochastic hill-climbing**, tetapi juga perlu menghitung probabilitas penerimaan solusi yang lebih buruk, yang mungkin dapat menambah sedikit kebutuhan komputasi.

function sa(*initial state of cube*) **returns state with highest value** {

current \leftarrow *initial state of cube*
max iterations \leftarrow **maximum iterations** (100000000)
initial temperature \leftarrow 1000.0
cooling rate \leftarrow 0.999999

for iterations from 1 to max iterations do

temperature \leftarrow *schedule(iterations, initial temperature, cooling rate)*

if *temperature* < near zero threshold then

return current

block 1 \leftarrow random position in *current*
block 2 \leftarrow random position in *current*

while *block 1* == *block 2* do

block 2 \leftarrow random position in *current*

```

neighbor ← current.swap(block1, block2)

delta h ← neighbor.value - current.value

if delta h > 0 then
    current ← neighbor
else
    probability ← exp(delta_h / temperature)
    if random value < probability then
        current ← neighbor

if current.value == highest possible h then

    return current

}

```

Fungsi sa di atas akan menerima sebuah initial state dari cube dan akan melakukan inisialisasi properties seperti current, max iterations, dan seterusnya. Fungsi kemudian akan menginisialisasi nilai temperature berdasarkan return dari fungsi schedule dan akan dicek apakah sudah mendekati threshold atau batasan, jika iya maka kembalikan current. Kemudian fungsi akan melakukan generasi neighbor dengan melakukan swap antar dua block di kubus kemudian akan dilakukan perbandingan antara nilai heuristic antar current dan neighbor, jika neighbor lebih bagus, dalam konteks ini delta h > 0, maka current akan menjadi neighbor, sebaliknya jika delta h < 0, dengan probabilitas sekian current dapat menjadi neighbor dimana looping akan dilakukan sampai max iterations atau selagi tidak ditemukan global maksimum. Pada implementasi algoritma ini akan diterapkan prosedur untuk performansi dan keperluan menampilkan beberapa nilai yang lokal pada prosedur .

1. Pendefinisian Schedule

```

double schedule1(int iterations, double temperature_0, double cooling_rate) {

    double new_t = temperature_0 * pow(cooling_rate, iterations);

    if (temperature_0 > 0.005f && temperature_0 < 0.05f) {
        return temperature_0 * pow(cooling_rate * 150, iterations);
    }
    else if (temperature_0 > 0.0005f && temperature_0 < 0.005f) {
        return temperature_0 * pow(cooling_rate * 1000, iterations);
    }
}

```

```
    return new_t;
}
```

Sebelum masuk ke algoritma, akan didefinisikan dahulu fungsi untuk schedule dimana fungsi menerima iterasi, temperatur, dan faktor pendingin. Schedule ini menerapkan geometric cooling schedule dimana ketika temperatur jatuh di range pada kondisi if, maka cooling akan diamplify atau dipercepat sebesar kondisi mana yang terpenuhi 150 atau 1000.

2. Inisialisasi Properties

```
void sa(Cube *cube){

    FILE *file = fopen("src/results/sa_result.csv", "w");

    if (file == NULL) {
        perror("Unable to open file");
        return;
    }

    fprintf(file, "Iteration,Heuristic Value,Temperature,Boltzmann
Factor,Stuck,Time\n");

    fprintf(file, "Iteration,Heuristic Value,Time\n");

    int h_current, h_new;
    uint8_t u1, u2;
    int iterations = 1;
    int stuck = 0;
    double boltzman;

    int MAX_ITERATIONS = 100000000;

    double initial_temperature = 1000.0;
    double temperature;
    double cooling_rate = 0.999999;
    uint8_t *linear_cube[TOTAL_VALUES];
    int index = 0;
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
```

```

        for (int k = 0; k < SIZE; k++) {
            linear_cube[index++] = &cube->blocks[i][j][k].value;
        }
    }
    .....
}

```

Pada tahap ini, algoritma masuk ke prosedur dan akan dilakukan inisialisasi properties yang diperlukan pada algoritma ini seperti initial_temperature, cooling rate dan seterusnya. Akan dilakukan juga inisialisasi linear cube sesuai yang telah dijelaskan pada algoritma-algoritma sebelumnya dimana 1D array ini akan membantu dalam swapping pada generasi neighbor.

3. Looping search (Generating neighbor)

```

void sa(Cube *cube){
    .....
    clock_t start_time = clock();
    clock_t end_time;
    double elapsed_time;

    h_current = calculate_heuristics(cube);

    for (iterations = 1; iterations < MAX_ITERATIONS; iterations++) {

        temperature = schedule1(iterations, initial_temperature, cooling_rate);

        if (temperature < 1e-6) {
            printf("Near 0 (harusnya 0 sih cman sigh)\n");
            break;
        }

        u1 = rand() % TOTAL_VALUES;
        u2 = rand() % TOTAL_VALUES;

        while (u1 == u2) {
            u2 = rand() % TOTAL_VALUES;
        }
        swap(linear_cube[u1], linear_cube[u2]);

        h_new = calculate_heuristics(cube);
        .....
    }
}

```

```
}
```

Kemudian, algoritma akan masuk looping sebanyak *max_iterations* dan akan dilakukan pengecekan terlebih dahulu apakah temperatur sudah sangat mendekati nol atau belum, jika belum maka generasi neighbor dengan mengambil suatu dua posisi acak dan menukaranya pada cube, akan dihitung juga nilai heuristik.

4. Looping search (move to best or worst neighbor)

```
void sa(Cube *cube){  
    .....  
    for (iterations = 1; iterations < MAX_ITERATIONS; iterations++){  
        .....  
        int delta_h = h_new - h_current;  
  
        if (delta_h > 0) {  
  
            h_current = h_new;  
        }  
        else {  
  
            double probability = exp(delta_h / temperature);  
  
            if (rand() / (double)RAND_MAX < probability) {  
  
                h_current = h_new;  
            }  
  
            else {  
  
                swap(linear_cube[u1], linear_cube[u2]);  
            }  
  
            printf("Iteration %d: Heuristic value: %d, Temperature: %f\n",  
                   iterations, h_current, temperature);  
  
            if (h_current == TOTAL_EDGES) {  
                printf("Found a magic cube!\n");  
                break;  
            }  
            end_time = clock();  
        }  
    }  
}
```

```
    elapsed_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC * 1000;
    fprintf(file, "%d,%d,%f,%f,%d,.4f\n", iterations, h_current,
temperature, boltzman, stuck,
elapsed_time);

}
fclose(file);
}
```

Pada tahap ini, akan dilakukan pemindahan ke neighbor yang lebih bagus atau buruk sesuai dengan nilai delta h, jika delta h positif maka pindah ke neighbor bagus, jika tidak, maka dengan probabilitas sebuah random value pindah ke neighbor buruk. Algoritma akan berhenti ketika mencapai global maksimum atau iterasi telah mencapai *max_iterations*.

6. Genetic Algorithm

Pada **Genetic algorithm** (ga), algoritma bekerja dengan cara men-generate sejumlah kromosom yang kemudian akan di *crossover* (kawin silang) untuk menghasilkan kromosom baru, orangtua yang memiliki nilai *objective function* (heuristic) yang lebih tinggi akan memiliki kemungkinan kawin yang lebih tinggi dibanding kromosom yang memiliki nilai heuristic yang rendah. Kemudian setelah crossover, setiap kromosom akan memiliki *chance* untuk mengalami mutasi.

Cara crossover bekerja adalah dengan memilih titik tengah secara acak dan menukar kromosom pada belahan setelah titik tengah tersebut, setelah itu akan dilakukan beberapa swap untuk memastikan tidak ada value duplikat pada suatu kromosom. Cara mutasi bekerja adalah dengan memilih beberapa value dari kromosom (1-10) yang kemudian akan ditukar satu sama lain.

Kebutuhan komutasi akan relatif tinggi. Algoritma ini melibatkan berbagai operasi seperti seleksi, *crossover*, dan mutasi, yang memerlukan populasi besar. Setiap generasi kemudian mengulang proses-proses tersebut hingga menemukan generasi terbaik. Belum lagi dengan jumlah populasi yang banyak (pada kasus kami sekitar 1,000 *parent* untuk 2,000 generasi). Ini akan sangat membebani segi komputasi. (Kami sempat me-run program sekitar 10 - 20 menit)

1. Algoritma Utama

```
void ga(Cube *cube) {
    srand(time(NULL));

    generate_chromosome(cube);
    Chromosomes_list *cl = (Chromosomes_list
*)malloc(sizeof(Chromosomes_list));
    Chromosomes_list *new_cl = (Chromosomes_list
*)malloc(sizeof(Chromosomes_list));
    read_chromosome(cl, fopen("src/Parents/gen-1.txt", "r"));
    int total_heuristic = calculate_total_h(cl);
    init_chance(cl, total_heuristic);

    for (int i = 0; i < TOTAL_GENERATION; i++) {
        printf("Current generation: %d\n", i);
        for (int j = 0; j < TOTAL_CHROMOSOME; j += 2) {
            Chromosome *parent1 = select_parent(cl);
            Chromosome *parent2 = select_parent(cl);
```

```

    Chromosome child1 = *parent1;
    Chromosome child2 = *parent2;

    crossover(&child1, &child2);

    mutate(&child1, MUTATION_RATE);
    mutate(&child2, MUTATION_RATE);

    new_cl->chromosomes[j] = child1;
    new_cl->chromosomes[j + 1] = child2;
}

Chromosomes_list *temp = cl;
cl = new_cl;
new_cl = temp;

total_heuristic = calculate_total_h(cl);
init_chance(cl, total_heuristic);
}

sort_chromosome(cl);
write_chromosome(cl, fopen("src/Parents/gen-2.txt", "w"));
read_chromosome(cl, fopen("src/Parents/gen-1.txt", "r"));
sort_chromosome(cl);
write_chromosome(cl, fopen("src/Parents/gen-1.txt", "w"));
free(cl);
free(new_cl);
}

```

2. Prosedur untuk menghasilkan kromosom

```

void generate_chromosome(Cube *cube) {
    FILE *file = fopen("src/Parents/gen-1.txt", "w");

    Block *flat_array = flatten_cube(cube);
    int h;

```

```

for (int i = 0; i < TOTAL_CHROMOSOME; i++) {
    Cube *c = unflatten_cube(flat_array);
    h = calculate_heuristics(c);
    fprintf(file, "%d";
    for (int j = 0; j < TOTAL_VALUES; j++) {
        fprintf(file, "%d:%d ", flat_array[j].value, flat_array[j].pos);
    }
    fprintf(file, "\n");
    shuffle(flat_array);
}

free(flat_array);
fclose(file);
}

```

3.fungsi untuk membaca chromosome

```

Chromosomes_list *read_chromosome(Chromosomes_list *cl, FILE *file) {
    char buf;
    int temp_val, temp_pos, temp_heuristic;
    for (int i = 0; i < TOTAL_CHROMOSOME; i++) {
        fscanf(file, "%d%c", &temp_heuristic, &buf);
        cl->chromosomes[i].h = temp_heuristic;
        for (int j = 0; j < TOTAL_VALUES; j++) {
            fscanf(file, "%d:%d%c", &temp_val, &temp_pos, &buf);
            cl->chromosomes[i].flat_array[j].value = (uint8_t)temp_val;
            cl->chromosomes[i].flat_array[j].pos = (uint8_t)temp_pos;
        }
        if (buf != '\n') {
            fscanf(file, "%c", &buf);
        }
    }

    fclose(file);
    return cl;
}

```

4.Prosedur untuk menyimpan kromosom

```
void write_chromosome(Chromosomes_list *cl, FILE *file) {
    for (int i = 0; i < TOTAL_CHROMOSOME; i++) {
        fprintf(file, "%d:", cl->chromosomes[i].h);
        for (int j = 0; j < TOTAL_VALUES; j++) {
            fprintf(file, "%d:%d ", cl->chromosomes[i].flat_array[j].value,
                    cl->chromosomes[i].flat_array[j].pos);
        }
        fprintf(file, "\n");
    }
    fclose(file);
}
```

5.Prosedur untuk mutasi

```
void mutate(Chromosome *chromosome, double mutation_rate) {
    // Determine the number of swaps to perform (between 1 and 10)
    int num_swaps = 1 + rand() % 10;

    for (int swap_count = 0; swap_count < num_swaps; swap_count++) {
        for (int i = 0; i < TOTAL_VALUES; i++) {
            if ((double)rand() / RAND_MAX < mutation_rate) {
                int pos2 = rand() % TOTAL_VALUES;
                Block temp = chromosome->flat_array[i];
                chromosome->flat_array[i] = chromosome->flat_array[pos2];
                chromosome->flat_array[pos2] = temp;
            }
        }
    }
}
```

6.Fungsi untuk mengkalkulasi heuristik kromosom

```
int calculate_total_h(Chromosomes_list *cl) {
    int h = 0;
```

```
for (int i = 0; i < TOTAL_CHROMOSOME; i++) {  
    h += cl->chromosomes[i].h;  
}  
return h;  
}
```

7.Prosedur untuk menghitung peluang

```
void init_chance(Chromosomes_list *cl, int h) {  
    for (int i = 0; i < TOTAL_CHROMOSOME; i++) {  
        cl->chromosomes[i].chance = ((double)cl->chromosomes[i].h + EPSILON) /  
(double)h;  
    }  
}
```

8.Fungsi untuk membandingkan kromosom

```
int compare_chromosomes(const void *a, const void *b) {  
    Chromosome *chromosome_a = (Chromosome *)a;  
    Chromosome *chromosome_b = (Chromosome *)b;  
    return chromosome_b->h - chromosome_a->h;  
}
```

9.Prosedur untuk mensort kromosom

```
void sort_chromosome(Chromosomes_list *cl) {  
    qsort(cl->chromosomes, TOTAL_CHROMOSOME, sizeof(Chromosome),  
    compare_chromosomes);  
}
```

10.Prosedur untuk crossover

```
void crossover(Chromosome *state1, Chromosome *state2) {  
    int crossover_point = rand() % TOTAL_VALUES;
```

```

for (int i = crossover_point; i < TOTAL_VALUES; i++) {
    Block temp = state1->flat_array[i];
    state1->flat_array[i] = state2->flat_array[i];
    state2->flat_array[i] = temp;
}

int value_count[TOTAL_VALUES] = {0};
for (int i = 0; i < TOTAL_VALUES; i++) {
    value_count[state1->flat_array[i].value]++;
}
for (int i = 0; i < TOTAL_VALUES; i++) {
    if (value_count[state1->flat_array[i].value] > 1) {
        for (int j = 0; j < TOTAL_VALUES; j++) {
            if (value_count[j] == 0) {
                value_count[state1->flat_array[i].value]--;
                state1->flat_array[i].value = j;
                value_count[j]++;
                break;
            }
        }
    }
}
for (int i = 0; i < TOTAL_VALUES; i++) {
    value_count[i] = 0;
}
for (int i = 0; i < TOTAL_VALUES; i++) {
    value_count[state2->flat_array[i].value]++;
}
for (int i = 0; i < TOTAL_VALUES; i++) {
    if (value_count[state2->flat_array[i].value] > 1) {
        for (int j = 0; j < TOTAL_VALUES; j++) {
            if (value_count[j] == 0) {
                value_count[state2->flat_array[i].value]--;
                state2->flat_array[i].value = j;
                value_count[j]++;
                break;
            }
        }
    }
}

```

```
        }
    }
}
}
```

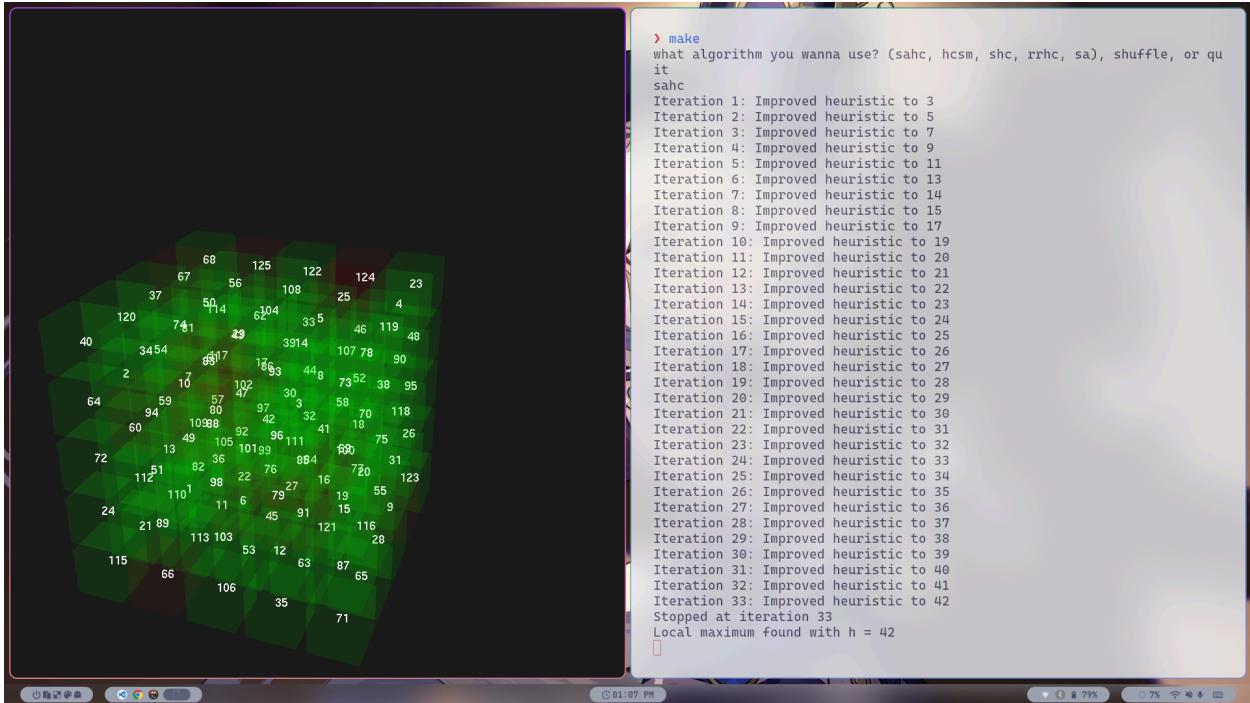
11.Fungsi memilih parent

```
Chromosome *select_parent(Chromosomes_list *cl) {
    double r = (double)rand() / RAND_MAX;
    double cumulative_probability = 0.0;

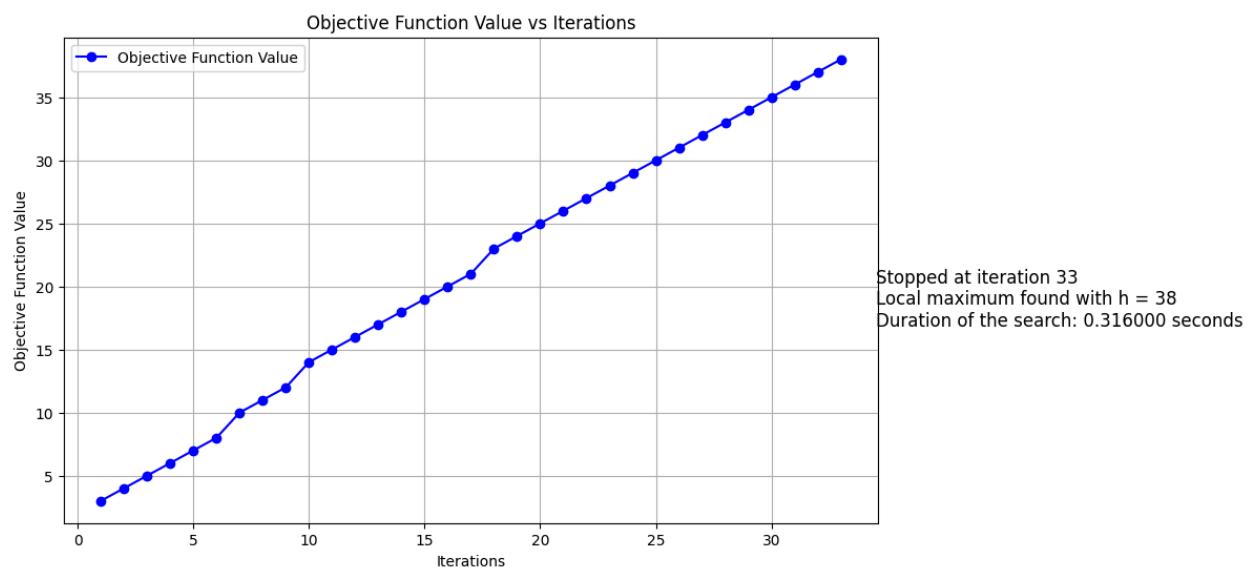
    for (int i = 0; i < TOTAL_CHROMOSOME; i++) {
        cumulative_probability += cl->chromosomes[i].chance;
        if (r <= cumulative_probability) {
            return &cl->chromosomes[i];
        }
    }
    return &cl->chromosomes[TOTAL_CHROMOSOME - 1];
}
```

2.3 Hasil Eksperimen dan Analisis

1. Steepest Ascent Hill-Climbing



1st Experiment:



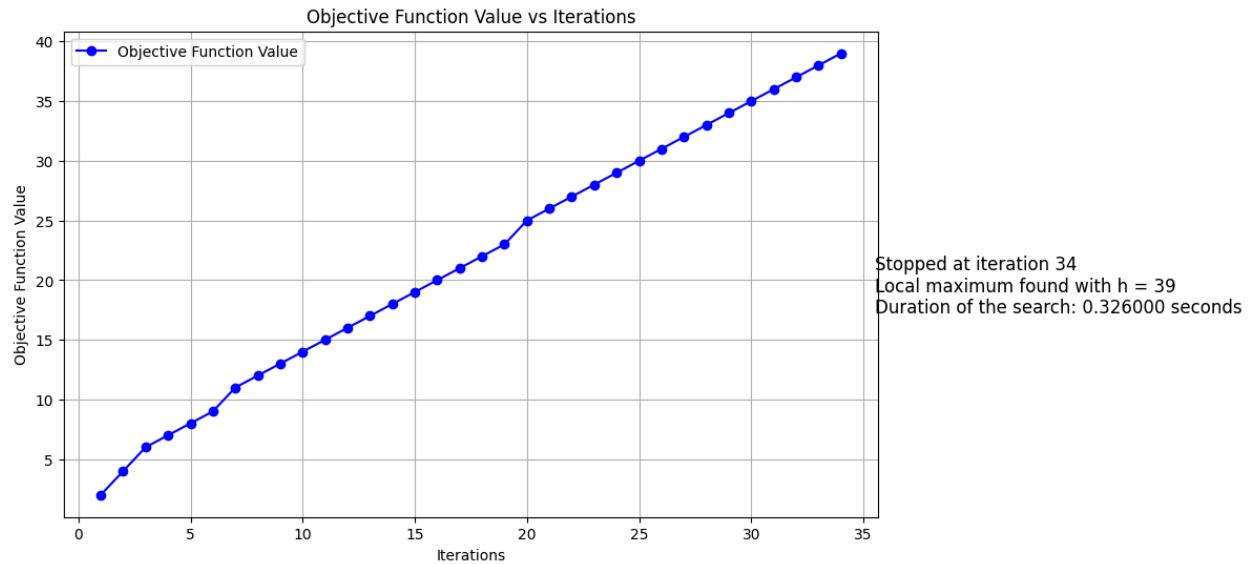
Inisial state:

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

Akhir state:

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

2nd Experiment:



Inisial state:

Layer 1:

59 5 67 69 64
92 15 45 112 30
18 54 49 17 47
98 9 99 73 26
77 72 6 121 60

Layer 2:

83 108 80 14 61
21 57 120 111 29
97 3 68 74 75
56 34 85 110 35
94 84 16 2 88

Layer 3:

93 76 55 4 105
33 66 46 10 51
19 31 20 62 71
1 81 50 58 114
102 32 122 86 96

Layer 4:

109 42 38 89 22
91 117 43 90 36
63 113 39 53 25
119 116 95 118 70
125 79 65 11 41

Layer 5:

107 28 7 12 48
115 78 27 40 52
106 104 82 101 13
103 8 24 23 123
124 87 37 44 100

Akhir state:

Layer 1:
15 121 46 69 64
107 9 57 112 30
18 54 3 25 2
98 59 99 6 65
77 72 71 103 60

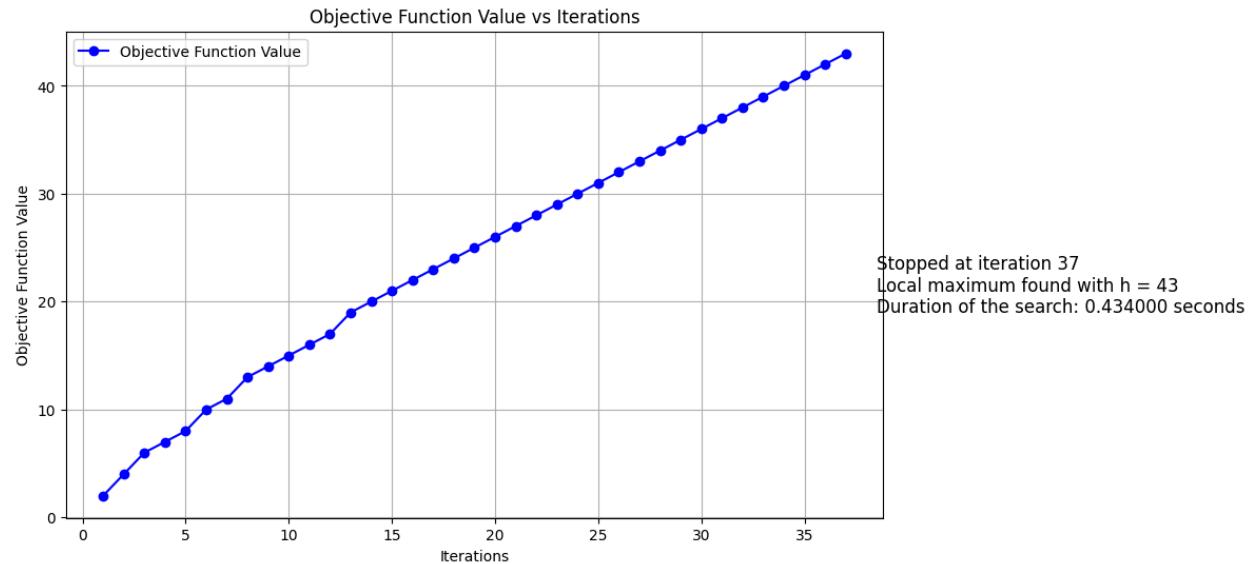
Layer 2:
83 108 49 14 61
21 34 120 111 29
92 80 45 74 93
56 67 47 110 35
91 26 16 85 97

Layer 3:
75 76 55 4 105
125 66 68 10 51
36 31 20 62 73
1 102 50 58 104
78 32 122 86 96

Layer 4:
109 42 38 89 37
94 117 43 90 19
63 113 39 53 17
119 116 95 23 70
33 100 84 11 88

Layer 5:
8 28 7 12 48
115 81 27 40 52
106 114 82 101 13
41 5 24 118 123
124 87 22 44 79

3rd Experiment:



Inisial state:

Layer 1:
 61 77 118 103 27
 94 73 82 60 72
 59 76 42 95 66
 113 5 8 15 20
 22 81 99 9 88

Layer 2:
 75 43 119 45 93
 2 16 62 1 96
 120 11 30 24 86
 91 37 121 105 85
 14 39 67 110 74

Layer 3:
 48 104 79 23 115
 51 41 102 26 29
 108 70 112 90 97
 13 123 64 106 4
 6 122 54 50 36

Layer 4:
 21 117 46 31 92
 18 28 80 116 69
 33 25 3 55 101
 100 111 52 89 78
 19 109 32 56 10

Layer 5:
 98 87 47 35 84
 40 68 58 34 71
 12 7 38 44 57
 49 83 125 124 17
 53 107 65 63 114

Akhir state:

Layer 1:
37 111 24 82 61
10 73 118 54 60
74 124 42 95 86
113 5 30 75 20
81 103 101 9 88

Layer 2:
15 43 119 45 93
2 16 62 85 96
8 11 120 110 66
91 27 44 105 1
12 39 67 115 59

Layer 3:
98 104 79 122 48
51 107 102 26 29
108 70 112 90 68
18 123 64 106 4
6 23 50 63 36

Layer 4:
21 117 46 31 100
22 28 80 116 69
33 25 3 55 89
14 77 52 94 78
19 109 32 56 99

Layer 5:
13 87 47 35 84
40 97 58 34 71
92 7 38 121 57
49 83 125 53 17
76 41 65 72 114

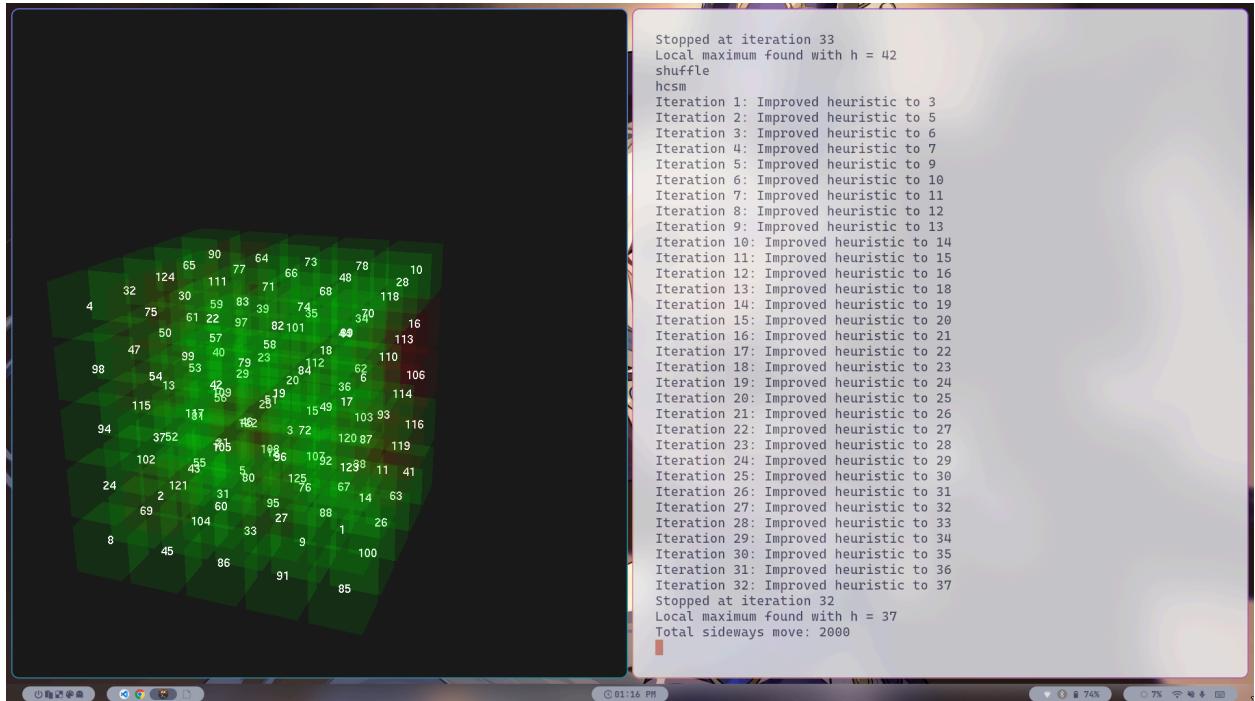
Ketiga eksperimen menunjukkan bahwa algoritma *Steepest Ascent Hill Climbing* tidak mendekati *global optima* (109). Hasil ini menunjukkan bahwa algoritma ini terjebak pada *local maxima* yang jauh dari *global optima*. Hal ini terjadi karena algoritma ini hanya bergerak ke arah peningkatan terbesar pada setiap iterasi tanpa mekanisme untuk keluar dari *local maxima*.

Hasil eksperimen menunjukkan variasi dalam nilai heuristik akhir yang ditemukan:

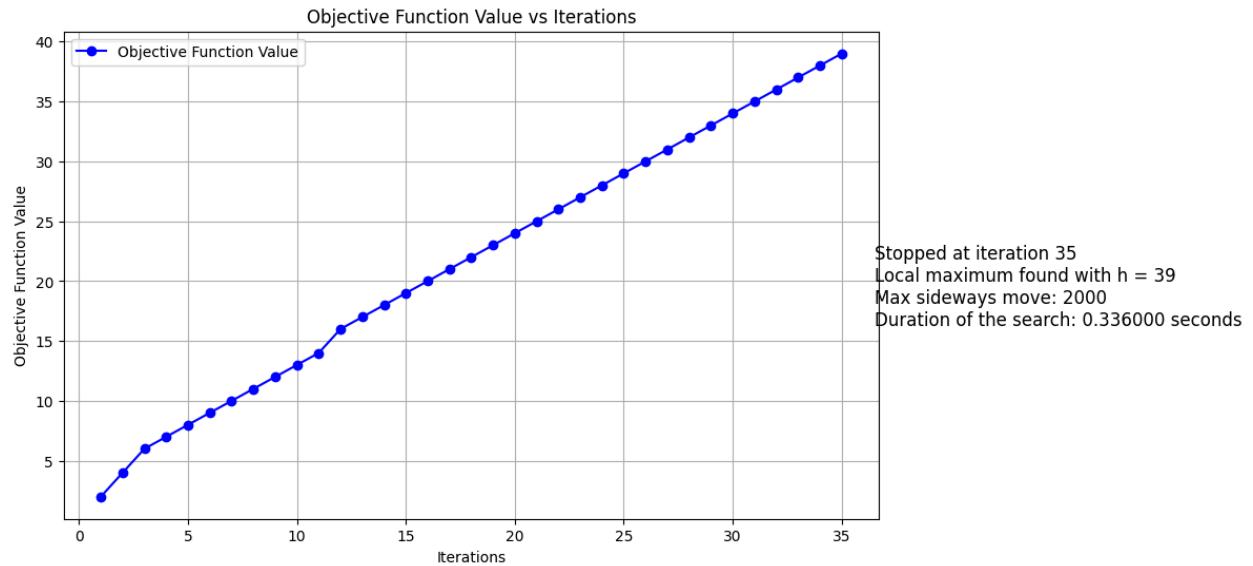
- Eksperimen pertama berhenti pada iterasi ke-33 dengan nilai heuristik 38.
- Eksperimen kedua berhenti pada iterasi ke-34 dengan nilai heuristik 39.
- Eksperimen ketiga berhenti pada iterasi ke-37 dengan nilai heuristik 43.

Ini menunjukkan bahwa hasil dari *Steepest Ascent Hill Climbing* bisa bervariasi tergantung pada kondisi awal dan jalur pencarian yang diambil. Konsistensi dapat dijaga dengan menjalankan algoritma beberapa kali dan memilih hasil sesuai kebutuhan dan tujuan.

2. Hill-climbing with Sideways Move



1st Experiment:



Inisial state si cube:

Layer 1:
65 3 63 18 80
88 13 106 5 38
19 122 119 10 85
84 16 49 41 59

Layer 2:
99 51 103 92 58
12 67 112 104 35
109 43 116 22 55
125 61 69 101 46

Layer 3:
70 95 124 118 56
73 91 32 121 37
62 93 6 76 50
117 120 81 113 78

75 53 44 82 57 115 11 4 54 31 105 30 27 110 90

Layer 4:
74 9 100 66 1
15 8 34 52 87
107 7 28 64 36
71 45 98 25 72
23 94 39 89 77

Layer 5:
33 114 29 123 60
97 108 86 42 40
24 68 48 14 47
17 2 79 111 21
102 20 83 96 26

Akhir state si cube:

Layer 1:
65 46 106 18 80
88 60 5 124 38
19 82 119 10 85
54 112 49 41 59
75 15 36 122 53

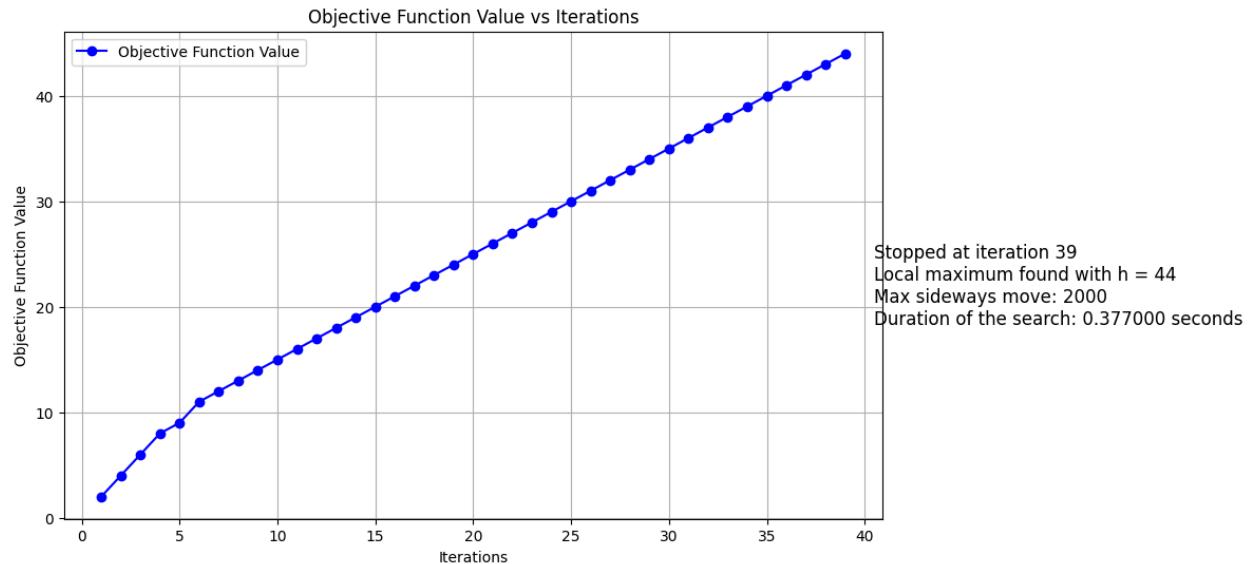
Layer 2:
76 51 110 92 57
12 67 16 104 99
109 43 116 22 55
3 61 69 101 73
115 11 4 84 31

Layer 3:
70 95 63 118 56
125 91 68 28 113
62 93 120 35 50
117 6 81 37 17
105 30 107 103 44

Layer 4:
71 9 100 66 1
58 8 34 52 87
27 7 13 64 78
74 45 24 25 72
23 123 39 108 77

Layer 5:
33 114 26 21 121
125 91 68 28 113
62 93 120 35 50
117 6 81 37 17
105 30 107 103 44

2nd Experiment:



Inisial state si cube:

Layer 1:

71 6 57 15 13
35 38 59 14 125
106 84 95 34 40
80 3 98 122 39
90 33 119 19 11

Layer 2:

73 70 93 50 75
76 24 47 112 118
101 8 72 28 48
31 46 1 12 9
77 16 55 18 43

Layer 3:

53 2 62 117 56
97 113 61 66 110
86 69 4 89 120
37 23 88 7 32
41 36 29 5 100

Layer 4:

111 42 124 121 60
123 91 105 51 79
22 26 103 64 67
78 30 115 87 54
49 83 108 92 68

Layer 5:

25 27 82 109 102
85 20 104 96 81
94 21 63 52 114
65 10 17 44 99
116 74 107 45 58

Akhir state si cube:

Layer 1:
 71 98 57 15 13
 35 38 105 125 85
 39 95 28 34 37
 80 59 6 122 48
 90 33 119 19 11

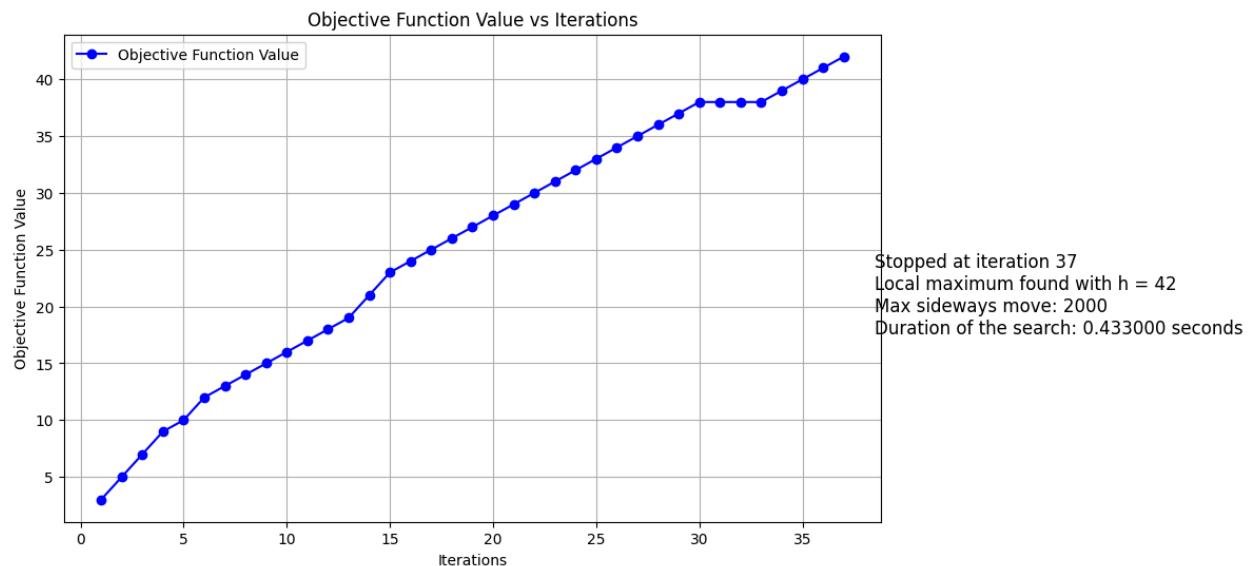
Layer 2:
 30 70 93 50 72
 76 24 47 112 118
 101 86 4 83 106
 31 109 1 14 9
 77 16 55 56 111

Layer 3:
 53 2 62 117 110
 97 113 61 66 60
 46 69 75 89 5
 124 23 88 7 73
 41 108 29 36 67

Layer 4:
 43 42 40 54 18
 123 120 3 114 8
 22 26 103 64 100
 78 32 115 87 121
 49 84 91 92 68

Layer 5:
 25 27 82 79 102
 52 20 99 63 81
 94 104 96 45 51
 65 10 17 116 107
 44 74 21 12 58

3rd Experiment:



Inisial state si cube:

Layer 1:
 73 87 86 47 10
 61 45 18 13 29
 40 103 1 48 41
 102 51 35 26 57
 116 33 36 39 66

Layer 2:
 59 5 78 121 76
 15 114 46 83 93
 89 56 74 71 4
 58 98 72 12 115
 101 108 79 11 107

Layer 3:
 2 20 62 23 25
 30 7 125 110 67
 88 68 38 81 24
 9 16 65 70 109
 21 44 99 113 52

Layer 4:	Layer 5:
96 60 97 105 69	49 77 27 53 84
42 122 85 111 112	117 32 43 124 90
28 17 120 55 75	91 50 104 22 64
92 8 94 34 19	6 37 3 31 123
80 106 54 100 63	119 118 14 82 95

Akhir state si cube:

Layer 1:	Layer 2:	Layer 3:
73 87 98 47 10	59 5 78 121 76	2 20 75 122 25
117 79 123 115 29	15 114 46 83 93	80 96 18 110 67
40 103 23 88 41	89 77 74 71 4	64 68 95 81 62
102 51 35 26 101	58 11 72 12 13	9 16 111 70 109
116 33 36 39 66	6 108 45 86 24	21 30 99 113 52

Layer 4:	Layer 5:
97 60 7 105 120	49 37 27 118 84
42 124 85 65 112	61 90 43 107 14
31 17 69 55 1	91 50 104 22 48
92 8 94 34 19	54 100 3 28 125
53 106 57 56 63	119 38 32 82 44

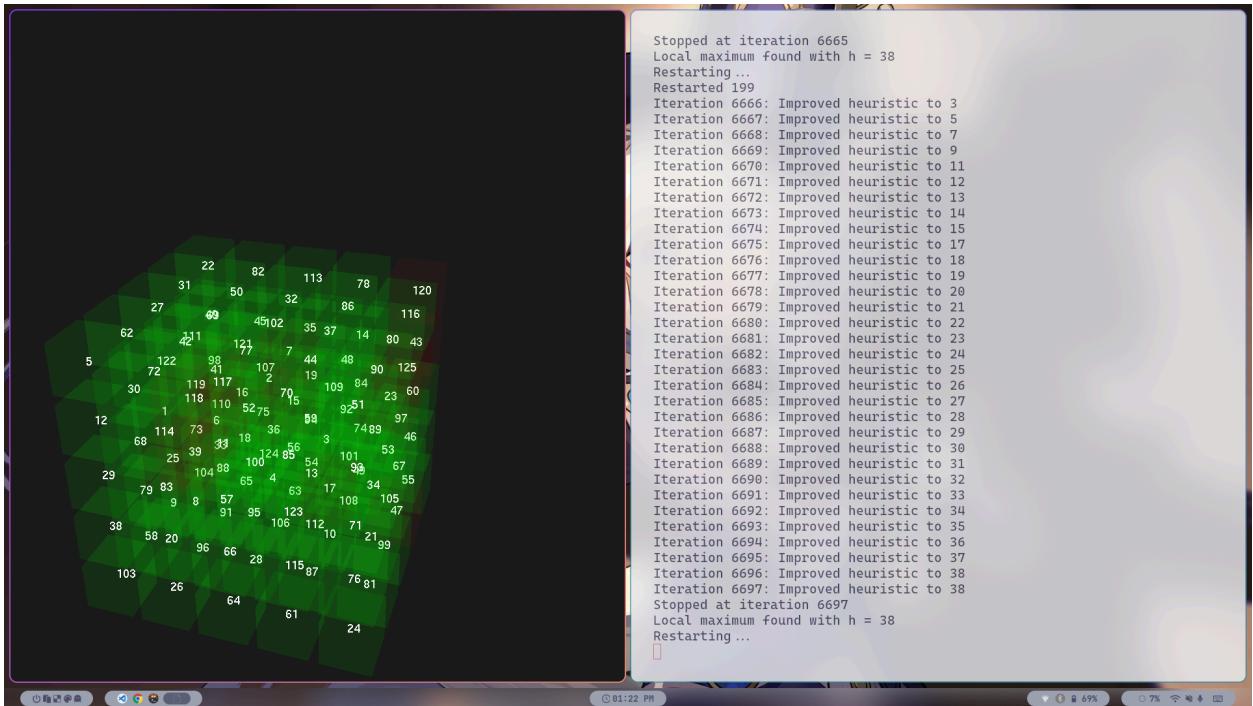
Ketiga eksperimen menunjukkan bahwa algoritma *Hill Climbing With Sideways Move* tidak mendekati *global optima* (109). Hasil ini menunjukkan bahwa algoritma ini terjebak pada *local maxima* yang jauh dari *global optima*. Meskipun *sideways move* membantu menghindari beberapa jebakan *local maxima*, algoritma ini masih terbatas dalam eksplorasi ruang solusi yang lebih luas.

Hasil eksperimen menunjukkan variasi dalam nilai heuristik akhir yang ditemukan:

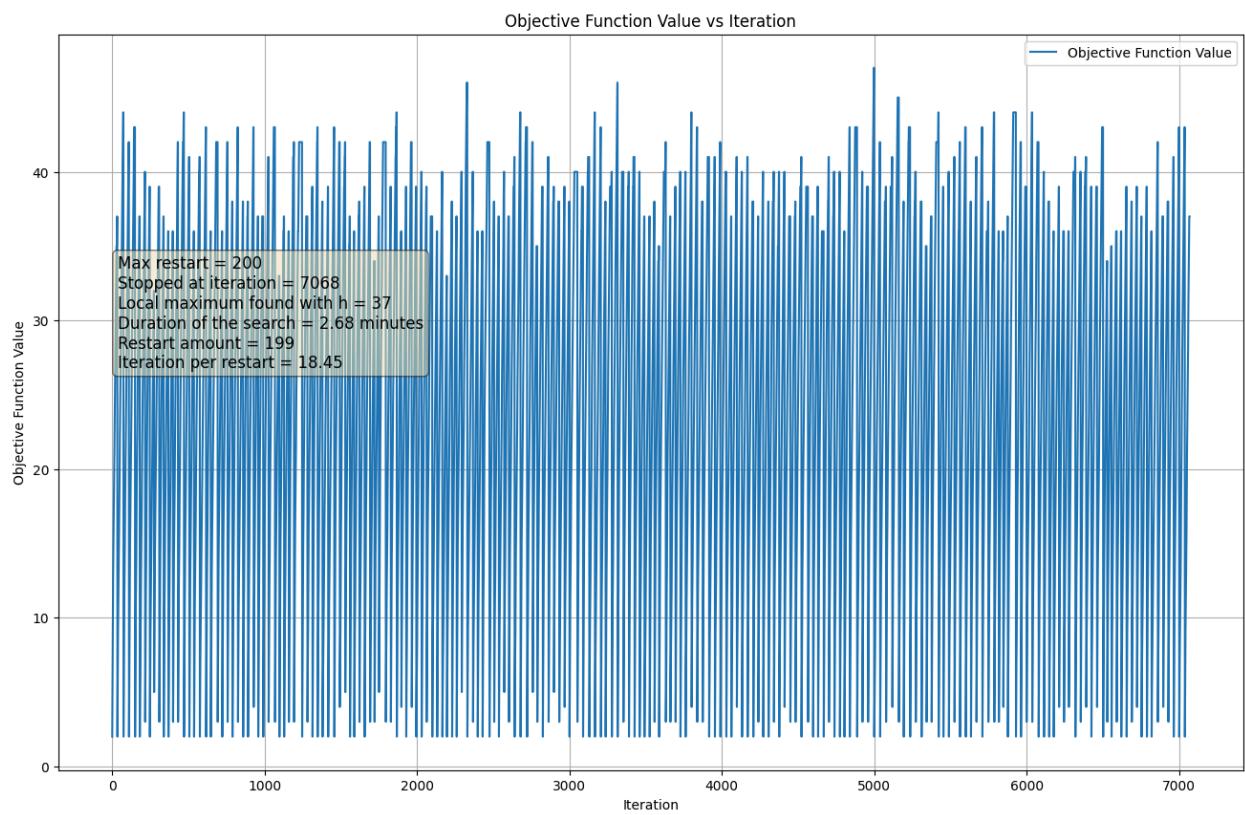
- Eksperimen pertama berhenti pada iterasi ke-35 dengan nilai heuristik 39.
- Eksperimen kedua berhenti pada iterasi ke-39 dengan nilai heuristik 44.
- Eksperimen ketiga berhenti pada iterasi ke-37 dengan nilai heuristik 42.

Ini menunjukkan bahwa hasil dari *Hill Climbing With Sideways Move* bisa bervariasi tergantung pada kondisi awal dan panjangnya shoulder pada neighbor. Konsistensi pada algoritma ini dapat dijaga dengan menjalankan algoritma beberapa kali dan memilih hasil sesuai kebutuhan atau tujuan.

3. Random Restart Hill-climbing



1st Experiment:



Initial State:

Layer 1:	Layer 2:	Layer 3:
13 53 70 51 27	14 54 98 79 90	25 113 4 116 8
29 62 83 9 33	66 86 57 43 35	38 102 88 109 30
72 111 119 108 36	5 117 17 100 104	123 44 84 120 124
48 6 19 89 20	58 56 82 112 121	94 31 15 107 50
85 103 28 32 2	92 93 106 73 49	22 80 114 55 64

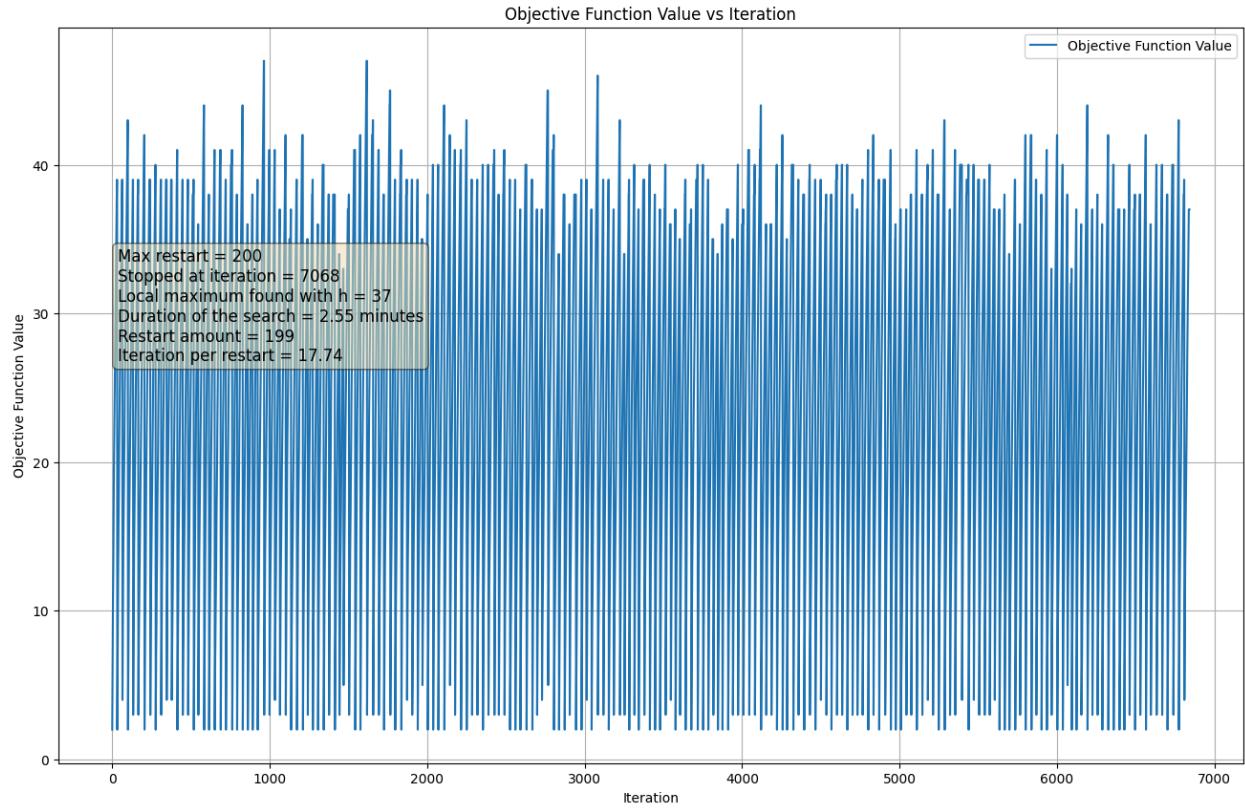
Layer 4:	Layer 5:
99 101 75 76 95	87 34 12 7 3
77 37 74 96 60	18 110 23 41 71
118 24 63 59 42	46 1 67 39 45
105 69 81 10 52	122 40 26 65 91
11 47 125 61 68	115 78 21 16 97

Akhir state si cube:

Layer 1:	Layer 2:	Layer 3:
103 8 96 95 109	119 24 59 42 71	49 66 80 35 85
77 124 123 101 116	64 26 13 114 98	30 48 65 25 20
23 89 104 60 39	87 14 19 70 32	94 50 99 86 1
102 52 91 47 44	63 121 28 74 100	115 75 82 118 97
106 120 55 12 22	117 18 105 34 41	31 76 45 51 112

Layer 4:	Layer 5:
29 69 110 113 83	15 72 92 46 90
37 81 6 93 27	107 36 16 10 54
58 84 88 9 43	53 78 5 57 122
33 11 3 62 7	2 56 111 4 67
40 68 108 38 61	21 73 125 17 79

2nd Experiment:



Initial State:

Layer 1:
35 31 122 21 26
114 44 72 34 74
97 33 57 113 65
18 14 3 22 63
79 99 8 60 27

Layer 2:
125 28 32 85 103
64 58 91 36 68
4 48 98 104 16
87 88 11 50 40
86 110 118 101 29

Layer 3:
73 15 55 76 9
25 116 93 75 78
1 46 120 117 84
45 53 54 10 83
41 111 112 67 49

Layer 4:
90 119 69 51 95
70 19 23 100 121
13 24 96 2 94
106 124 30 80 56
109 7 62 115 5

Layer 5:
66 17 82 102 39
89 6 92 81 20
123 59 105 61 77
71 38 52 47 12
107 42 43 108 37

Akhir state si cube:

Layer 1:
31 92 10 55 122
46 9 95 76 23
40 114 83 1 77
5 68 66 80 90
7 32 41 103 53

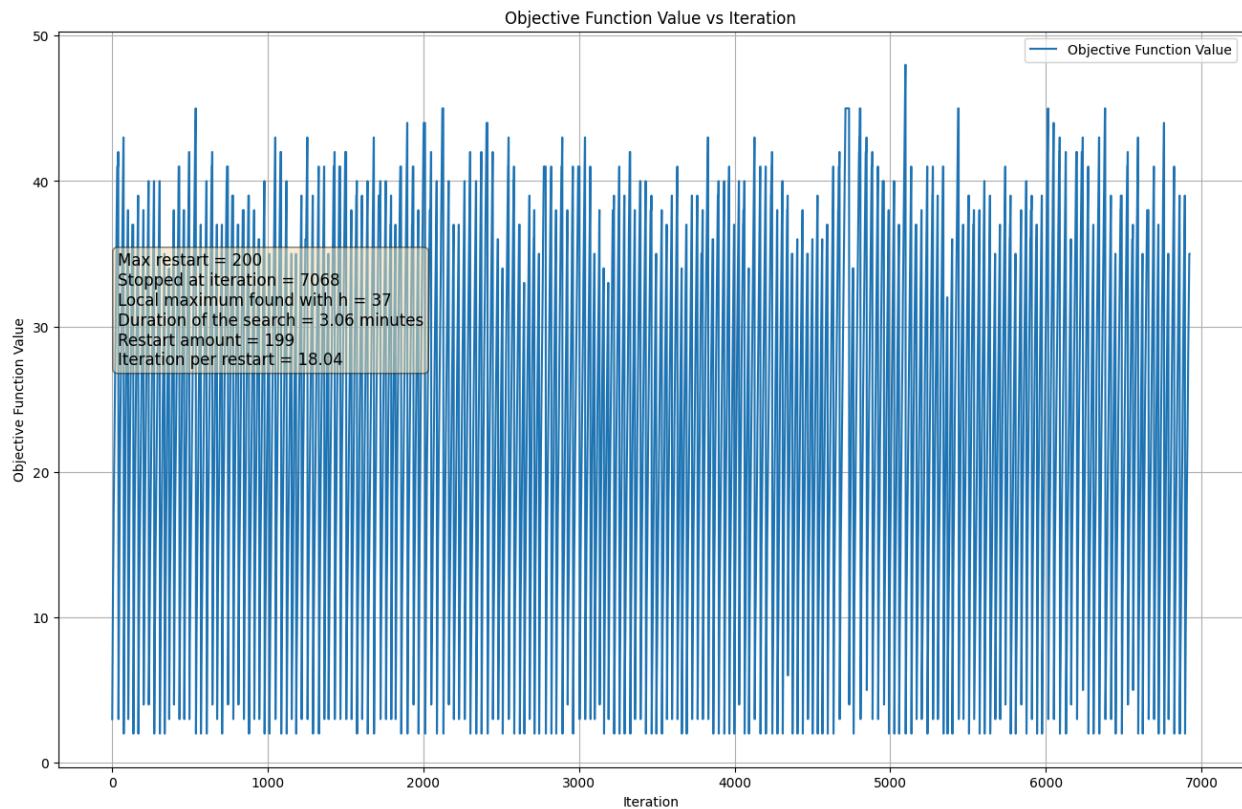
Layer 2:
2 51 89 67 106
8 63 3 54 123
113 117 110 29 108
84 20 97 14 82
101 64 16 19 115

Layer 3:
37 6 56 111 42
81 88 78 116 107
57 69 4 62 85
105 28 65 49 87
35 38 112 61 44

Layer 4:
52 72 109 39 43
120 100 48 119 21
26 70 86 121 12
13 124 34 99 45
98 93 36 58 30

Layer 5:
47 94 125 17 74
60 18 91 96 50
79 104 59 102 33
22 75 25 73 71
11 24 15 27 118

3rd Experiment:



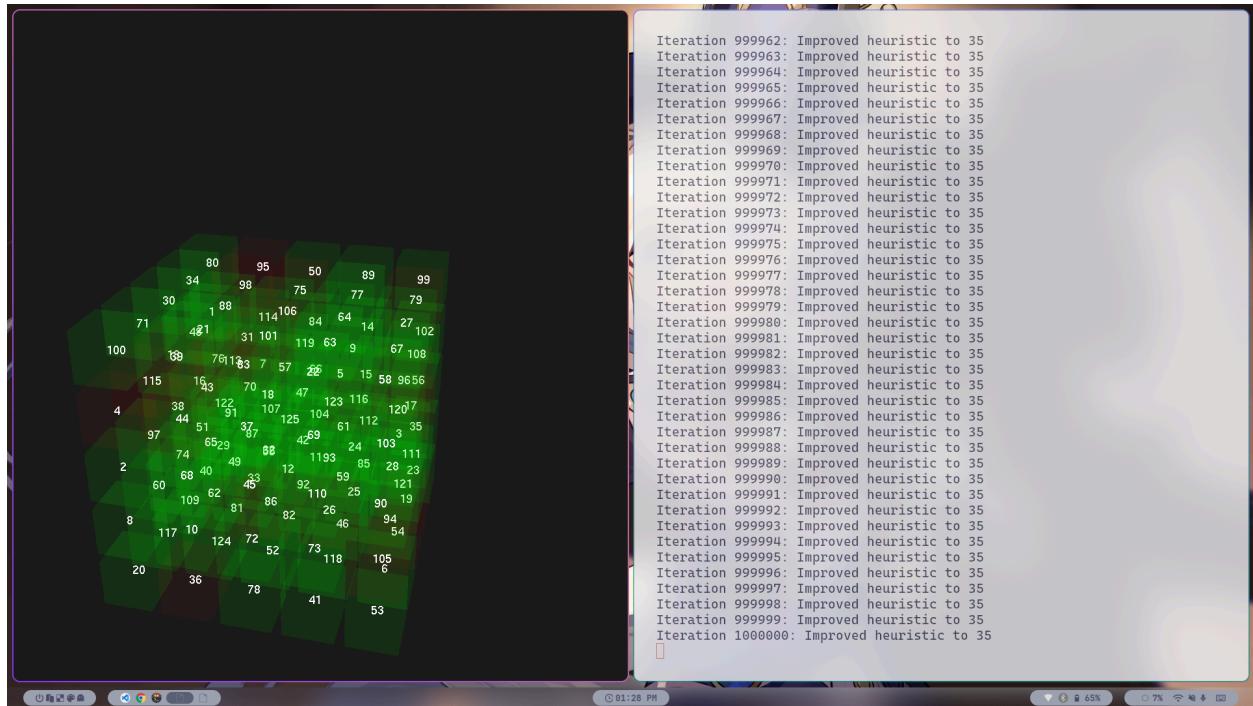
Inisial state si cube:

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

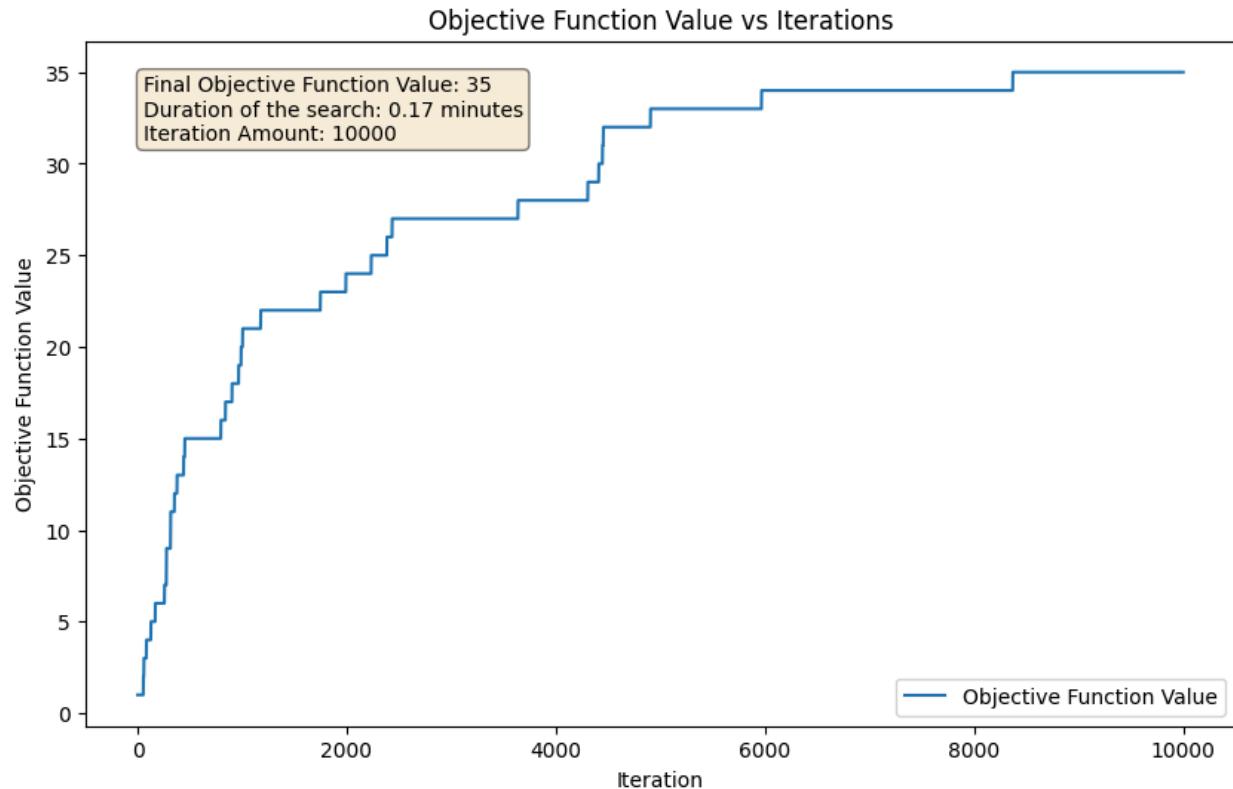
Akhir state si cube:

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

4. Stochastic Hill-climbing



1st Experiment:



Inisial state si cube:

Layer 1:
43 20 53 44 92
81 63 3 7 114
75 27 2 113 4
12 72 17 76 123
110 100 68 95 35

Layer 2:
85 103 79 47 86
40 62 52 120 117
25 28 10 87 102
96 116 39 24 89
29 48 50 106 82

Layer 3:
118 23 14 16 88
66 32 36 34 1
49 58 61 38 60
19 56 18 13 93
55 26 101 59 37

Layer 4:
31 94 97 84 22
51 64 41 57 80
122 99 45 5 11
125 73 70 8 65
124 33 67 112 71

Layer 5:
119 121 111 54 21
98 83 107 6 46
77 9 91 105 42
104 78 30 108 115
90 74 69 109 15

Akhir state si cube:

Layer 1:
43 20 116 44 92
81 63 89 7 114
69 27 1 113 65
12 105 17 76 123
110 100 68 75 53

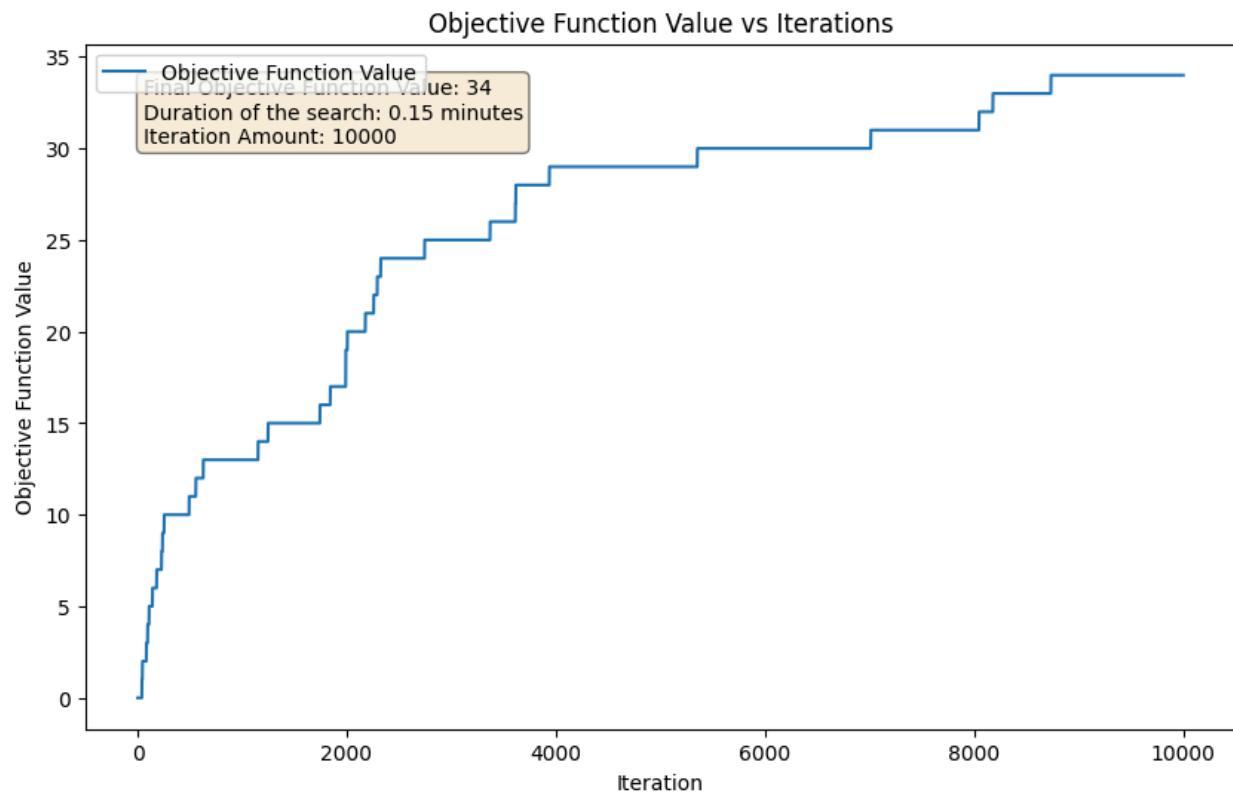
Layer 2:
85 103 57 107 86
104 101 32 120 74
25 28 23 13 102
72 35 34 56 3
29 48 82 106 50

Layer 3:
71 10 14 16 88
66 52 60 87 2
125 58 61 38 95
19 24 18 39 93
55 26 62 59 37

Layer 4:
124 94 97 5 22
51 64 41 79 80
15 99 40 115 11
49 73 70 4 119
31 33 67 112 83

Layer 5:
8 121 111 54 21
98 118 47 6 46
77 9 91 96 42
45 78 30 117 84
90 108 36 109 122

2nd Experiment:



Inisial state si cube:

Layer 1:
53 121 115 69 21
104 22 27 70 60
12 34 24 101 80
125 102 68 66 47
91 32 84 2 63

Layer 2:
112 3 99 8 83
39 62 67 113 1
48 107 41 73 61
105 46 116 59 90
57 4 38 100 111

Layer 3:
103 52 108 87 7
33 98 86 81 16
51 28 65 42 120
110 44 45 26 96
55 89 92 37 74

Layer 4:
31 97 29 50 79
114 122 6 19 75
85 93 13 43 88
82 49 35 20 72
95 25 78 30 14

Layer 5:
10 117 36 18 109
5 17 106 56 76
40 124 9 58 54
11 123 64 71 23
94 77 15 119 118

Akhir state si cube:

Layer 1:
52 121 56 69 17
104 2 108 70 31
91 34 99 20 80
88 53 12 66 96
76 32 42 102 63

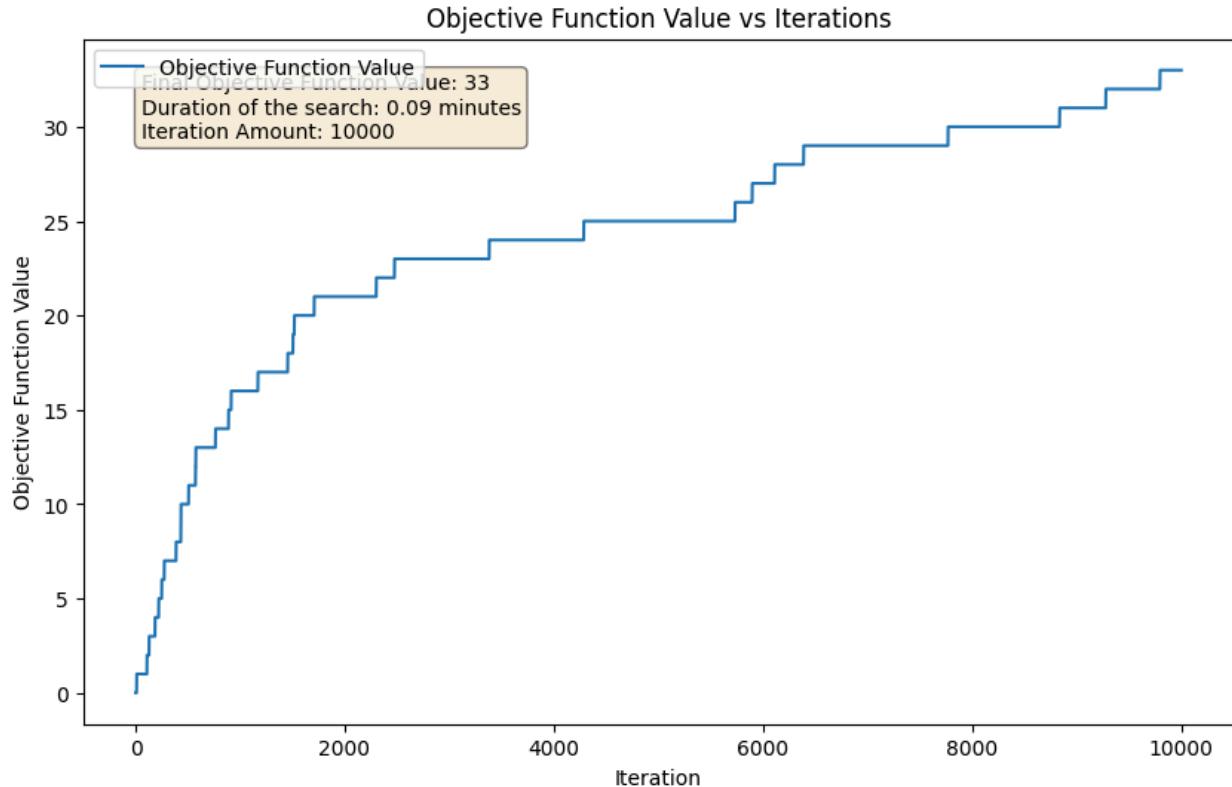
Layer 2:
118 3 112 8 83
59 62 67 113 1
48 107 117 13 30
24 46 116 39 90
57 9 38 100 111

Layer 3:
103 93 27 87 7
33 47 86 81 16
51 28 65 84 120
110 44 45 26 98
18 75 92 37 74

Layer 4:
29 97 60 50 79
114 122 6 19 54
85 22 73 43 125
82 49 35 77 72
95 25 78 61 14

Layer 5:
10 41 36 101 109
5 21 106 115 68
40 124 4 58 89
11 123 64 94 23
71 55 15 119 105

3rd Experiment:



Inisial state si cube:

Layer 1:
56 68 30 103 123
66 89 98 47 100
20 65 45 4 26
124 58 50 121 35
67 12 16 29 43

Layer 2:
64 111 57 97 8
19 51 31 84 95
108 10 42 54 40
87 34 74 79 9
18 102 96 24 14

Layer 3:
77 33 118 7 36
110 52 11 55 27
15 112 44 32 13
76 71 17 107 6
94 120 116 122 28

Layer 4:
106 101 41 104 117
2 22 109 49 5
60 80 61 37 39
82 115 86 21 90

Layer 5:
93 81 75 53 72
73 63 92 48 69
125 25 114 85 38
23 119 59 113 46

105 91 70 99 1

62 88 78 83 3

Akhir state si cube:

Layer 1:
56 68 30 13 123
50 89 27 47 102
18 105 20 103 12
124 58 31 93 35
67 107 69 29 43

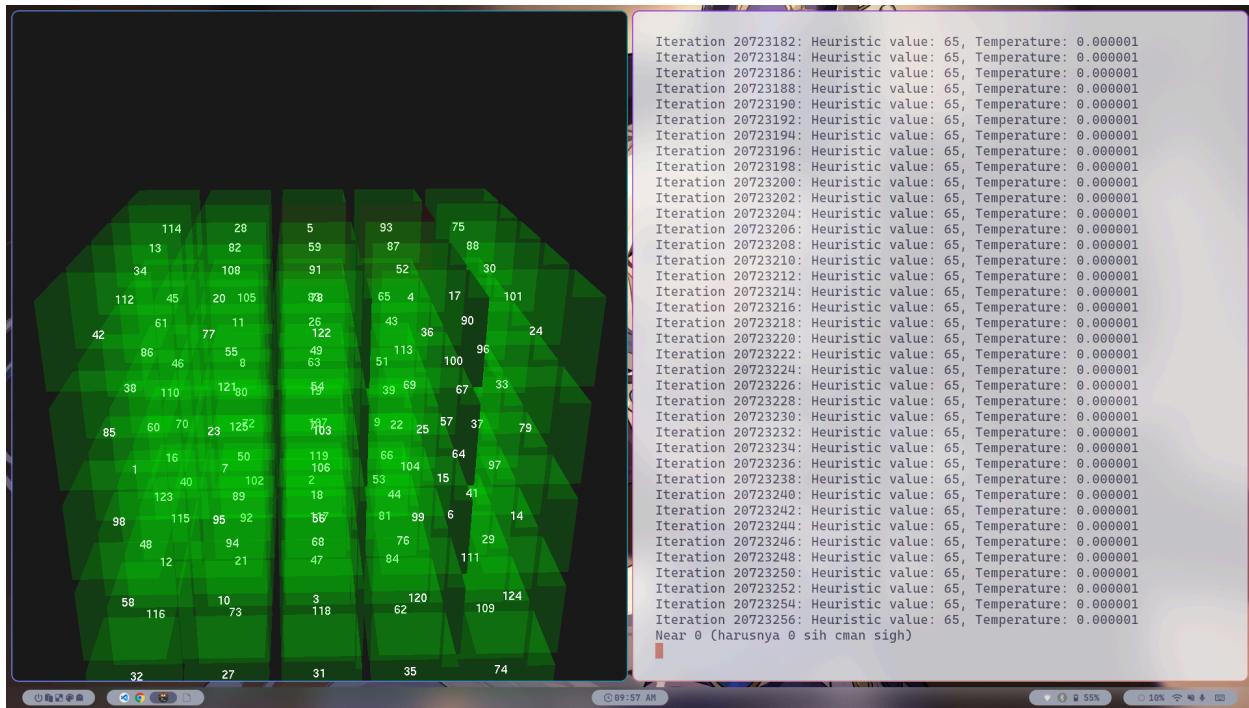
Layer 2:
64 111 57 97 8
19 51 66 84 95
108 62 42 54 49
87 34 74 79 65
37 100 76 24 98

Layer 3:
77 33 118 7 36
88 52 114 55 14
38 112 44 32 4
96 71 17 60 6
109 120 116 122 28

Layer 4:
16 63 45 104 117
2 22 53 81 73
26 80 61 10 39
82 115 86 21 90
9 91 70 99 46

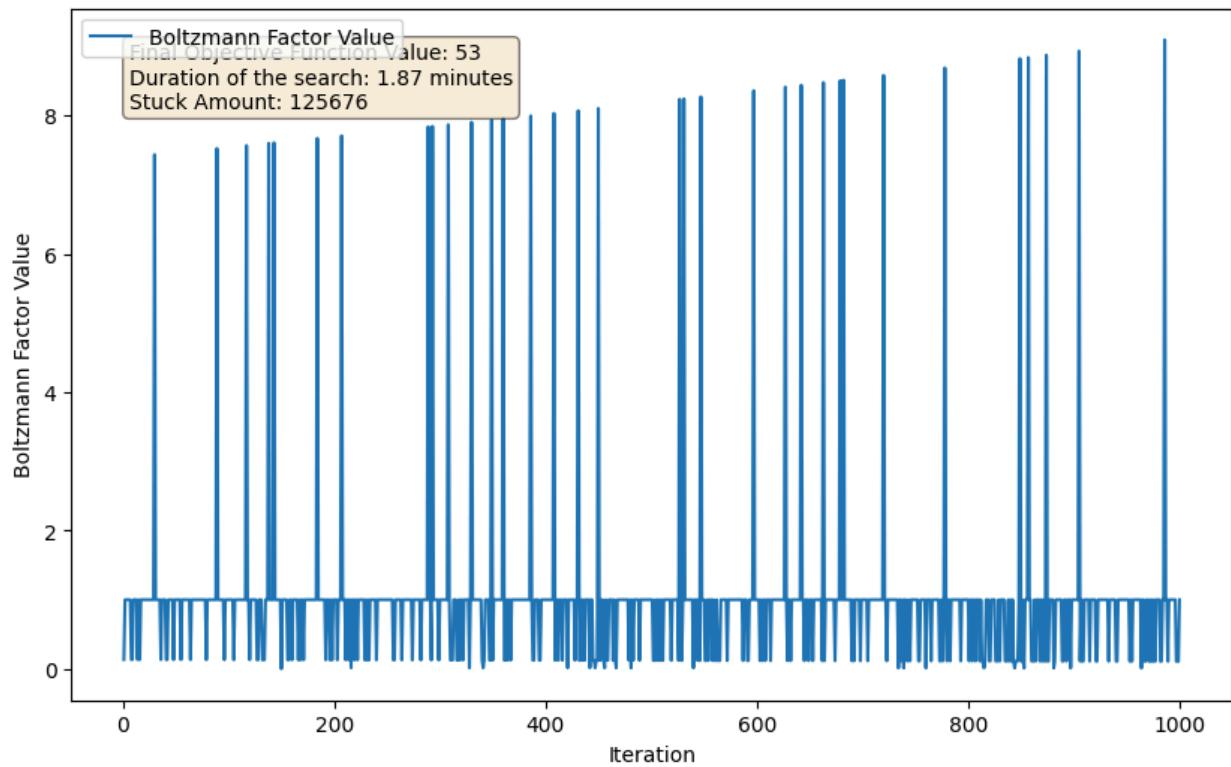
Layer 5:
121 40 75 94 72
5 101 92 48 106
125 25 11 85 15
23 1 59 113 119
41 110 78 83 3

5. Simulated Annealing



1st Experiment:

Boltzmann Factor Value vs Iterations



Inisial state si cube:

Layer 1:
91 75 119 111 7
86 65 31 69 117
24 50 89 17 33
83 113 40 63 20
122 12 124 4 15

Layer 2:
90 73 61 16 67
123 103 121 101 93
34 116 52 29 56
114 1 48 27 99
32 39 102 100 10

Layer 3:
112 43 47 14 53
41 25 6 2 35
74 54 23 66 44
82 88 11 95 8
60 55 76 68 97

Layer 4:
28 70 62 64 22
45 107 108 26 3
79 98 78 36 81
37 85 21 125 49
72 46 18 58 42

Layer 5:
71 38 115 59 92
94 80 120 77 84
51 19 105 30 87
57 106 9 5 118
104 13 109 96 110

Akhir state si cube:

Layer 1:
5 110 109 37 3
8 120 22 125 40
82 2 48 23 79
101 58 66 34 56
119 25 70 57 33

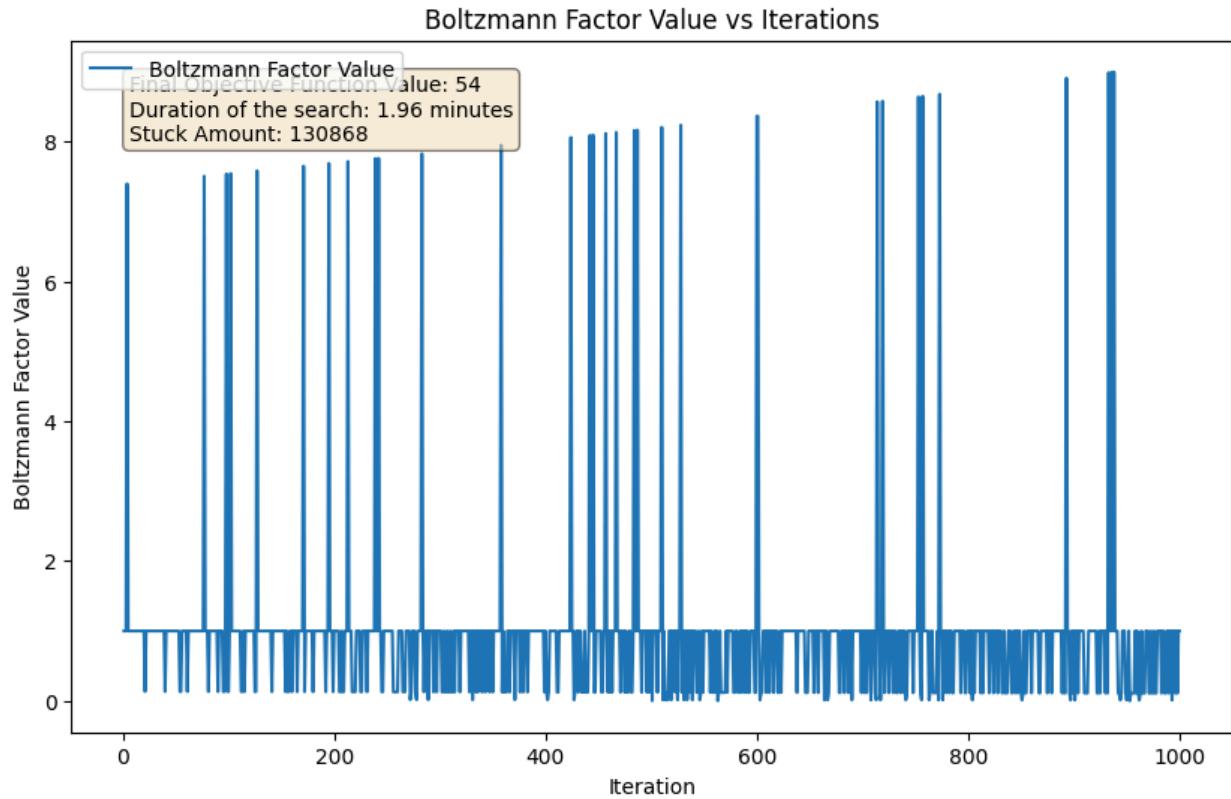
Layer 2:
81 26 118 15 75
106 31 76 99 102
64 69 6 24 35
41 68 32 103 9
10 54 83 74 94

Layer 3:
21 122 90 55 17
97 78 50 18 4
73 111 29 42 71
108 115 80 87 123
16 27 59 113 100

Layer 4:
65 116 51 96 7
85 67 121 1 104
36 105 93 43 38
45 62 30 13 89
84 95 20 39 77

Layer 5:
91 44 63 112 98
19 117 46 72 61
60 28 47 88 92
14 12 107 49 53
86 114 52 124 11

2nd Experiment:



Inisial state si cube:

Layer 1:
17 113 33 9 41
77 10 92 55 58
91 105 38 5 54
60 21 57 18 30
56 107 75 50 19

Layer 4:
63 94 108 48 120
102 42 51 88 47
16 78 13 12 66
3 70 86 115 87
121 84 81 37 14

Layer 2:
85 89 101 106 26
90 82 32 40 25
110 2 93 95 53
11 118 45 4 46
83 64 35 116 59

Layer 5:
99 23 39 112 1
22 20 43 100 79
103 67 76 119 97
74 34 122 123 29
6 68 124 44 27

Layer 3:
114 8 80 49 28
31 24 117 104 109
69 98 65 73 61
52 36 111 72 96
62 125 71 15 7

Akhir state si cube:

Layer 1:
87 84 55 22 67
78 83 34 8 112
90 113 92 6 14
33 12 63 93 42
27 23 71 74 120

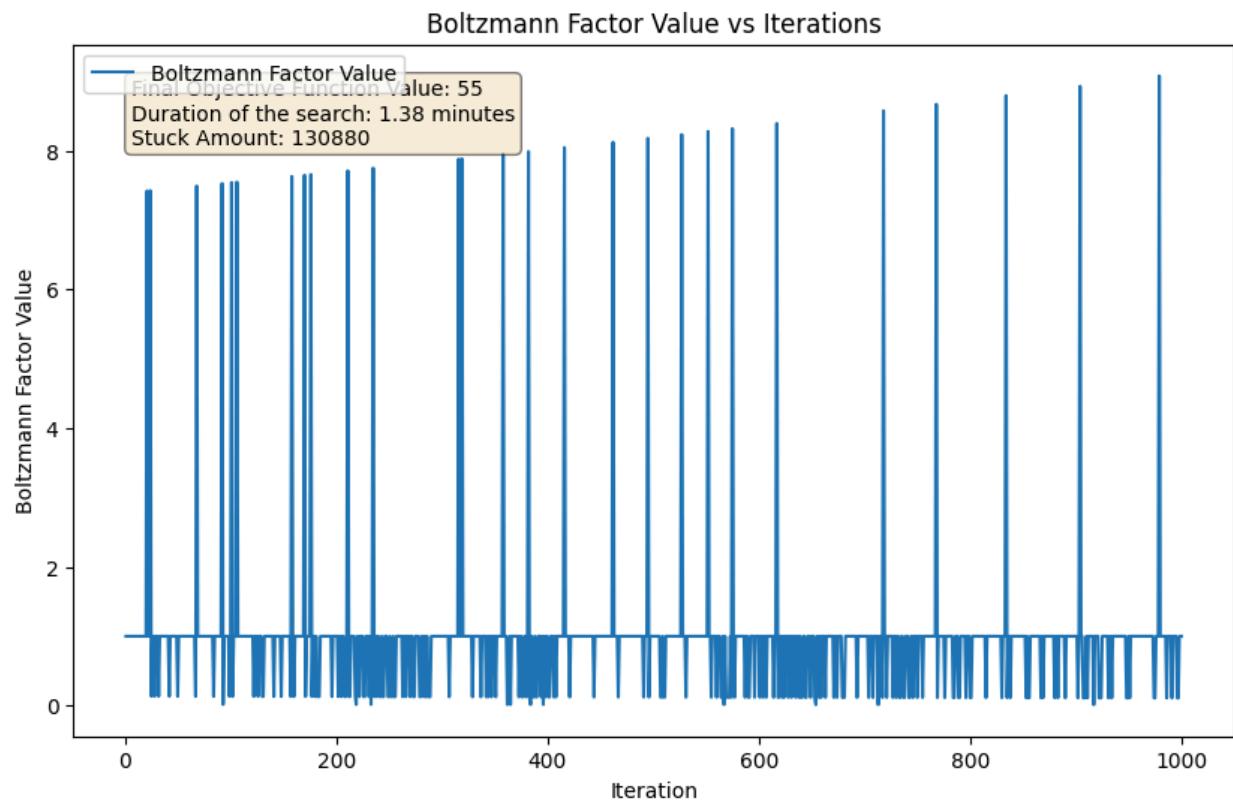
Layer 2:
4 100 107 13 91
80 45 53 73 64
39 11 98 54 81
118 116 17 16 44
109 43 76 52 35

Layer 3:
31 26 86 51 97
111 99 96 19 18
38 70 29 56 122
115 15 103 32 50
20 105 1 65 124

Layer 4:
114 47 57 9 88
30 48 24 94 119
28 49 60 110 68
61 69 108 95 3
82 123 66 7 37

Layer 5:
102 58 10 41 104
46 40 106 121 2
5 72 36 89 125
85 75 62 79 25
77 21 101 117 59

3rd Experiment:



Inisial state si cube:

Layer 1:

21 20 35 50 9
62 42 120 60 69
68 32 119 76 70
12 97 118 7 27
1 73 63 44 112

Layer 2:

6 87 14 18 3
124 43 90 111 4
19 85 98 110 77
114 72 58 59 31
64 47 5 103 75

Layer 3:

74 71 41 46 78
55 109 105 51 104
38 33 99 22 48
102 100 80 10 83
29 45 96 117 115

Layer 4:

37 13 40 2 65
66 113 95 15 34
30 91 93 89 116
82 56 11 53 57
106 49 52 122 121

Layer 5:

86 123 108 84 28
23 36 92 24 125
67 61 8 81 88
25 17 94 54 107
79 26 16 39 101

Akhir state si cube:

Layer 1:

65 7 40 124 79
104 74 10 88 4
60 39 107 76 33
23 89 78 122 22
63 106 80 32 34

Layer 2:

86 95 125 46 87
30 36 83 114 52
15 109 28 11 14
103 5 72 69 66
81 70 9 59 96

Layer 3:

26 55 68 108 58
71 2 27 42 54
61 120 94 17 48
18 101 24 62 110
20 37 102 111 45

Layer 4:

93 84 44 12 82
50 41 97 35 92
85 31 19 105 75
112 123 90 99 115
56 73 116 64 119

Layer 5:

38 43 91 25 118
100 1 98 3 113
53 16 67 121 6
77 13 51 117 57
47 29 8 49 21

6. Genetic Algorithm

```
File Edit Selection View Go Run Terminal Help
ga.c gen-1.txt main.c makefile ...
Tucil1-DIA-IF3070 > src > Parents > gen-1.txt
1 4;64:18 72:119 98:51 11:50 22:80 63:74 119:76
2 3:30:4 119:76 54:94 36:12 57:13 122:6 58:24 8
3 3:118:121 95:116 119:76 79:58 123:77 18:103 1
4 3:56:38 113:79 38:6 117:97 21:69 61:23 64:18
5 3:18:103 72:119 100:8 64:18 7:98 4:22 75:101
6 3:40:112 58:24 3:55 79:58 77:107 99:113 78:10
7 3:36:12 107:35 1:3 16:47 101:118 93:11 49:11
8 3:91:89 22:80 10:25 107:35 49:11:11 14:6 47:75
9 3:21:69 27:55 35:45 5:68 18:103 62:42 84:39 4
10 3:27:56 23:14 104:87 37:108 71:123 118:121 74
11 3:118:121 13:49 69:3 9:0 26:106 34:110 96:96
12 3:51:82 56:38 38:6 75:101 100:8 64:18 55:7 5
13 3:13:49 86:90 55:7 41:92 51:82 65:93 102:36 1
14 3:6:17 53:9 64:18 74:115 21:69 4:22 10:25 29
15 3:124:40 39:27 99:113 53:59 101:118 84:39 107
16 3:77:107 50:8 121:44 123:77 42:53 110:47 46:
17 2:8:54 76:5 72:119 80:72 106:62 117:97 123:77
18 2:20:81 38:64 96:96 23:14 62:42 122:6 92:105
19 2:86:90 96:96 3:55 11:50 109:16 110:47 118:12
20 2:16:66 114:48 82:104 29:117 79:58 24:60 12:2
21 2:44:122 88:41 110:47 56:38 16:66 9:0 74:115
22 2:107:35 122:6 64:18 31:15 110:47 10:25 121:4
23 2:105:28 92:185 95:116 162:36 72:119 45:61 11
24 2:11:120 8:54 112:31 10:18:7 47:5 101:73:46 23:
25 2:31:15 78:102 33:88 65:93 112:1 62:42 53:59
26 2:119:76 25:99 105:28 93:11 82:184 55:7 109:1
27 2:108:100 32:21 119:76 17:109 44:122 43:19 83
28 2:73:46 66:3 70:8 33:88 85:85 77:107 68:2 5
29 2:121:44 44:2 122:98:51 73:46 3:55 54:94 35:45
30 2:118:121 51:82 108:100 1:63 110:47 73:46 36:
31 2:19:84 50:8 73:119 57:13 45:61 60:26 30:4 4
32 2:22:80 108:100 4:22 15:37 78:102 31:15 33:88
33 2:102:19 80:72 86:90 101:118 113:70 114:102 126
Tucil1-DIA-IF3070 > src > Parents > gen-2.txt
1 4;97:31 6:23 63:1 59:6 12:95 79:32 81:29 43:7
2 4:68:23 12:19 19:57 36:24 120:113 14:6 56:7 6
3 4;68:43 80:39 27:31 4:18 28:113 66:105 43:1 4
4 4:97:23 86:63 20:40 65:33 70:40 16:59 48:7:4
5 4:50:23 74:52 39:123 9:75 61:60 95:43 14:91 3
6 4:89:37 5:6 96:52 54:33 122:23 114:39 57:19 4
7 4:3:121 98:7 95:43 15:89 18:76 51:78 9:45 31:
8 4:109:49 72:32 102:120 4:99 64:35 6:44 34:87
9 4:122:6 83:51 45:107 103:33 0:33 116:93 80:7
10 4:55:57 58:23 11:23 61:1 82:23 10:34 94:83 16
11 4:51:23 111:43 37:85 85:123 18:76 67:60 101:8
12 4:32:30 25:45 118:59 23:52 27:22 49:50 59:23
13 4:13:117 44:7 70:55 23:24 83:7 104:3 42:3 87:19
14 4:49:32 100:107 33:6 38:59 48:105 36:7 52:33
15 4:80:31 108:31 64:1 93:69 95:76 97:121 27:60
16 4:119:65 113:23 67:31 28:18 83:67 52:7 62:79 47:76
17 4:61:52 13:23 67:33 29:11 11:113 70:23 105:8
18 4:75:67 26:105 48:107 37:19 52:70 45:75 54:7
19 4:95:83 22:32 4:66 46:7 3:39 79:60 23:33 76:2
20 4:15:113 3:76 57:6 72:113 124:93 8:23 12:120
21 4:62:7 45:5 59:1 7:10 63:60 23:29 32:95 124:43 99:
22 4:20:45 75:31 68:32 118:83 27:76 0:107 79:34
23 4:110:111 112:60 78:110 19:23 12:11 86:23 6:1
24 4:3:70 12:91 29:32 48:51 98:110 103:19 9:75 1
25 4:65:37 7:113 24:23 26:23 121:30 81:40 52:7 8
26 4:7:7 74:78 72:123 88:23 75:63 107:35 30:44 2
27 4:65:111 86:57 22:95 44:43 95:7 114:45 61:57
28 4:74:7 4:51 25:115 72:7 54:76 14:45 120:59:8
29 4:39:121 116:78 51:34 11:33 3:67 8:60 29:7 22
30 4:118:60 113:77 63:7 47:11 121:60 48:24 73:83
31 4:34:113 33:1 100:43 95:34 69:75 23:23 15:35
32 4:50:121 5:19 6:17 19:43 1:17 82:121 118:31 1
33 4:101:7 112:117 118:22 106:77 60:27 115:76 55:
```

Pada eksperimen ini, algoritma genetika diterapkan dengan populasi awal sebanyak 1,000 kromosom, tetapi ditemukan bahwa banyak kromosom memiliki nilai *heuristic* sebesar 0 yang menunjukkan bahwa solusi-solusi awal tersebut sangat jauh dari optimal. Hanya satu kromosom dalam populasi awal yang memiliki nilai *heuristic* sebesar 4.

Setelah 2,000 generasi, seluruh kromosom dalam populasi mencapai nilai heuristik yang tidak melebihi nilai tersebut. Hal ini menunjukkan bahwa algoritma genetika tidak berhasil menemukan solusi yang lebih optimal di luar titik tersebut.

Seberapa dekat tiap-tiap algoritma bisa mendekati global optima dan mengapa hasilnya demikian?

Algoritma *Simulated Annealing* paling mendekati *global optima* dengan hasil terbaik mencapai nilai ($h = 55$). Hal ini disebabkan oleh mekanisme *cooling* yang memungkinkan algoritma ini untuk keluar dari jebakan *local optima* dan terus mencari solusi yang lebih baik. *Hill Climbing with Sideways Move* juga cukup efektif dengan hasil terbaik ($h = 44$), karena memungkinkan pergerakan lateral untuk keluar dari jebakan *local optima*. *Random Restart* juga menunjukkan kemampuan yang baik dalam mendekati *global optima* dengan melakukan *restart* secara acak meskipun membutuhkan waktu lebih lama. *Steepest Ascent Hill Climbing* dan *Stochastic* cenderung terjebak di *local optima* karena kurangnya mekanisme untuk keluar dari jebakan tersebut, sehingga hasilnya tidak sebaik dua algoritma lainnya.

Bagaimana perbandingan hasil pencarian tiap-tiap algoritma dengan algoritma local search yang lain?

Simulated Annealing menunjukkan hasil yang paling efektif dalam mendekati *global optima* dibandingkan dengan algoritma lainnya. *Random Restart* juga efektif, tetapi membutuhkan waktu pencarian yang lebih lama. *Hill Climbing with Sideways Move* menunjukkan hasil yang baik dengan nilai heuristik yang lebih tinggi dibandingkan dengan *Steepest Ascent Hill Climbing* dan *Stochastic*. *Steepest Ascent Hill Climbing* cepat dalam menemukan *local maxima* tetapi sering terjebak di *local optima*.

Secara keseluruhan, *Simulated Annealing* dan *Random Restart* lebih unggul dalam mendekati *global optima* dibandingkan dengan *Random Restart* dan *Stochastic*.

Bagaimana perbandingan durasi proses pencarian tiap algoritma relatif terhadap algoritma lainnya?

Durasi pencarian *Simulated Annealing* berada di tengah-tengah, sekitar 1.38 hingga 1.96 menit, yang cukup efisien mengingat hasil yang dicapai. *Random Restart* membutuhkan waktu yang lebih lama, sekitar 2.55 hingga 3.06 menit karena melakukan restart secara acak. *Hill Climbing with Sideways Move* memiliki durasi pencarian yang singkat, sekitar 0.336 hingga 0.433 detik, dan menunjukkan hasil yang baik. *Steepest Ascent Hill Climbing* memiliki durasi pencarian yang sangat singkat, sekitar 0.25 hingga 0.29 detik, tetapi sering terjebak di *local optima*. *Stochastic* juga memiliki durasi pencarian yang relatif singkat, sekitar 0.09 hingga 0.17 menit.

Seberapa konsisten hasil akhir yang didapatkan dari tiap-tiap eksperimen yang dilakukan?

Simulated Annealing menunjukkan konsistensi yang tinggi dengan hasil yang mendekati *global optima* pada setiap eksperimen. *Random Restart* juga cukup konsisten, meskipun membutuhkan waktu lebih lama. *Hill Climbing with Sideways Move* menunjukkan hasil yang cukup konsisten dengan nilai heuristik yang lebih tinggi dibandingkan dengan *Steepest Ascent Hill Climbing*. *Steepest Ascent Hill Climbing* menunjukkan hasil yang cukup konsisten tetapi sering terjebak di *local optima*. *Stochastic* memiliki hasil yang kurang konsisten dan lebih

bervariasi karena pemilihan elemen acak dalam algoritma ini dapat menghasilkan hasil yang berbeda-beda pada setiap eksperimen.

Secara keseluruhan, *Simulated Annealing* adalah yang paling konsisten dalam menghasilkan hasil yang mendekati *global optima*.

Pemilihan *Local Search* Terbaik

Pada akhirnya, kami menentukan *local search* terbaik merupakan *Simulated Annealing*. Setelah kami *run* berkali-kali, *Simulated Annealing* konsisten selalu memiliki skor tertinggi. Berikut penjelasan mengapa *search* tersebut dapat memberikan hasil terbaik pada permasalahan tersebut bagi kami.

Simulated annealing memberikan hasil yang paling baik dibandingkan algoritma pencarian *local search* lainnya karena kemampuannya untuk keluar dari jebakan *local maximum*. Pada algoritma seperti *Steepest Ascent Hill Climbing*, *Hill Climbing with Sideways Move*, *Random Restart*, *Stochastic*, atau *Genetic*, pencarian akan langsung berhenti saat mencapai *local maximum* atau setidaknya cenderung tidak mau untuk mencoba melakukan **gerakan** yang lebih “buruk”/ skor *heuristic* yang lebih rendah .

Ini akan membuat algoritma sering terjebak pada solusi yang suboptimal. *Simulated Annealing*, di sisi lain, menggunakan konsep "temperatur" yang memungkinkan algoritma untuk menerima solusi yang lebih buruk dengan probabilitas tertentu, terutama pada tahap awal ketika temperatur masih tinggi. Ini memungkinkan eksplorasi yang lebih luas dari ruang solusi dan membantu algoritma untuk tidak terjebak di *local maximum*. Seiring waktu, temperatur menurun, membuat algoritma lebih selektif dalam menerima solusi baru, mirip dengan **hill-climbing** dan algoritma sudah memiliki kesempatan lebih baik untuk menemukan jalur menuju solusi *global maximum*.

Oleh karena itu, kombinasi antara eksplorasi awal dan pemilihan *neighbor* yang lebih selektif di akhir membuat *Simulated Annealing* lebih efektif dalam menemukan solusi terbaik dalam masalah pencarian yang kompleks, seperti *magic cube*. Dan menurut kami, khususnya pada permasalahan dengan jumlah kombinasi variasi ($125!$ pada *magic cube* $5 \times 5 \times 5$) yang banyak. Jumlah yang banyak ini diikuti dengan *constraint magic cube* yang cukup rumit akan membuat banyak sekali *state-state* kubus yang rendah tidak optimal. Pada akhirnya algoritma tersebut dapat membuat kubus *bergerak* kepada solusi yang lebih jelek (*shoulder* pada grafik) pada awal-awal dengan harapan dapat langsung meningkat kepada *global maximum*. Berbeda dengan algoritma lain yang langsung memilih solusi “terbaik” tanpa mengeksplorasi kemungkinan-kemungkinan *state* lainnya.

3. Kesimpulan dan Saran

Berdasarkan hasil analisis yang telah dilakukan, terlihat bahwa algoritma local search terbaik ada pada simulated annealing dengan nilai heuristik tertinggi yang dicapai berkisar di angka 50. Bahkan di beberapa percobaan di luar 3 eksperimen yang dilakukan diatas, simulated annealing mencapai 67.

Kriteria terbaik dipilih berdasarkan pendekatan nilai yang dihasilkan algoritma terhadap global maksima dan juga konsistensi nilai yang dihasilkan. Namun jika kita mengkonsiderasi beban komputasi yang diperlukan dan juga waktu yang diperlukan, algoritma ini tidak bisa dibilang baik dari sisi ini, ini disebabkan oleh algoritma tergantung pada initial temperature, jumlah iterasi, dan cooling rate yang dipilih. Jika dibandingkan dengan algoritma local search lainnya, seperti stochastic/steepest ascent/random restart, algoritma simulated annealing termasuk lebih lambat. Oleh sebab untuk mencapai nilai tertinggi, simulated annealing memerlukan waktu/iterasi yang cukup lama agar mengurangi “kecenderungan” pindah ke *neighbour* yang lebih rendah/sama. Dengan kata lain, agar bisa mencapai nilai yang bagus, harus mendekati temperatur 0, dan ini membutuhkan waktu yang cukup lama dengan *setup* yang kami buat (i.e. initial temperature, cooling rate, scheduling)

Algoritma simulated annealing dapat dioptimalisasi lebih untuk memberikan *heuristic value* / nilai *objective function* yang lebih baik. Saran perbaikan dapat berupa mencari fungsi *scheduling* yang lebih efektif, menentukan *cooling rate* yang lebih baik, menentukan inisial temperature agar lebih tinggi (untuk memungkinkan agen lebih banyak mengeksplorasi spektrum *neighbour* yang ada, i.e. berpindah ke *neighbour* yang memiliki nilai heuristik yang sama / lebih jelek) atau temperature rendah (untuk memberi kecenderungan agen untuk fokus terhadap *neighbour* yang lebih baik saja).

Akan tetapi secara keseluruhan, saran yang bisa didapatkan adalah sebagai berikut. Untuk kembali melakukan eksperimen terus menerus. Agar bisa menentukan kira-kira parameter apa yang harus diperbaiki. Ini agar memastikan lingkungan agen kita berada merupakan lingkungan yang paling optimal.

Selain itu, kata-kata terakhir dari kami adalah untuk permohonan maaf kode Genetic Algorithm yang bisa dibilang belum optimal. Akibat beberapa kendala, kode tersebut tidak bisa diimplementasikan secara maksimal.

4. Pembagian Tugas

Nama	NIM	Tugas
Matthew Lim	18222005	Dokumen
Alessandro Jusack H	18222025	Source Code (Steepest Ascent Hill Climbing, Random Restart Hill Climbing, Stochastic Hill Climbing, Simulated Annealing)
Satria Wisnu Wibowo	18222087	Dokumen
Wisyendra Lunarmalam	18222095	Source Code (Genetic Algorithm)

5. Referensi

Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson. [Penjelasan mengenai *local search* secara umum.]

Trump, J. *Magic Cubes: Theory and Examples*. Trump.de. Retrieved from <https://www.trump.de/magic-squares/magic-cubes/cubes-1.html> [Penjelasan mengenai *magic cube* secara umum.]

An Optimal Cooling Schedule Using a Simulated Annealing Based Approach. Scientific Research Publishing. Retrieved from <https://www.scirp.org/journal/paperinformation?paperid=78834> [Formula scheduling untuk *simulated annealing*.]

Simulated Annealing. [Video]. YouTube. Retrieved from <https://youtu.be/C86j1AoMRr0?si=iYXblq0Brkog2JE3> [Penjelasan tentang *simulated annealing*.]

Repository Proyek: Tucil1-DIA-IF3070. *Tugas Kecil IF3070 Dasar-dasar Intelelegensi Artifisial*. GitHub Repository. Retrieved from <https://github.com/wisye/Tucil1-DIA-IF3070>