**Genetic Algorithm for Image Generation Using Squares**

Soumik Saha - 2105672

**Abstract**

This report presents a genetic algorithm (GA) for image generation using squares. The code generates an image composed of a predefined number of squares, each characterized by its position, size, and color. The GA aims to evolve a population of images to closely resemble a target image through fitness evaluation, selection, crossover, and mutation operations.

**Introduction**

Image generation using genetic algorithms is a complex problem that involves optimizing a population of images to mimic a given target image. In this project, we implement a GA to generate images using squares as building blocks. The following sections provide an overview of the key components of the code.

**Components**

1. Initialization

The code initializes a population of images, each composed of a specified number of squares. The squares' positions, sizes, and colors are initialized randomly.

2. Fitness Function

A fitness function measures how well an image matches the target image. The provided fitness function calculates the sum of squared differences in color values between corresponding pixels in the generated and target images. The lower the difference, the higher the fitness.

3. Selection

The code employs tournament selection to choose the best-performing images from the population based on their fitness scores. The tournament size can be adjusted according to the desired selection pressure.

4. Crossover

Crossover operations combine squares from two parent images to create a child image. A one-point crossover is employed in the code, where a random crossover point is chosen, and squares from both parents are alternately selected.

5. Mutation

Mutation introduces small random changes to the child images. The code randomly selects a square in an image and mutates its position or size.

6. Termination

The algorithm runs for a specified number of generations, and the best-performing image is selected as the output. Optionally, you can add a termination criterion based on a target fitness score or other relevant criteria.

**Results**

The algorithm iteratively refines the population of images to generate an image that closely matches the target image. The fitness values are monitored across generations, and the best image's fitness score is displayed. The generated image can be visualized, and further customization can be applied as needed.

**Conclusion**

The genetic algorithm for image generation using squares presents a basic framework for evolving images to resemble a target image. The provided code can serve as a starting point for projects involving image synthesis, but further customization and optimization may be required based on the specific application's complexity and requirements.

**Future Work**

Future work could include enhancing the fitness function with perceptual metrics, exploring parallelism to speed up optimization, and refining the selection, crossover, and mutation strategies. Additionally, incorporating constraints, such as non-overlapping squares, may be necessary for more complex image generation tasks.

**Code**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <math.h>

#define IMAGE\_WIDTH 200

#define IMAGE\_HEIGHT 200

#define N 50

#define POPULATION\_SIZE 100

#define MAX\_GENERATIONS 1000

#define MUTATION\_RATE 0.01

#define CROSSOVER\_RATE 0.7

typedef struct {

int x, y;

int size;

int color;

} Square;

typedef struct {

Square squares[N];

double fitness;

} Image;

void initialize\_population(Image population[POPULATION\_SIZE]) {

for (int i = 0; i < POPULATION\_SIZE; i++) {

for (int j = 0; j < N; j++) {

population[i].squares[j].x = rand() % IMAGE\_WIDTH;

population[i].squares[j].y = rand() % IMAGE\_HEIGHT;

population[i].squares[j].size = rand() % 20 + 10;

population[i].squares[j].color = rand();

}

}

}

double color\_difference(int color1, int color2) {

int r1 = (color1 >> 16) & 0xFF;

int g1 = (color1 >> 8) & 0xFF;

int b1 = color1 & 0xFF;

int r2 = (color2 >> 16) & 0xFF;

int g2 = (color2 >> 8) & 0xFF;

int b2 = color2 & 0xFF;

double diff = sqrt((r1 - r2) \* (r1 - r2) + (g1 - g2) \* (g1 - g2) + (b1 - b2) \* (b1 - b2));

return diff;

}

double evaluate\_fitness(Image \*image, Image \*target) {

double fitness = 0.0;

for (int i = 0; i < N; i++) {

int x = image->squares[i].x;

int y = image->squares[i].y;

int color = image->squares[i].color;

int target\_color = target->squares[i].color;

double color\_diff = color\_difference(color, target\_color);

fitness += 1000.0 / (1.0 + color\_diff);

}

return fitness;

}

void selection(Image population[POPULATION\_SIZE], Image selected[POPULATION\_SIZE]) {

for (int i = 0; i < POPULATION\_SIZE; i++) {

int tournament\_size = 5;

int best\_index = rand() % POPULATION\_SIZE;

for (int j = 1; j < tournament\_size; j++) {

int random\_index = rand() % POPULATION\_SIZE;

if (population[random\_index].fitness > population[best\_index].fitness) {

best\_index = random\_index;

}

}

selected[i] = population[best\_index];

}

}

void crossover(Image parent1, Image parent2, Image \*child) {

int crossover\_point = rand() % N;

for (int i = 0; i < N; i++) {

if (i < crossover\_point) {

child->squares[i] = parent1.squares[i];

} else {

child->squares[i] = parent2.squares[i];

}

}

}

void mutate(Image \*image) {

int index = rand() % N;

int mutation\_type = rand() % 2;

if (mutation\_type == 0) {

image->squares[index].x = rand() % IMAGE\_WIDTH;

image->squares[index].y = rand() % IMAGE\_HEIGHT;

} else {

image->squares[index].size = rand() % 20 + 10;

}

}

int main() {

srand(time(NULL));

Image target\_image;

for (int i = 0; i < N; i++) {

target\_image.squares[i].x = rand() % IMAGE\_WIDTH;

target\_image.squares[i].y = rand() % IMAGE\_HEIGHT;

target\_image.squares[i].size = rand() % 20 + 10;

target\_image.squares[i].color = rand();

}

Image population[POPULATION\_SIZE];

initialize\_population(population);

for (int generation = 0; generation < MAX\_GENERATIONS; generation++) {

for (int i = 0; i < POPULATION\_SIZE; i++) {

population[i].fitness = evaluate\_fitness(&population[i], &target\_image);

}

Image selected[POPULATION\_SIZE];

selection(population, selected);

Image new\_population[POPULATION\_SIZE];

for (int i = 0; i < POPULATION\_SIZE; i++) {

if (rand() < CROSSOVER\_RATE \* RAND\_MAX) {

int parent1 = rand() % POPULATION\_SIZE;

int parent2 = rand() % POPULATION\_SIZE;

crossover(selected[parent1], selected[parent2], &new\_population[i]);

} else {

new\_population[i] = selected[i];

}

if (rand() < MUTATION\_RATE \* RAND\_MAX) {

mutate(&new\_population[i]);

}

}

memcpy(population, new\_population, sizeof(new\_population));

printf("Generation %d: Best Fitness = %lf\n", generation, selected[0].fitness);

}

return 0;

}