CIA-2 DAA

NAME: A.S.Deepesh kumar

Roll no: 21011102027

1)implementation of genetic algorithm for Travelling Salesman Problem (TSP) in c:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define POP\_SIZE 100

#define MAX\_GENERATIONS 1000

#define MUTATION\_RATE 0.01

int distances[5][5] = {

{0, 10, 15, 20, 25},

{10, 0, 35, 25, 20},

{15, 35, 0, 30, 10},

{20, 25, 30, 0, 35},

{25, 20, 10, 35, 0}

};

struct Individual {

int genes[5];

int fitness;

};

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void shuffle(int\* arr, int size) {

srand(time(NULL));

for (int i = size - 1; i > 0; i--) {

int j = rand() % (i + 1);

swap(&arr[i], &arr[j]);

}

}

int calculate\_fitness(struct Individual\* individual) {

int fitness = 0;

for (int i = 0; i < 4; i++) {

fitness += distances[individual->genes[i]][individual->genes[i+1]];

}

fitness += distances[individual->genes[4]][individual->genes[0]];

return fitness;

}

void crossover(struct Individual\* parent1, struct Individual\* parent2, struct Individual\* child1, struct Individual\* child2) {

int pivot = rand() % 4 + 1;

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

child1->genes[i] = parent1->genes[i];

child2->genes[i] = parent2->genes[i];

}

for (int i = pivot+1; i < 5; i++) {

child1->genes[i] = parent2->genes[i];

child2->genes[i] = parent1->genes[i];

}

}

void mutate(struct Individual\* individual) {

for (int i = 0; i < 5; i++) {

if ((double)rand() / (double)RAND\_MAX < MUTATION\_RATE) {

int j = rand() % 5;

swap(&individual->genes[i], &individual->genes[j]);

}

}

}

void generate\_population(struct Individual\* population) {

int base\_genes[5] = {0, 1, 2, 3, 4};

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

shuffle(base\_genes, 5);

for (int j = 0; j < 5; j++) {

population[i].genes[j] = base\_genes[j];

}

population[i].fitness = calculate\_fitness(&population[i]);

}

}

int select\_parent(struct Individual\* population) {

int sum\_fitness = 0;

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

sum\_fitness += population[i].fitness;

}

}

}

return POP\_SIZE - 1;

}

1. 2) Cultural Algorithm for TSP in c:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define POP\_SIZE 100

#define MAX\_GENERATIONS 1000

#define MUTATION\_RATE 0.01

#define LEARNING\_RATE 0.1

#define NUM\_CULTURES 5

#define CULTURAL\_DIFFERENCE\_THRESHOLD 5

int distances[5][5] = {

{0, 10, 15, 20, 25},

{10, 0, 35, 25, 20},

{15, 35, 0, 30, 10},

{20, 25, 30, 0, 35},

{25, 20, 10, 35, 0}

};

struct Individual {

int genes[5];

int fitness;

};

struct Culture {

struct Individual population[POP\_SIZE];

int best\_fitness;

int num\_generations\_without\_improvement;

};

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void shuffle(int\* arr, int size) {

srand(time(NULL));

for (int i = size - 1; i > 0; i--) {

int j = rand() % (i + 1);

swap(&arr[i], &arr[j]);

}

}

int calculate\_fitness(struct Individual\* individual) {

int fitness = 0;

for (int i = 0; i < 4; i++) {

fitness += distances[individual->genes[i]][individual->genes[i+1]];

}

fitness += distances[individual->genes[4]][individual->genes[0]];

return fitness;

}

void crossover(struct Individual\* parent1, struct Individual\* parent2, struct Individual\* child1, struct Individual\* child2) {

int pivot = rand() % 4 + 1;

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

child1->genes[i] = parent1->genes[i];

child2->genes[i] = parent2->genes[i];

}

for (int i = pivot+1; i < 5; i++) {

child1->genes[i] = parent2->genes[i];

child2->genes[i] = parent1->genes[i];

}

}

void mutate(struct Individual\* individual) {

for (int i = 0; i < 5; i++) {

if ((double)rand() / (double)RAND\_MAX < MUTATION\_RATE) {

int j = rand() % 5;

swap(&individual->genes[i], &individual->genes[j]);

}

}

}

void generate\_population(struct Individual\* population) {

int base\_genes[5] = {0, 1, 2, 3, 4};

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

shuffle(base\_genes, 5);

for (int j = 0; j < 5; j++) {

population[i].genes[j] = base\_genes[j];

}

population[i].fitness = calculate\_fitness(&population[i]);

}

}

int select\_parent(struct Individual\* population) {

int sum\_fitness = 0;

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

sum\_fitness += population[i].fitness;

}

int target\_fitness = rand() % sum\_fitness + 1;

int fitness\_so\_far = 0;

}

}

}

return POP\_SIZE - 1;

}

1. Genetic algorithm for Sine function :

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <time.h>

#define POP\_SIZE 100

#define MAX\_GENERATIONS 1000

#define MUTATION\_RATE 0.01

#define CROSSOVER\_RATE 0.8

#define ELITISM\_RATE 0.1

double target\_function(double x) {

return sin(x);

}

struct Individual {

double x;

double fitness;

};

void shuffle(struct Individual\* arr, int size) {

srand(time(NULL));

for (int i = size - 1; i > 0; i--) {

int j = rand() % (i + 1);

struct Individual temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

double calculate\_fitness(double x) {

return 1.0 / (1.0 + fabs(target\_function(x) - sin(x)));

}

void crossover(struct Individual\* parent1, struct Individual\* parent2, struct Individual\* child1, struct Individual\* child2) {

double alpha = (double)rand() / (double)RAND\_MAX;

child1->x = alpha \* parent1->x + (1.0 - alpha) \* parent2->x;

child2->x = alpha \* parent2->x + (1.0 - alpha) \* parent1->x;

}

void mutate(struct Individual\* individual) {

double new\_x = individual->x + ((double)rand() / (double)RAND\_MAX - 0.5) \* 2.0;

if (new\_x >= 0 && new\_x <= M\_PI) {

individual->x = new\_x;

}

}

void generate\_population(struct Individual\* population) {

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

double x = ((double)rand() / (double)RAND\_MAX) \* M\_PI;

population[i].x = x;

population[i].fitness = calculate\_fitness(x);

}

}

void sort\_population(struct Individual\* population) {

for (int i = 0; i < POP\_SIZE - 1; i++) {

for (int j = i+1; j < POP\_SIZE; j++) {

if (population[i].fitness < population[j].fitness) {

struct Individual temp = population[i];

population[i] = population[j];

population[j] = temp;

}

}

}

}

void elitism(struct Individual\* population, struct Individual\* offspring) {

int num\_elites = ELITISM\_RATE \* POP\_SIZE;

for (int i = 0; i < num\_elites; i++) {

offspring[i] = population[i];

}

}

void select\_parents(struct Individual\* population, struct Individual\* parent1, struct Individual\* parent2) {

shuffle(population, POP\_SIZE);

double best\_fitness = 0.0;

int best\_index = 0;

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

if (population[i].fitness > best\_fitness) {

best\_fitness = population[i].fitness;

best\_index = i;

}

}

\*parent1 = population[best\_index];

double second\_best\_fitness = 0.0;

int second\_best\_index = 0;

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

if (population[i].fitness > second\_best\_fitness && i != best\_index) {

second\_best\_fitness = population[i].fitness;

second\_best\_index = i;

}

4) Particle Swarm Optimization for the sine function in c :

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <time.h>

#define POP\_SIZE 50

#define MAX\_ITERATIONS 1000

#define C1 2.0

#define C2 2.0

#define W 0.7

double target\_function(double x) {

return sin(x);

}

struct Particle {

double x;

double velocity;

double pbest\_x;

double pbest\_fitness;

};

void initialize\_particles(struct Particle\* particles) {

srand(time(NULL));

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

double x = ((double)rand() / (double)RAND\_MAX) \* M\_PI;

particles[i].x = x;

particles[i].velocity = ((double)rand() / (double)RAND\_MAX - 0.5) \* 2.0;

particles[i].pbest\_x = x;

particles[i].pbest\_fitness = target\_function(x);

}

}

double calculate\_fitness(double x) {

return 1.0 / (1.0 + fabs(target\_function(x) - sin(x)));

}

void update\_particle(struct Particle\* particle, double gbest\_x) {

double r1 = (double)rand() / (double)RAND\_MAX;

double r2 = (double)rand() / (double)RAND\_MAX;

particle->velocity = W \* particle->velocity + C1 \* r1 \* (particle->pbest\_x - particle->x) + C2 \* r2 \* (gbest\_x - particle->x);

particle->x += particle->velocity;

if (particle->x < 0) {

particle->x = 0;

} else if (particle->x > M\_PI) {

particle->x = M\_PI;

}

double fitness = calculate\_fitness(particle->x);

if (fitness > particle->pbest\_fitness) {

particle->pbest\_fitness = fitness;

particle->pbest\_x = particle->x;

}

}

void update\_swarm(struct Particle\* particles, double\* gbest\_fitness, double\* gbest\_x) {

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

if (calculate\_fitness(particles[i].x) > \*gbest\_fitness) {

\*gbest\_fitness = calculate\_fitness(particles[i].x);

\*gbest\_x = particles[i].x;

}

}

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

update\_particle(&particles[i], \*gbest\_x);

}

}

void print\_particle(struct Particle\* particle) {

printf("x = %f, velocity = %f, pbest\_x = %f, pbest\_fitness = %f\n", particle->x, particle->velocity, particle->pbest\_x, particle->pbest\_fitness);

}

void print\_swarm(struct Particle\* particles) {

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

printf("Particle %d: ", i);

print\_particle(&particles[i]);

}

}

int main() {

struct Particle particles[POP\_SIZE];

initialize\_particles(particles);

double gbest\_fitness = 0.0;

double gbest\_x = 0.0;

for (int i = 0; i < MAX\_ITERATIONS; i++) {

update\_swarm(particles, &gbest\_fitness, &gbest\_x);

printf("Iteration %d: gbest\_fitness = %f, gbest\_x = %f)

5) Ant Colony Optimization for finding the shortest path in a graph:

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <math.h>

#include <time.h>

#define MAX\_NODES 100

#define MAX\_EDGES 1000

#define MAX\_ANTS 50

#define MAX\_ITERATIONS 1000

#define ALPHA 1.0

#define BETA 5.0

#define RHO 0.5

#define Q 100.0

int num\_nodes, num\_edges;

int edges[MAX\_EDGES][3];

double pheromone[MAX\_NODES][MAX\_NODES];

double heuristic[MAX\_NODES][MAX\_NODES];

int visited[MAX\_NODES];

int path[MAX\_NODES];

double path\_length;

double euclidean\_distance(int node1, int node2) {

double x1 = edges[node1][1];

double y1 = edges[node1][2];

double x2 = edges[node2][1];

double y2 = edges[node2][2];

return sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2));

}

void initialize\_pheromone() {

for (int i = 0; i < num\_nodes; i++) {

for (int j = 0; j < num\_nodes; j++) {

if (i != j) {

pheromone[i][j] = 0.01;

}

}

}

}

void initialize\_heuristic() {

for (int i = 0; i < num\_nodes; i++) {

for (int j = 0; j < num\_nodes; j++) {

if (i != j) {

heuristic[i][j] = 1.0 / euclidean\_distance(i, j);

}

}

}

}

void initialize\_visited() {

for (int i = 0; i < num\_nodes; i++) {

visited[i] = 0;

}

}

int select\_next\_node(int current\_node, double\*\* probabilities) {

double r = (double)rand() / (double)RAND\_MAX;

double sum = 0.0;

for (int i = 0; i < num\_nodes; i++) {

if (!visited[i]) {

sum += probabilities[current\_node][i];

if (r <= sum) {

return i;

}

}

}

return -1;

}

void construct\_solution(int ant, int start\_node) {

path[0] = start\_node;

visited[start\_node] = 1;

for (int i = 1; i < num\_nodes; i++) {

double probabilities[MAX\_NODES];

double sum = 0.0;

for (int j = 0; j < num\_nodes; j++) {

if (!visited[j]) {

probabilities[j] = pow(pheromone[path[i-1]][j], ALPHA) \* pow(heuristic[path[i-1]][j], BETA);

sum += probabilities[j];

}

}

for (int j = 0; j < num\_nodes; j++) {

probabilities[j] /= sum;

}

int next\_node = select\_next\_node(path[i-1], &probabilities);

path[i] = next\_node;

visited[next\_node] = 1;

}

}

double calculate\_path\_length(int\* path) {

double length = 0.0;

for (int i = 0; i < num\_nodes-1; i++) {

length += euclidean\_distance(path[i], path[i+1]);

}

return length;

}