***Assignment 4***

Name: James Shepherd

Student Number: 217257968

Date: 15 November 2021

I declare that this is my own, original work.



Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*Parameter Values*

**Genetic Algorithm**

Stopping Conditions: 1000 iterations (generally not much change after this point)

Initialization: Uniform distribution [-5, 5]

Population size: 100

Selection pressure: 0.1 (standard value. changing it didn’t make much difference)

Mutation rate: 0.6 (gave the best results)

Mutation magnitude: 0.1 (gave the best results)

Crossover rate: 0.5 (standard value. changing it didn’t make much difference)

**Differential Evolution**

Stopping Conditions: 1000 iterations (generally not much change after this point)

Initialization: Uniform distribution [-5, 5]

Population size: 100

Mutation rate: 1 (gave the best results)

Crossover rate: 0.9 (gave the best results)

**Particle Swarm**

Stopping Conditions: 1000 iterations (generally no change after this point; could been less but 1000 worked well for graphing)

Initialization: Uniform distribution [-5, 5]

Population size: 150 (more individuals can cover more ground)

w: 0.1 (this produced the best results)

c1: 0.99 (to maximize exploration, gave best results)

c2: 0.01 (to minimize exploitation, gave the best results)

*Results*

**Ackley function**

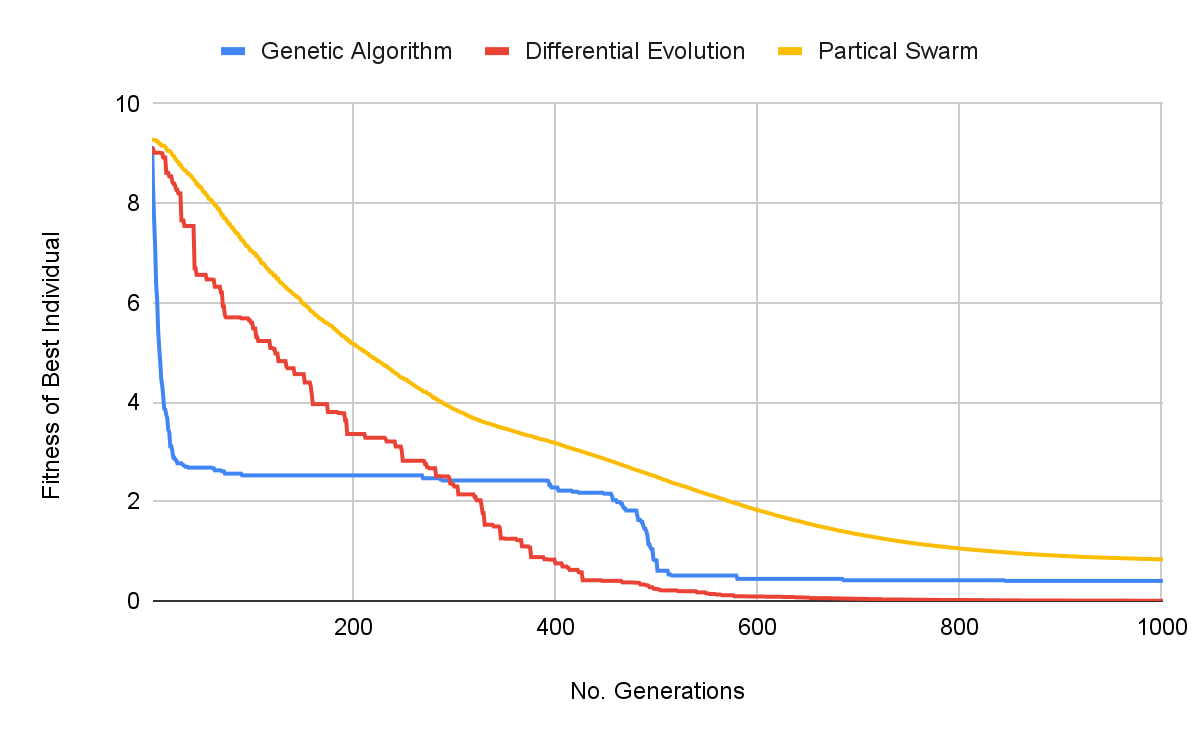


Figure 1: Fitness vs generations for various algorithms

**Griewank function**

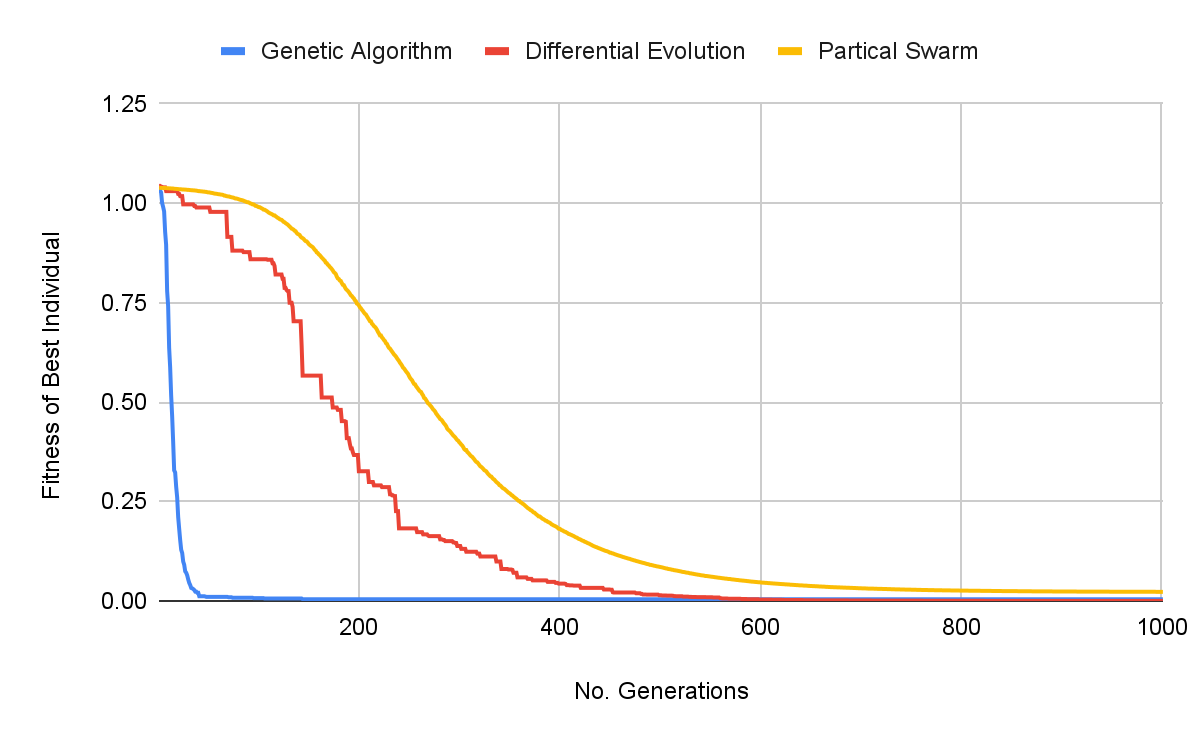


Figure 2: Fitness vs generations for various algorithms

**Rosenbrock function**

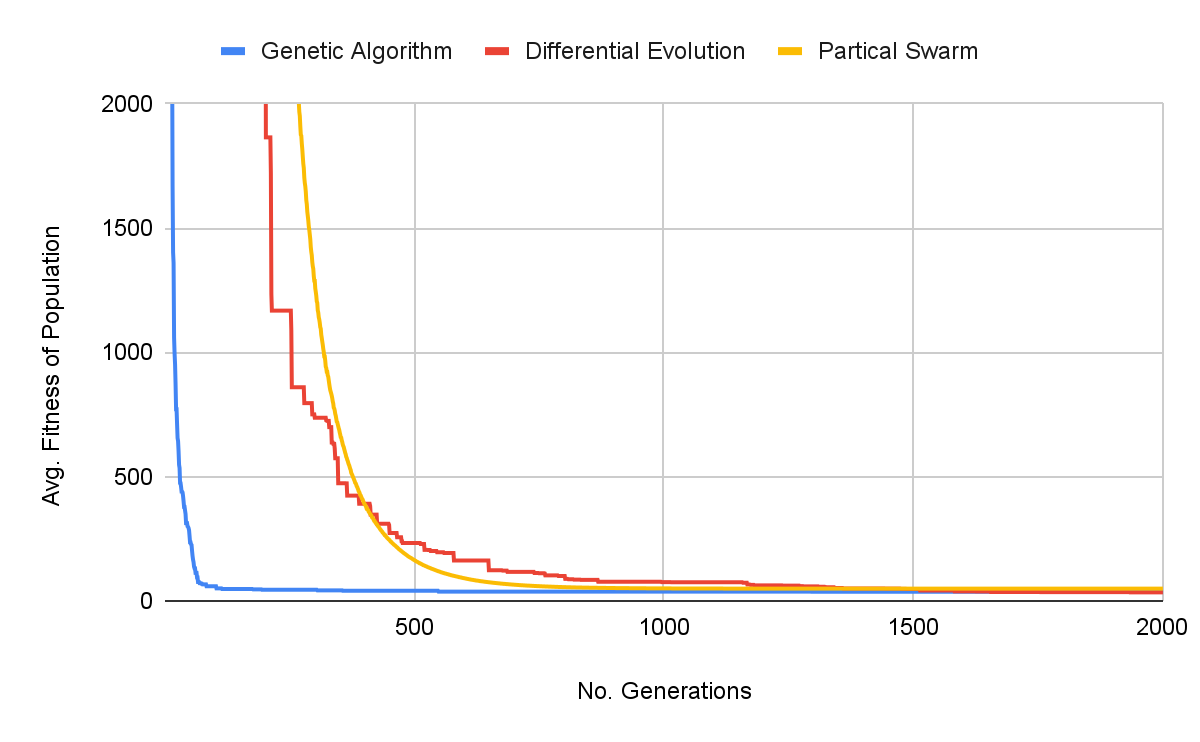
I let this one run for 2000 iterations to show that Differential Evolution does eventually produce the best results.

Figure 3: Fitness vs generations for various algorithms

**Spherical function**

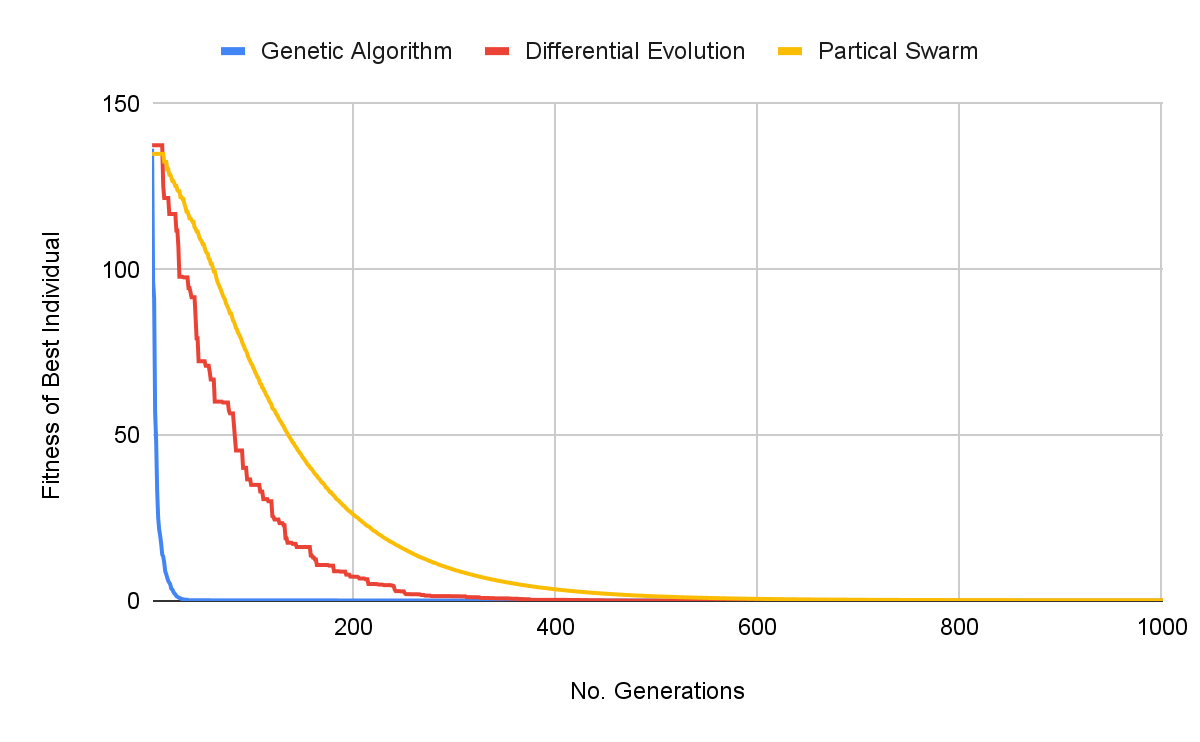


Figure 4: Fitness vs generations for various algorithms

**Whitley function**

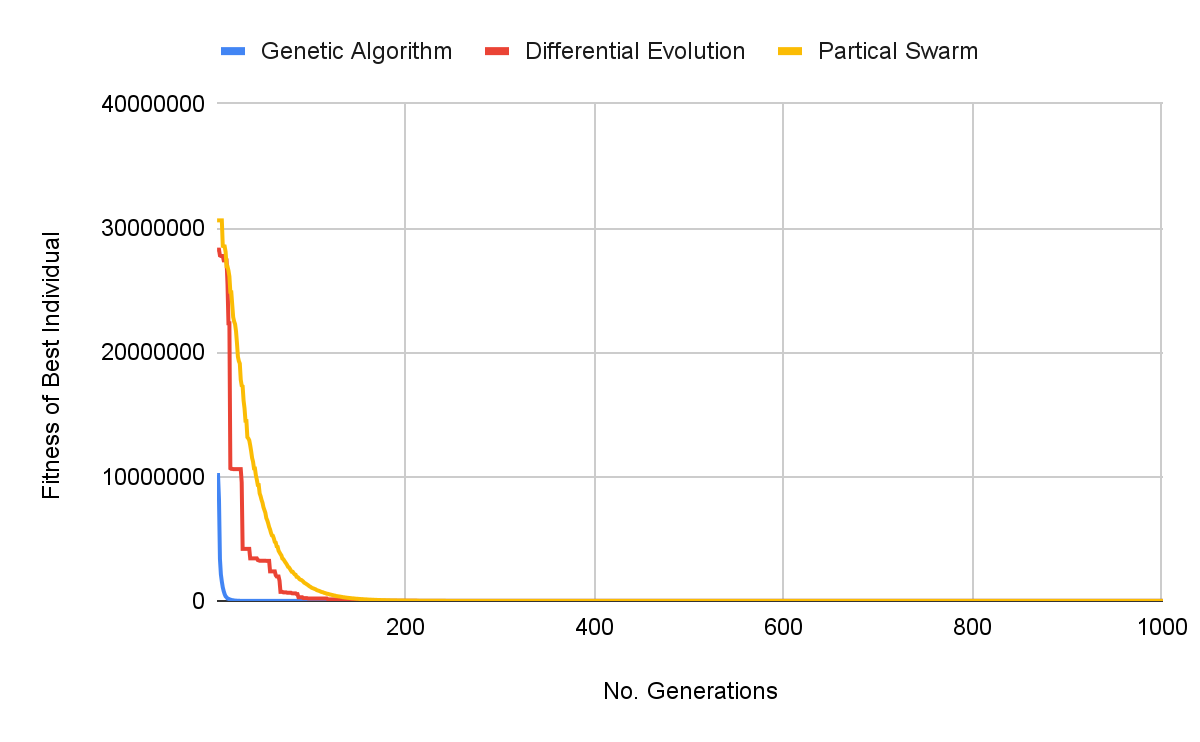


Figure 5: Fitness vs generations for various algorithms

**Overview -** best fitness

Best algorithm 2nd best algorithm Worst algorithm

|  | **Ackley** | **Griewank** | **Rosenbrock** | **Spherical** | **Whitley** |
| --- | --- | --- | --- | --- | --- |
| **Genetic Algorithm** | 0.4017857 | 0.0041806 | 36.8390390 | 0.0738962 | 560.2784718 |
| **Differential Evolution** | 3.598E-03 | 1.708E-05 | 34.2552912 | 2.671E-05 | 356.9605288 |
| **Particle Swarm** | 0.8360799 | 0.0228237 | 49.3448260 | 0.1875321 | 587.2804225 |

*Observations*

From the overview table above, we can clearly see that the DE algorithm produced the best results every time. It is also worth noting that for some functions, if left to run for more iterations, the DE algorithm actually reached 0 (or at the very least IntelliJ rounded off to 0).

The graphs for each function show that DE and PSO both converge gradually, although PSO converges slightly more gradually and generally takes the longest of the three algorithms. The table above also shows that PSO produces the worst results of the three algorithms.

GA on the other hand converges very quickly but doesn’t reach results quite as good as those reached by DE. Thus, if finding a decent estimate quickly is more important for the given situation then GA would be a better option as it has a steep gradient and reaches fairly decent estimates much faster than DE or PSO.

Overall, I would conclude that DE is the best option if you want to get the best results but it will take slightly longer than GA (especially if you are dealing with large populations and big search spaces). On the other hand, GA is the best option if you need results within fewer iterations and if you are willing to settle for decent results (not the best results). PSO may be applicable to certain problems but according to these tests it is not the best option to use.

**Main Class**

public class Main { *// Uncomment the algorithm you want to use*

public static void main(String[] args) throws IOException {

GeneticAlgorithm algorithm = new GeneticAlgorithm(1);

algorithm.run();

*/\*DifferentialEvolution algorithm = new DifferentialEvolution(1);*

*algorithm.run();\*/*

*/\*ParticleSwarm algorithm = new ParticleSwarm(1);*

*algorithm.run();\*/*

}

}

**Genetic Algorithm**

public class GeneticAlgorithm {

ContinuousFunction function;

int populationSize = 100;

double selectionPressure = 0.1;

double mutationRate = 0.6;

double mutationMagnitude = 0.1;

ArrayList<ArrayList<Double>> population;

ArrayList<Double> populationFitness;

public GeneticAlgorithm(int function) {

population = new ArrayList<>();

populationFitness = new ArrayList<>();

switch (function){

case 1 -> this.function = new Ackley();

case 2 -> this.function = new Griewank();

case 3 -> this.function = new Rosenbrock();

case 4 -> this.function = new Spherical();

case 5 -> this.function = new Whitley();

}

}

public void run() {

generatePopulation(); *// initial random population*

*// Check population fitness*

populationFitness.clear();

for(ArrayList<Double> individual : population)

populationFitness.add(checkFitness(individual));

*// Several generation*

int genCount = 0;

int fittestPos = findFittestIndividual(population, populationFitness);

while(genCount++ < 1000) {

System.*out*.println(populationFitness.get(fittestPos));

*// Create next gen*

ArrayList<ArrayList<Double>> children = new ArrayList<>();

children.add(population.get(fittestPos)); *// Elitism*

for (int i = 1; i < populationSize; i++) {

ArrayList<Double> parent1 = tournamentSelection(selectionPressure);

ArrayList<Double> parent2 = tournamentSelection(selectionPressure);

children.add(uniformCrossover(parent1, parent2));

}

population = children; *// Move to next gen*

*// Check population fitness*

populationFitness.clear();

for(ArrayList<Double> individual : population)

populationFitness.add(checkFitness(individual));

fittestPos = findFittestIndividual(population, populationFitness);

}

System.*out*.println(populationFitness.get(fittestPos));

}

*// Generate random individuals as the initial population*

private void generatePopulation() {

Random random = new Random();

double gene;

ArrayList<Double> individual;

for (int x = 0; x < populationSize; x++) {

individual = new ArrayList<>(); *// Create new individual*

*// Generate new individual's genes*

for (int y = 0; y < function.getDimension(); y++) {

do {

gene = (random.nextDouble() \* 2 - 1) \* 5; *// randomly generate new gene*

} while (gene == 0);

individual.add(gene); *// add valid gene to individual*

}

population.add(individual); *// add individual to population*

}

}

*// Calculate the fitness of an individual*

private double checkFitness(ArrayList<Double> individual) {

return function.evaluate(individual);

}

*// Select the single best parent from a number of random individuals in the population*

private ArrayList<Double> tournamentSelection(double selectionPressure) {

Random rand = new Random();

ArrayList<ArrayList<Double>> tournamentPopulation = new ArrayList<>();

ArrayList<Double> tournamentFitness = new ArrayList<>();

*// Select random tournament contestant*

int tournamentSize = (int)(populationSize\*selectionPressure);

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

int pos = rand.nextInt(populationSize);

tournamentPopulation.add(population.get(pos));

tournamentFitness.add(populationFitness.get(pos));

}

int fittestPos = findFittestIndividual(tournamentPopulation, tournamentFitness);

return tournamentPopulation.get(fittestPos);

}

*// Find the position of the best individual*

private int findFittestIndividual(ArrayList<ArrayList<Double>> population, ArrayList<Double> populationFitness) {

double minFitness = Double.*MAX\_VALUE*;

int fittestIndividual = 0;

for(int i = 0; i < population.size(); i++) {

double fitness = populationFitness.get(i);

if (fitness < minFitness) {

minFitness = fitness;

fittestIndividual = i;

}

}

return fittestIndividual;

}

*// Use uniform crossover to produce a child from 2 parents*

private ArrayList<Double> uniformCrossover(ArrayList<Double> parent1, ArrayList<Double> parent2) {

Random rand = new Random();

ArrayList<Double> child = new ArrayList<>();

for (int i = 0; i < function.getDimension(); i++){

if (rand.nextDouble() <= 0.5)

child.add(parent1.get(i) + mutation());

else

child.add(parent2.get(i) + mutation());

}

return child;

}

*// Generates a random mutation value based on the mutationRate and mutationMagnitude*

private double mutation() {

Random rand = new Random();

if (rand.nextDouble() <= mutationRate)

return rand.nextGaussian() \* mutationMagnitude;

return 0;

}

}

**Differential Evolution**

public class DifferentialEvolution {

ContinuousFunction function;

int populationSize = 100;

double mutationRate = 1;

double crossoverRate = 0.9;

ArrayList<ArrayList<Double>> population;

ArrayList<Double> populationFitness;

public DifferentialEvolution(int function) {

population = new ArrayList<>();

populationFitness = new ArrayList<>();

switch (function){

case 1 -> this.function = new Ackley();

case 2 -> this.function = new Griewank();

case 3 -> this.function = new Rosenbrock();

case 4 -> this.function = new Spherical();

case 5 -> this.function = new Whitley();

}

}

*// DE\rand\1\bin*

public void run() {

generatePopulation(); *// initial random population*

*// Check population fitness*

populationFitness.clear();

for(ArrayList<Double> individual : population)

populationFitness.add(checkFitness(individual));

*// Several generation*

Random rand = new Random();

int genCount = 0;

int fittestPos = findFittestIndividual(population, populationFitness);

while(genCount++ < 1000) {

System.*out*.println(populationFitness.get(fittestPos));

*// Create next gen*

ArrayList<ArrayList<Double>> children = new ArrayList<>();

children.add(population.get(fittestPos)); *// Elitism*

for (int i = 1; i < populationSize; i++) {

ArrayList<Double> cur = population.get(i);

*// Pick 3 random 'parents'*

ArrayList<Double> x1 = population.get(rand.nextInt(populationSize));

ArrayList<Double> x2 = population.get(rand.nextInt(populationSize));

ArrayList<Double> x3 = population.get(rand.nextInt(populationSize));

*// Create mutant vector and trial vector*

ArrayList<Double> mutant = mutantVector(x1, x2, x3);

ArrayList<Double> trialVec = uniformCrossover(cur, mutant);

double fitness = checkFitness(trialVec);

*// Compare individual and trial to choose child*

if(fitness < populationFitness.get(i))

children.add(trialVec);

else

children.add(cur);

}

population = children; *// Move to next gen*

*// Check population fitness*

populationFitness.clear();

for(ArrayList<Double> individual : population)

populationFitness.add(checkFitness(individual));

fittestPos = findFittestIndividual(population, populationFitness);

}

System.*out*.println(populationFitness.get(fittestPos));

}

*// Generate random individuals as the initial population*

private void generatePopulation() {

Random random = new Random();

double gene;

ArrayList<Double> individual;

for (int x = 0; x < populationSize; x++) {

individual = new ArrayList<>(); *// Create new individual*

*// Generate new individual's genes*

for (int y = 0; y < function.getDimension(); y++) {

do {

gene = (random.nextDouble() \* 2 - 1) \* 5; *// randomly generate new gene*

} while (gene == 0);

individual.add(gene); *// add valid gene to individual*

}

population.add(individual); *// add individual to population*

}

}

*// Generates a mutant vector based on the 3 input vectors*

private ArrayList<Double> mutantVector(ArrayList<Double> x1, ArrayList<Double> x2, ArrayList<Double> x3) {

ArrayList<Double> diff = vectorDiff(x2, x3);

ArrayList<Double> mutant = new ArrayList<>();

for(int i = 0; i < x1.size(); i++)

mutant.add(x1.get(i) + (mutationRate \* diff.get(i)));

return mutant;

}

*// Calculate the fitness of an individual*

private double checkFitness(ArrayList<Double> individual) {

return function.evaluate(individual);

}

*// Finds the position of the fittest individual*

private int findFittestIndividual(ArrayList<ArrayList<Double>> population, ArrayList<Double> populationFitness) {

double minFitness = Double.*MAX\_VALUE*;

int fittestIndividual = 0;

for(int i = 0; i < population.size(); i++) {

double fitness = populationFitness.get(i);

if (fitness < minFitness) {

minFitness = fitness;

fittestIndividual = i;

}

}

return fittestIndividual;

}

*// Use uniform crossover to produce a child from 2 parents*

private ArrayList<Double> uniformCrossover(ArrayList<Double> parent1, ArrayList<Double> parent2) {

Random rand = new Random();

ArrayList<Double> child = new ArrayList<>();

double guarantee = rand.nextInt(function.getDimension());

for (int i = 0; i < function.getDimension(); i++){

if (rand.nextDouble() <= crossoverRate || i == guarantee)

child.add(parent1.get(i));

else

child.add(parent2.get(i));

}

return child;

}

private ArrayList<Double> vectorDiff(ArrayList<Double> v1, ArrayList<Double> v2){

ArrayList<Double> diff = new ArrayList<>();

for(int i = 0; i < v1.size(); i++)

diff.add(v1.get(i) - v2.get(i));

return diff;

}

}

**Particle Swarm Optimization**

**public class ParticleSwarm {**

**ContinuousFunction function;**

**int populationSize = 150;**

**double w = 1;**

**double c1 = 0.9;**

**double c2 = 0.1;**

**ArrayList<ArrayList<Double>> population;**

**ArrayList<ArrayList<Double>> populationVelocity;**

**ArrayList<ArrayList<Double>> personalBests;**

**ArrayList<Double> globalBest;**

**ArrayList<Double> populationFitness;**

**public ParticleSwarm(int function) {**

**population = new ArrayList<>();**

**populationFitness = new ArrayList<>();**

**populationVelocity = new ArrayList<>();**

**personalBests = new ArrayList<>();**

**switch (function){**

**case 1 -> this.function = new Ackley();**

**case 2 -> this.function = new Griewank();**

**case 3 -> this.function = new Rosenbrock();**

**case 4 -> this.function = new Spherical();**

**case 5 -> this.function = new Whitley();**

**}**

**}**

**public void run() {**

**generatePopulation(); *// initial random population***

***// Check population fitness***

**populationFitness.clear();**

**for(ArrayList<Double> individual : population)**

**populationFitness.add(checkFitness(individual));**

***// Initialize global best***

**int fittestPos = findFittestIndividual(population,populationFitness);**

**globalBest = population.get(fittestPos);**

***// Start swarming***

**int genCount = 0;**

**while(genCount++ < 1000) {**

**System.*out*.println(populationFitness.get(fittestPos));**

***// Update population positions***

**for (int i = 1; i < populationSize; i++) {**

**ArrayList<Double> x = population.get(i); *// current individual***

**ArrayList<Double> v = populationVelocity.get(i); *// current individual's velocity***

**ArrayList<Double> y = personalBests.get(i); *// current individual's personal best***

***// Update personal best***

**if (checkFitness(x) < checkFitness(personalBests.get(i)))**

**personalBests.set(i, x);**

***// Update global best***

**if (checkFitness(personalBests.get(i)) < populationFitness.get(fittestPos)) {**

**globalBest = personalBests.get(i);**

**fittestPos = i;**

**}**

***// Move individual one dimension at a time***

**Random rand = new Random();**

**for (int j = 0; j < function.getDimension(); j++){**

**double r1 = rand.nextDouble(); *// randomizer 1***

**double r2 = rand.nextDouble(); *// randomizer 2***

***// Calculate new velocity dimension and update individual***

**double newVelocity = (w \* v.get(j)) + (c1 \* r1 \* (y.get(j) - x.get(j))) + (c2 \* r2 \* (globalBest.get(j) - x.get(j)));**

**v.set(j, newVelocity);**

**x.set(j, x.get(j) + newVelocity);**

**}**

***// Check individual's fitness***

**populationFitness.set(i, checkFitness(x));**

**}**

**}**

**System.*out*.println(populationFitness.get(fittestPos));**

**}**

***// Generate random individuals as the initial population***

**private void generatePopulation() {**

**Random random = new Random();**

**double gene;**

**ArrayList<Double> individual;**

**ArrayList<Double> velocity;**

**for (int x = 0; x < populationSize; x++) {**

**individual = new ArrayList<>(); *// Create new individual***

**velocity = new ArrayList<>();**

***// Generate new individual's genes***

**for (int y = 0; y < function.getDimension(); y++) {**

**do {**

**gene = (random.nextDouble() \* 2 - 1) \* 5; *// randomly generate new gene***

**} while (gene == 0);**

**individual.add(gene); *// add valid gene to individual***

**velocity.add(0.0); *// initialize velocity***

**}**

**population.add(individual); *// add individual to population***

**populationVelocity.add(velocity);**

**}**

**personalBests = (ArrayList<ArrayList<Double>>) population.clone(); *// initialize personal bests***

**}**

***// Calculate the fitness of an individual***

**private double checkFitness(ArrayList<Double> individual) {**

**return function.evaluate(individual);**

**}**

***// Finds the position of the fittest individual***

**private int findFittestIndividual(ArrayList<ArrayList<Double>> population, ArrayList<Double> populationFitness) {**

**double minFitness = Double.*MAX\_VALUE*;**

**int fittestIndividual = 0;**

**for(int i = 0; i < population.size(); i++) {**

**double fitness = populationFitness.get(i);**

**if (fitness < minFitness) {**

**minFitness = fitness;**

**fittestIndividual = i;**

**}**

**}**

**return fittestIndividual;**

**}**

**}**