Inteligência Artificial
Relatório TP1-P2

Turno Prático: P2
Professora: Susana Nascimento

Henrique Garcês, nº42077
João Diogo Rodrigues, nº41762
Luís Sousa, nº41848

Classes

ObservationData

-Esta classe caracteriza o mapa dos pontos a visitar pelo *Rover*. É responsável pela codificação dos dados do problema do *Rover*, nomeadamente: O intervalo de observação (*ObservationInterval*), os pontos de observação (*ObservationSpots*), o custo de um *observation spot* para o seguinte e o número de *spots* necessários visitar pelo *Rover*.

Individual

-Esta classe abstracta tem como objectivo representar um individuo da população, isto é, um cromossoma. Aqui são definidos os métodos a implementar para qualquer individuo que se faça representar no algoritmo genético, nomeadamente: função de fitness, cruzamento, mutação e clonagem.

RoverCircuit

-Esta classe instacia a classe abstracta Individual, tornando um *RoverCircuit* (circuito marciano) num elemento genético com a possibilidade de cruzamentos com outros circuitos, sofrer mutações e clonagem.

Population

-Esta classe representa o conjunto de entidades genéticas, neste caso, circuitos marcianos.
São definidos, aqui, os métodos que permitem seleccionar um individuo da população de acordo com o seu valor de função de fitness, adicionar um individuo à população, obter o melhor e pior individuo da população, o melhor e pior valor de fitness, e o registo permanente da elite (os melhores individuos).

GeneticAlgorithm

-Esta classe implementa o algoritmo genético.
São definidos, aqui, o método de pesquisa (*search()*) que pesquisa e devolve o melhor individuo encontrado. O método *search()* implementa três critérios de paragem sendo eles: *Número de gerações*, *limite de tempo* e *limite para o valor da função de fitness*; implementa dois métodos de selecção: *Roleta* e *Elitismo*; e implementa a possibilidade de usar a estratégia de *hill-climbing*.

CircuitTest

-Esta classe executa o algoritmo genético implementado em *GeneticAlgorithm*.
Os valores de inicialização são todos eles passados pelo utilizador, com a possibilidade de utilizar valores por defeito para *probabilidade de cruzamento* e *mutação*, t*amanho da população* e da *elite*. O utilizador tem, também, a possibilidade de escolher a utilização ou não da estratégia de *hill-climbing*.

Operações

Inicialização: É feita com uma população definida pelo utilizador ou população aleatória.

Selecção: É feita através do método da *Roleta* ou *Elitismo*. O método de selcção, também, é escolhida pelo utilizador.

Reprodução: Utilizamos para reprodução os métodos de cruzamento *OX1* e *OX2*, e os métodos de mutação por *Troca* e por *Inversão* do intervalo de corte do cromossoma.

Actualização: Os individuos resultantes da reprodução são inseridos na população consoante o critério de selecção.

Finalização: Aos individuos de cada geração resultante, é aplicada a estratégia de *hill-climbing* caso o utilizador assim o ordene.

Desempenho

Resultados experimentais

Foram utilizados como critério de paragem o número de gerações = 50 e tamanho de elite por defeito.

Dimensão População	Método Selecção	Crossover	Probabilidade Crossover	Mutação	Probabilidade Mutação	Hill Climbing	Melhor Fitness	Tempo (s)
20	Roleta	OX1	Default	Troca	Default	Não	2716087.0	0.0447
20	Roleta	OX2	Default	Troca	Default	Não	2486162.0	0.0490 3
100	Roleta	OX1	Default	Troca	Default	Sim	2583146.0	0.2374 5
100	Roleta	OX2	Default	Troca	Default	Sim	2673146.0	0.2150 3
20	Elitismo	OX1	Default	Troca	Default	Não	2607899.0	0.0639 0
20	Elitismo	OX2	Default	Troca	Default	Não	2669992.0	0.0615 3
100	Elitismo	OX1	Default	Troca	Default	Sim	2485982.0	0.2800
100	Elitismo	OX2	Default	Troca	Default	Sim	2445215.0	0.2863
20	Roleta	OX1	0.5	Troca	0.001	Não	2223365.0	0.0437 1
20	Roleta	OX2	0.5	Troca	0.001	Não	2441891.0	0.0441 7
100	Roleta	OX1	0.5	Troca	0.001	Sim	248007	0.1972

							6.0	4
100	Roleta	OX2	0.5	Troca	0.001	Sim	2392099.0	0.20636
20	Elitismo	OX1	0.5	Troca	0.001	Não	2677292.0	0.05713
20	Elitismo	OX2	0.5	Troca	0.001	Não	2634094.0	0.06806
100	Elitismo	OX1	0.5	Troca	0.001	Sim	2492507.0	0.2564
100	Elitismo	OX2	0.5	Troca	0.001	Sim	2335867.0	.29021
20	Roleta	OX1	Default	Inversão	Default	Não	2451165.0	0.04534
20	Roleta	OX2	Default	Inversão	Default	Não	2648464.0	0.04449
100	Roleta	OX1	Default	Inversão	Default	Sim	2421363.0	0.18938
100	Roleta	OX2	Default	Inversão	Default	Sim	2516502.0	0.19001
20	Elitismo	OX1	Default	Inversão	Default	Não	2564344.0	0.05468
20	Elitismo	OX2	Default	Inversão	Default	Não	2634895.0	0.05773
100	Elitismo	OX1	Default	Inversão	Default	Sim	2275690.0	0.24005
100	Elitismo	OX2	Default	Inversão	Default	Sim	2424791.0	0.24613
20	Roleta	OX1	0.5	Inversão	0.001	Não	2491633.0	0.04206
20	Roleta	OX2	0.5	Inversão	0.001	Não	2711736.0	0.04024

100	Roleta	OX1	0.5	Inversão	0.001	Sim	2300604.0	0.175952
100	Roleta	OX2	0.5	Inversão	0.001	Sim	2482738.0	0.177434
20	Elitismo	OX1	0.5	Inversão	0.001	Não	2617356.0	0.05431
20	Elitismo	OX2	0.5	Inversão	0.001	Não	2418497.0	0.053662
100	Elitismo	OX1	0.5	Inversão	0.001	Sim	2387118.0	0.229588
100	Elitismo	OX2	0.5	Inversão	0.001	Sim	2372620.0	0.23422

Comparações

Função de mutação = *Troca*
Dimensão da população = *20*
Com *hill-climbing* vs Sem *hill-climbing*

Sem hill-climbing:
- **-Melhor fitness =** 2716087.0
- **-Tempo =** 0.044774294 s

Com hill-climbing:
- **-Melhor fitness =** 2519735.0
- **-Tempo =** 0.115519056 s

Função de mutação = *Troca*
Dimensão da população = *100*
Com *hill-climbing* vs Sem *hill-climbing*

Sem hill-climbing:
- **-Melhor fitness =** 2551945.0
- **-Tempo =** 0.150093659 s

Com hill-climbing:
- **Melhor fitness =** 2335867.0
- **Tempo =** 0.290219963 s

Comparações

Função de mutação = *Inversão*
Dimensão da população = *20*
Com hill-climbing vs Sem hill-climbing

Sem *hill-climbing*:
-Melhor fitness = 2711736.0
-Tempo = 0.040247137 s
Com *hill-climbing*:
-Melhor fitness = 2224611.0
-Tempo = 0.097722472 s

Função de mutação = *Inversão*
Dimensão da população = *100*
Com hill-climbing vs Sem hill-climbing

Sem *hill-climbing*:
-Melhor fitness = 2398757.0
-Tempo = 0.146210068 s

Com *hill-climbing*:
-Melhor fitness = 2275690.0
-Tempo = 0.24005664 s

Comparando os resultados obtidos entre si, observamos que independentemente da função de mutação, população e com a aplicação da estratégia de hill-climbing obtemos melhores individuos (menor valor de função de fitness) nas gerações seguintes.

Código Fonte

Population

```
package circuit;

import java.util.ArrayList;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Random;
import java.util.Collections;

/**
 *     Classe usada para a representação de uma população.
 */
public class Population {

    private final static int CAP = 100;

    ArrayList<Individual> pop;
    ArrayList<Double> acum;

    private static Random gen = new Random();
    private double sumOfFitness;
    private int size;
    private boolean currupt;
    private Individual bestInd;
    private Individual worstInd;
    private double bestFit;
    private double worstFit;


    /**
     *     Construtor relativo à classe Population
```

```
     */
    public Population() {
            this.size = 0;
            this.pop = new ArrayList<Individual>(CAP);
            this.acum = new ArrayList<Double>(CAP);
            this.sumOfFitness = 0.0;
            this.currupt = true;
            this.bestInd = null;
            this.worstInd = null;
            this.bestFit = Double.POSITIVE_INFINITY;
            this.worstFit = Double.NEGATIVE_INFINITY;

    }

    /**
     *     Construtor onde se especifica a popolação
     * @param p um array de indivíduos
     */
    public Population(Individual[] p) {
            this();
            for(Individual i : p)
                    addIndividual(i);
    }

    /**
     * Selecciona e devolve um indivíduo da população, tendo em conta a sua fitness
     * @return um array de indivíduos
     */
    public Individual selectIndividual() {

            // Verifica se necessita de calcular os valores de probabilidade de selecção de
cada indivíduo
            if( currupt ) {
                    double total=0.0;
                    for(int i=0; i < pop.size(); i++) {
                            total += 1/pop.get(i).fitness();
                            acum.add(total/sumOfFitness);
                    }
                    currupt = false;
```

```
        }

        double r = gen.nextDouble();
        Collections.sort(acum);
        int pos = Collections.binarySearch(acum, r);

        if( pos >= 0)
                return pop.get(pos);
        else if (-(pos + 1) >= size)
                return pop.get(size - 1);
        else
                return pop.get(-(pos+1));

}
/**
 * Adiciona um indivíduo à população
 * @param ind, um indivíduo
 */
public void addIndividual(Individual ind) {
        size++;
        pop.add(ind);
        double f = ind.fitness();
        sumOfFitness += 1/f;
        if( f > worstFit ) {
                worstFit = f;
                worstInd = ind;
        }
        if( f < bestFit ) {
                bestFit = f;
                bestInd = ind;
        }

}

public Individual getBestIndividual() {
        return this.bestInd;
}

public Individual getWorstIndividual() {
```

```
        return this.worstInd;
    }

    public double getBestFitness() {
        return this.bestFit;
    }

    public double getWorstFitness() {
        return this.worstFit;
    }

    public Population getElite(int n) {
        //ArrayList<Individual> aux = (ArrayList<Individual>) pop.clone();
        //Collections.sort(aux, new FitnessComparator());
        Queue<Individual> pqueue = new PriorityQueue<Individual>(10,new
FitnessComparator());
        pqueue.addAll(pop);

        if(n > size)
            n = size;
        Individual[] eliteArray = new Individual[size];
        pqueue.toArray(eliteArray);
        Individual[] e = new Individual[n];
        for(int i = 0; i < e.length; i++) {
            e[i] = eliteArray[i];
        }
        Population elite = new Population(e);
        return elite;
    }

    public double getAvgFitness() {
        return sumOfFitness/size;
    }

    public int getSize() {
        return this.size;
    }

    public Individual[] getEliteArray(int n) {
```

```
        Queue<Individual> pqueue = new PriorityQueue<Individual>(10,new
FitnessComparator());
        pqueue.addAll(pop);

        if(n > size)
                n = size;
        Individual[] eliteArray = new Individual[size];
        pqueue.toArray(eliteArray);
        return eliteArray;
    }

    public Individual getFromElite(int dimElite)
    {
        int n = gen.nextInt(dimElite);
        Population elite = getElite(n); // os 10 melhores da elite, por exemplo
        int genNum = gen.nextInt(elite.size-1);
        return elite.pop.get(genNum);

    }
}
```

Código Fonte

RoverCircuit

```
package circuit;

import java.util.*;

/**
 *     Classe que instancia a classe abstracta Individual
 */
public class RoverCircuit extends Individual {

    public static final int OX1 = 1;
    public static final int OX2 = 2;
    public static final int SWAP_MUT = 1;
    public static final int INVERSION_MUT = 2;

    private int crossoverType;
    private int mutationType;

    private static Random gen = new Random();
    private static Individual[] children = new Individual[2];
    private int size;
    private int[] circuit;
    private ObservationData data;
    private double fitness;    //tempo corrente


    public RoverCircuit(ObservationData data, int crossType, int mutatType) {
        this.data = data;
        this.size = data.getSize();
        this.circuit = new int[size];
```

```
        this.fitness = 0.0;
        this.crossoverType = crossType;
        this.mutationType = mutatType;

        List<Integer> listInt = new ArrayList<Integer>();
        for(int i = 0; i < size; i++)
                listInt.add(i);

        Collections.shuffle(listInt);
        for(int i = 0; i < size; i++)
                circuit[i] = listInt.get(i);
    }

    public RoverCircuit(ObservationData data, int[] circuit, int crossType, int mutatType) {
        this.data = data;
        this.size = data.getSize();
        this.circuit = circuit;
        this.fitness = 0.0;
        this.crossoverType = crossType;
        this.mutationType = mutatType;
    }

    @Override
    public double fitness() {
         int fit = data.getSpot(circuit[0]).firstTime();
        fit += data.getSpot(circuit[0]).durationObservation(fit);
        for(int i=1; i < size; i++ ) {
                fit += data.getSpot(circuit[i]).durationObservation(fit) +
data.getCost(circuit[i-1],circuit[i]);
        }
        fit += data.getCost(circuit[size-1], circuit[0]); //adiciona o custo do ultimo para o
primeiro

        fitness = (double) fit;
         return fitness;
    }

    @Override
    public Individual[] crossover(Individual other) {
```

```
        if(crossoverType == OX1)
                return crossOX1(other);
        else
                return crossOX2(other);
}

private Individual[] crossOX1(Individual other)
{
        RoverCircuit mother = (RoverCircuit) other;
        int r1 = gen.nextInt(size-1);
        int r2 = gen.nextInt(size-2);
        int cut1, cut2;

        if(r2 >= r1) {
                cut1 = r1 + 1;
                cut2 = r2 + 2;
        }
        else {
                cut1 = r2 + 1;
                cut2 = r1 + 2;
        }
        List<Integer> tmp1 = new ArrayList<Integer>();
        List<Integer> tmp2 = new ArrayList<Integer>();
        int[] c1 = new int[this.size];    //filho1
        int[] c2 = new int[this.size];    //filho2
        Arrays.fill(c1, -1);
        Arrays.fill(c2, -1);

        for(int i = cut1; i < cut2; i++) {
                c1[i] = circuit[i];
                tmp1.add(circuit[i]);

                c2[i] = mother.circuit[i];
                tmp2.add(mother.circuit[i]);
        }


        int aux1, aux2;
        aux1 = cut2;
```

```
aux2 = cut2;

while(tmp1.size() < size) //copia valores do 2ยบ progenitor, partindo de cut2
{
	if(aux2 == size) {
		aux2 = 0;
		aux1 = 0;
	}

	while(c1[aux1] == -1)
	{
		if(!tmp1.contains(mother.circuit[aux2]))
		{
			tmp1.add(mother.circuit[aux2]);
			c1[aux1] = mother.circuit[aux2];
			aux2++;
			aux1++;
			if(aux1 == size)
				aux1 = 0;
			if(aux2 == size)
				aux2 = 0;
		} else {
			aux2++;
			if(aux2 == mother.circuit.length)
				aux2 = 0;
		}
	}
}

aux1 = cut2;
aux2 = cut2;
while(tmp2.size() < size) //copia valores do 1ยบ progenitor, partindo de cut2
{
	if(aux2 == size) {
		aux2 = 0;
		aux1 = 0;
	}
	while(c2[aux1] == -1)
	{
```

```
                    if(!tmp2.contains(circuit[aux2]))
                    {
                            tmp2.add(circuit[aux2]);
                            c2[aux1] = circuit[aux2];
                            aux2++;
                            aux1++;
                            if(aux1 == size)
                                    aux1 = 0;
                            if(aux2 == size)
                                    aux2 = 0;

                    } else {
                            aux2++;
                            if(aux2 == size)
                                    aux2 = 0;
                    }

            }
        }

        children[0] = new RoverCircuit(data, c1, crossoverType, mutationType);
        children[1] = new RoverCircuit(data, c2, crossoverType, mutationType);

        return children;
}

private Individual[] crossOX2(Individual other) {
        RoverCircuit mother = (RoverCircuit) other;
        int r1 = gen.nextInt(size-1);
        int r2 = gen.nextInt(size-2);
        int cut1, cut2;

        //atribui os valores aleatorios ao cut1 e cut2
        if(r2 >= r1) {
                cut1 = r1 + 1;
                cut2 = r2 + 2;
        }
        else {
                cut1 = r2 + 1;
```

```
            cut2 = r1 + 2;
        }

        int[] c1 = new int[this.size];    //filho1
        int[] c2 = new int[this.size];    //filho2
        List<Integer> tmp1 = new ArrayList<Integer>();     //listas temporarias para
verificar se um filho ja contem
        List<Integer> tmp2 = new ArrayList<Integer>();    //um determinado numero

        Arrays.fill(c1, -1);
        Arrays.fill(c2, -1);

        for(int i = cut1; i < cut2; i++) {
            c1[i] = circuit[i];
            tmp1.add(circuit[i]);

            c2[i] = mother.circuit[i];
            tmp2.add(mother.circuit[i]);
        }

        int m = 0, n = 0;
        boolean done = false;
        //cria filho1
        while(!done) {

            if(c1[n] == -1) {
                if(!tmp1.contains(mother.circuit[m])) {
                    c1[n++] = mother.circuit[m];
                    tmp1.add(mother.circuit[m]);
                    m++;

                    if(m == size || n == size)
                        done = true;
                }
                else {
                    m++;
                    if(m == size )
                        done = true;
                }
```

```
            }

            else {
                n++;
                if(n == size )
                    done = true;
            }
        }

        //cria filho2
        m = 0; n = 0; done = false;
        while(!done) {
            if(c2[n] == -1) {
                if(!tmp2.contains(circuit[m])) {
                    c2[n] = circuit[m];
                    tmp2.add(circuit[m]);
                    m++;
                    if(m == size || n == size)
                        done = true;
                }
                else {
                    m++;
                    if(m == size )
                        done = true;
                }
            }
            else {
                n++;
                if(n == size )
                    done = true;
            }
        }

        children[0] = new RoverCircuit(data, c1, crossoverType, mutationType);
        children[1] = new RoverCircuit(data, c2, crossoverType, mutationType);

        return children;

    }
```

```
    @Override
    //Random swap
    public void mutate() {
        if(mutationType == SWAP_MUT)
            swap();
        else
            inversion();
    }

    private void swap() {
        int r1 = gen.nextInt(size-1);
        int r2 = gen.nextInt(size-1);
        if(r2 >= r1)
            r2++;
        int aux = circuit[r1];
        circuit[r1] = circuit[r2];
        circuit[r2] = aux;
    }

    private void inversion() {
        int r1 = gen.nextInt(size-1);
        int r2 = gen.nextInt(size-2);
        if(r2 >= r1)
            r2++;
        int cut1 = Math.min(r1, r2);
        int cut2 = Math.max(r1, r2);
        for(int i = cut1; i < cut2; i++) {
            int tmp = circuit[i];
            circuit[i] = circuit[cut2];
            circuit[cut2--] = tmp;
        }
    }

//    public void mutateByInsertion() {
//        int r1 = gen.nextInt(size-1);
//        int r2 = gen.nextInt(size-1);
//        if(r2 == r1)
//            r2++;
```

```
//          int aux = circuit[r2];
//          circuit[r1] = aux;
//    }

    @Override
    public Object clone() {
          return new RoverCircuit(data, circuit.clone(), crossoverType, mutationType);
    }

    @Override
    public String toString() {
          return Arrays.toString(circuit);
    }
}
```

Código Fonte

GeneticAlgorithm

```
package circuit;

import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Random;

import org.jfree.data.xy.XYSeries;
/**
 * Classe que "implementa" o algoritmo genetico
 */
public class GeneticAlgorithm {

    public static final float DEFAULT_PROB_CROSSOVER = 0.9f;
    public static final float DEFAULT_PROB_MUTATION = 0.01f;
    public static final int DEFAULT_ELITE_SIZE = 5;
    public static final int DEFAULT_GENERATION_SIZE = 100;
    public static final int NUM_GENERATION = 1;
    public static final int TIME_LIMIT = 2;
    public static final int FITNESS_BOUND = 3;
    public static final int ROULETTE_WHEEL_SELECTION = 1;
    public static final int ELITISM_SELECTION = 2;

    private float pc =  DEFAULT_PROB_CROSSOVER;
    private float pm = DEFAULT_PROB_MUTATION;
    private Population pop;
    private int elite = DEFAULT_ELITE_SIZE;
    private int generations = DEFAULT_GENERATION_SIZE;
    private int stopCriteria = NUM_GENERATION;
    private int selectCriteria = ELITISM_SELECTION;
```

```
public int hillClimbingOption;
private double elapseTime;
private double fitnessBound;
private double bestSolution;
private double worstSolution;
private int generationOfBest;
private int generationOfWorst;
private int nTotalOfGenerations;
private Individual bestIndividual;
private Individual worstIndividual;
private long startTime;
private long endTime;
private XYSeries series;

/**
 * Construtor
 * @param pop uma popula��o
 */
GeneticAlgorithm(Population pop) {
	this.pop = pop;
	this.bestIndividual = null;
	this.worstIndividual = null;
	this.bestSolution = 0.0;
	this.worstSolution = 0.0;
	this.generationOfBest = 1;
	this.generationOfWorst = 1;
	this.nTotalOfGenerations = 0;
}
/**
 * Construtor
 * @param pop uma popula��o
 * @param pcrossover a probabilidade de crossover
 * @param pmutate a probabilidade de muta��o
 */
GeneticAlgorithm(Population pop, float pcrossover, float pmutate) {
	this(pop);
	this.pc = pcrossover;
	this.pm = pmutate;
}
```

```
/**
 *     M�todo que pesquisa e devolve o melhor indiv�duo encontrado
 * @return pop.getBestIndividual(), o melhor indiv�duo
 */

public Individual search() {
	Random gen = new Random();
	startTime = System.nanoTime();
	int count_gen = 1;
	Individual[] children = new Individual[2];
	Individual x,y;
	bestSolution = pop.getBestIndividual().fitness();
	worstSolution = pop.getWorstIndividual().fitness();
	bestIndividual = pop.getBestIndividual();
	worstIndividual = pop.getWorstIndividual();
	Population newpop;
	if(stopCriteria == NUM_GENERATION) {
		while(count_gen <= generations) {
			series.add(count_gen, pop.getBestIndividual().fitness());
			if(selectCriteria == ELITISM_SELECTION)
				newpop = pop.getElite(elite);
			else
				newpop = new Population();
			while(newpop.getSize() < pop.getSize()) {
				//selecionar individuo
				//gerar numeros reais para as probabilidades

				x = pop.selectIndividual();
				y = pop.selectIndividual();

				if(gen.nextFloat() < pc) {
					children = x.crossover(y);
				}
				else {
					children[0] = x;
					children[1] = y;
				}
```

```
                        if(gen.nextFloat() < pm)
                                children[0].mutate();
                        if(gen.nextFloat() < pm)
                                children[1].mutate();

                        newpop.addIndividual(children[0]);
                        newpop.addIndividual(children[1]);
                }
                pop = newpop;

                if(pop.getWorstIndividual().fitness() > worstSolution) {
                        worstIndividual = pop.getWorstIndividual();
                        worstSolution = worstIndividual.fitness();
                        generationOfWorst = count_gen+1;
                }
                if(pop.getBestIndividual().fitness() < bestSolution) {
                        bestIndividual = pop.getBestIndividual();
                        bestSolution = bestIndividual.fitness();
                        generationOfBest = count_gen+1;

                        if(generationOfBest > generations)
                                generationOfBest--;
                }

                if(hillClimbingOption==1)
                        pop.addIndividual(hillClimbing(count_gen));
                count_gen++;
        }
}

else if(stopCriteria == 2) {
        while((System.nanoTime() - startTime)/1E9 < elapseTime) {
                series.add(count_gen, pop.getBestIndividual().fitness());
                if(selectCriteria == ELITISM_SELECTION)
                        newpop = pop.getElite(elite);
                else
                        newpop = new Population();
                while(newpop.getSize() < pop.getSize()) {
```

```
                x = pop.selectIndividual();
                y = pop.selectIndividual();

                if(gen.nextFloat() < pc) {
                        children = x.crossover(y);
                }
                else {
                        children[0] = x;
                        children[1] = y;
                }

                if(gen.nextFloat() < pm)
                        children[0].mutate();
                if(gen.nextFloat() < pm)
                        children[1].mutate();

                newpop.addIndividual(children[0]);
                newpop.addIndividual(children[1]);
        }
        pop = newpop;

        if(pop.getWorstIndividual().fitness() > worstSolution) {
                worstIndividual = pop.getWorstIndividual();
                worstSolution = worstIndividual.fitness();
                generationOfWorst = count_gen;
        }

        if(pop.getBestIndividual().fitness() < bestSolution) {
                bestIndividual = pop.getBestIndividual();
                bestSolution = bestIndividual.fitness();
                generationOfBest = count_gen;

                if(generationOfBest > generations)
                        generationOfBest--;
        }
        if(hillClimbingOption==1)
                pop.addIndividual(hillClimbing(count_gen));
        count_gen++;
}
```

```
}
else if(stopCriteria == 3)
{

        while(pop.getBestIndividual().fitness() >= fitnessBound)
        {
                series.add(count_gen, pop.getBestIndividual().fitness());
                if(selectCriteria == ELITISM_SELECTION)
                        newpop = pop.getElite(elite);
                else
                        newpop = new Population();
                while(newpop.getSize() < pop.getSize()) {


                        x = pop.selectIndividual();
                        y = pop.selectIndividual();

                        if(gen.nextFloat() < pc) {
                                children = x.crossover(y);
                        }
                        else {
                                children[0] = x;
                                children[1] = y;
                        }

                        if(gen.nextFloat() < pm)
                                children[0].mutate();
                        if(gen.nextFloat() < pm)
                                children[1].mutate();

                        newpop.addIndividual(children[0]);
                        newpop.addIndividual(children[1]);
                }
                pop = newpop;

                if(pop.getWorstIndividual().fitness() > worstSolution) {
                        worstIndividual = pop.getWorstIndividual();
                        worstSolution = worstIndividual.fitness();
                        generationOfWorst = count_gen;
```

```
                    }

                    if(pop.getBestIndividual().fitness() < bestSolution) {
                        bestIndividual = pop.getBestIndividual();
                        bestSolution = bestIndividual.fitness();
                        generationOfBest = count_gen;

                        if(generationOfBest > generations)
                            generationOfBest--;
                    }
                    if(hillClimbingOption==1)
                        pop.addIndividual(hillClimbing(count_gen));
                    count_gen++;
                }
            }
            nTotalOfGenerations = count_gen-1;
            endTime = System.nanoTime() - startTime;

            return pop.getBestIndividual();

        }


        public Individual getWorstIndividual() {
            return worstIndividual;
        }

        public Map<String,Number> getResults() {
            Map<String,Number> results = new LinkedHashMap<String,Number>();
            results.put("Fitness of the best", bestSolution);
            results.put("Generation of the best", generationOfBest);
            results.put("\nFitness of the worst", worstSolution);
            results.put("Generation of the worst", generationOfWorst);
            results.put("\nNum total of generations", nTotalOfGenerations);
            results.put("Time", endTime/1E9);

            return results;
        }
```

```
public void setStopCriteria(int n, double value) {

	if(n == NUM_GENERATION) {
		this.stopCriteria = NUM_GENERATION;
		this.generations = (int) value;
	}
	else if(n == TIME_LIMIT) {
		this.stopCriteria = TIME_LIMIT;
		this.elapseTime =  value;
	}
	else if(n == FITNESS_BOUND) {
		this.stopCriteria = FITNESS_BOUND;
		this.fitnessBound = value;
	}
}

public void setSelectCriteria(int n, int eliteSize) {
	if(n == ROULETTE_WHEEL_SELECTION)
		selectCriteria = ROULETTE_WHEEL_SELECTION;
	else if(n == ELITISM_SELECTION)
		selectCriteria = ELITISM_SELECTION;
	this.elite = eliteSize;
}

public Individual hillClimbing(int gen) {
	Individual[] elite = pop.getEliteArray(this.elite);
	int i = 0;
	boolean done = false;
	while(!done) {
		Individual ind = elite[i];
		ind.mutate();

		if(i == elite.length-1) {
			if(ind.fitness() < elite[0].fitness()) {
				//series.add(gen, ind.fitness());
				return ind;
			}
			else
				i = 0;
```

```
                }
                else if(ind.fitness() < elite[++i].fitness()) {
                        //series.add(gen, ind.fitness());
                        return ind;
                }

        }
        return null;
    }

    public void setDataSet(XYSeries s) {
        this.series = s;
    }
}
```

Código Fonte

CircuitTest

```
package circuit;

import java.util.*;
import java.awt.BorderLayout;
import java.io.*;
import java.nio.file.Path;
import java.nio.file.Paths;

import javax.swing.JFrame;
import javax.swing.JTextArea;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;

public class CircuitTest {

    private static JFrame frame;
    private static XYSeries series;
    private static JTextArea output;

    private static int crossoverType = RoverCircuit.OX2;
    private static int mutationType = RoverCircuit.SWAP_MUT;
    private static int select;
    private static int dimElite;
    private static int hillOption;
```

```
   private static float probCrossover =
GeneticAlgorithm.DEFAULT_PROB_CROSSOVER;
   private static float probMutation = GeneticAlgorithm.DEFAULT_PROB_MUTATION;


   /**
    * @param args
    */
   public static void main(String[] args) throws IOException, ClassNotFoundException {
         System.out.println("NOME FICHEIRO.txt: \n");
         Scanner in = new Scanner(System.in);
         String command = "";
         command = in.next();

         String datastr = "";
         BufferedReader reader = new BufferedReader( new
FileReader(readFile(command)));

         String line = reader.readLine();
         while( line != null) {
               datastr += line + "\n";
               line = reader.readLine();
         }
         reader.close();
         ObservationData r = new ObservationData(datastr);
         GeneticAlgorithm ga;

         System.out.println("POPULAÇÃO: 1-Inserir  2-Random\n");
         int popType = in.nextInt();
         int dimPop;
         if(popType == 1) //POP INSERIDA PELO USER
         {
               System.out.println("Dimensão: \n");
               dimPop = in.nextInt();
         } else {
               System.out.println("Dimensão Random\n");
               Random gen = new Random();
               dimPop = gen.nextInt(100);
               System.out.println("Dimensão: "+ dimPop +"\n");
```

```
        }
        System.out.println("Método Selecção: 1-ROLETA  2-ELITISMO\n");
        int selectMethod =in.nextInt();
        if(selectMethod == 1)
                select = GeneticAlgorithm.ROULETTE_WHEEL_SELECTION;
        else
        {
                select = GeneticAlgorithm.ELITISM_SELECTION;
                System.out.println("Dimensão da elite:  1-Inserir  2-Default\n");
                int choice = in.nextInt();
                if(choice ==1) {
                        System.out.println("Dimensão: \n");
                        dimElite = in.nextInt();
                }
                else System.out.println("Dimensão Default: "+
GeneticAlgorithm.DEFAULT_ELITE_SIZE+"\n");
        }
        System.out.println("Operador de cruzamento: 1-OX1  2-0X2\n");
        int cross = in.nextInt();
        if(cross==1)
                crossoverType =RoverCircuit.OX1;
        else crossoverType = RoverCircuit.OX2;

        System.out.println("Probabilidade de cruzamento:  1-Inserir  2-Default\n");
        int crossP = in.nextInt();
        if(crossP == 1)
        {    System.out.println("Probabilidade: \n");
                float probC = in.nextFloat();
                probCrossover = probC;
        }
        else
                System.out.println("Probabilidade de cruzamento Default: "+
GeneticAlgorithm.DEFAULT_PROB_CROSSOVER+"\n");

        System.out.println("Operador de mutação: 1-TROCA  2-INVERSÃO\n");
        int mutate = in.nextInt();
        if(mutate == 1)
                mutationType = RoverCircuit.SWAP_MUT;
        else mutationType = RoverCircuit.INVERSION_MUT;
```

```
        System.out.println("Probabilidade de mutação:  1-Inserir  2-Default\n");
        int mutateP = in.nextInt();
        if(mutateP == 1)
        {   System.out.println("Probabilidade: \n");
                float probM = in.nextFloat();
                probMutation = probM;
        }
        else
                System.out.println("Probabilidade de mutação Default: "+
GeneticAlgorithm.DEFAULT_PROB_MUTATION+"\n");

        Individual[] ind = new Individual[dimPop];
        for(int i=0; i < ind.length; i++)
                ind[i] = new RoverCircuit(r, crossoverType, mutationType);

        Population p = new Population(ind);
        ga = new GeneticAlgorithm(p, probCrossover, probMutation);
        ga.setSelectCriteria(select,dimElite);

        System.out.println("Critério de Paragem: 1-NÚMERO DE GERAÇÕES
2-TEMPO LIMITE  3-VALOR DE FITNESS\n");
        int criterio = in.nextInt();
        if(criterio ==1)
                System.out.println("Número de Gerações: \n");
        if(criterio ==2)
                System.out.println("Limite de tempo (s): \n");
        if(criterio ==3)
                System.out.println("Limite do valor de fitness: \n");
        double valor = in.nextDouble();
        ga.setStopCriteria(criterio, valor);

        System.out.println("Hill-Climbing: 1-SIM  2-NÃO\n");
        hillOption = in.nextInt();
        if(hillOption ==1)
        {
                System.out.println("Hill-Climbing: SIM\n");
                ga.hillClimbingOption = 1;
        }
```

```
        if(hillOption ==2)
        {
                System.out.println("Hill-Climbing: NÃO\n");
                ga.hillClimbingOption =2;
        }

        getGraphic();
        ga.setDataSet(series);
        Individual best = ga.search();
        output.setText("");
        String s ="";
        s +="Best individual: "+best;
        s +="\nWorst individual: "+ga.getWorstIndividual();
        s+="\n"+ga.getResults();
        output.setText(s);
        System.out.println("--------------------");
        System.out.println("Best individual: "+best);
        System.out.println("Worst individual: "+ga.getWorstIndividual());
        System.out.println(ga.getResults());
}

public static String readFile(String filename) {
        Path path = Paths.get(filename);
        String filepath = path.toAbsolutePath().toString();
        String javaPath = filepath.replace("\\", "/");
        return javaPath;
}

public static void getGraphic() {
        frame = new JFrame();
        frame.setSize(700, 500);
        frame.setTitle("RoverCircuit");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);

        XYSeriesCollection dataset = new XYSeriesCollection();
        series = new XYSeries("Best Solution");
        dataset.addSeries(series);
```

```
        JFreeChart chart = ChartFactory.createXYLineChart("Genetic Algorithm",
"generation", "fitness", dataset);

        frame.add(new ChartPanel(chart),BorderLayout.CENTER);
        output = new JTextArea();
        output.setEditable(false);
        frame.add(output, BorderLayout.SOUTH);
    }

}
```