Universidade Federal de São carlos

Trabalho
Inteligência Artificial

Algoritmo Genético

Problema do Caixeiro Viajante

Nome: Iago Aleksander Zanardo
R.A.: 386162

Professor: Flavio Aldrovandi Montoro

Data: 21/01/2013

Indice

Descrição do Problema

O problema do caixeiro viajante (Travelling Salesman Problem – TSP) é de natureza combinatória e é uma referência para diversas aplicações (como projeto de circuitos integrados, roteamento de veículos, programação de produção, robótica, etc).

Em sua forma mais simples, no TSP o caixeiro deve visitar cada cidade somente uma vez e depois retornar a cidade de origem. Dado o custo da viagem (ou distância) entre cada uma das cidades, o problema do caixeiro é determinar qual o itinerário que possui o menor custo, obtendo uma resposta satisfatória em um determinado tempo.

Neste projeto, considere que o grafo (mapa) que representa as ligações entre as cidades é completo, ou seja, uma cidade se liga a todas as outras e que suas distâncias são randomizadas e variam entre 10 e 100.

.

Objetivo

Este trabalho consiste em implementar um algoritmo genético para resolver o problema do caixeiro, considerando 10 cidades e suas respectivas distâncias.
Serão realizadas simulações com e sem a aplicação de elitismo, nas quais será feita uma análise.

Definições do Algoritmo

Definições Iniciais ----------

- Número de cidades: 10

- Tamanho da população inicial: 200

- Número de iterações: 50 iterações sem melhoria da população ou 150

- Número de cromossomos preservados no elitismo: 4

- Número de cromossomos selecionados pelo critério de seleção para terem uma maior chance de permanecerem intactos para a próxima iteração: 4

- Taxa de cruzamento: 70%

- Taxa de mutação: 0.5%

- Taxa de seleção: 70%

Codificação cromossômica usada ----------

Durante todo o algoritmo os cromossomos são tratados como um vetor de inteiro, com cada valor representando uma cidade (de 0 a 9).

Na exibição, cada número é substituído por uma letra (de A a J, respectivamente).

Exemplo:
Durante o algoritmo: 0 1 2 3 4 5 6 7 8 9
Durante a exibição: A => B => C => D => E => F => G => H => I => J => A

Operador de seleção ----------

Os 4 cromossomos mais aptos possuem uma chance maior de permanecerem inalterados para a próxima iteração. Além da chance que todos os cromossomos possuem (a de não haver cruzamento menos a chance de haver mutação), estes possuem uma chance extra de não se reproduzirem e permanecerem intactos (a chance de haver seleção menos a chance de haver mutação).

Crossover ----------

Baseado em Order Operator (OX)
Constrói um descendente escolhendo uma subsequência de
um tour de um pai e preservando a ordem relativa das cidades do outro pai.

Exemplo:
pai 1 = (1 2 3 | **4 5 6 7** | 8 9)
pai 2 = (4 5 2 | **1 8 7 6** | 9 3) - mantém os segmentos selecionados
filho1 = (x x x | 4 5 6 7 | x x)
filho2 = (x x x | 1 8 7 6 | x x)

A seguir, partindo do ponto do segundo corte de um dos pais, as cidades do outro pai são copiadas na mesma ordem, omitindo-se as cidades que estão entre os pontos de corte. Randomizam-se as outras cidades nas posições restantes.

filho1 = (2 1 8 | 4 5 6 7 | 9 3)
filho2 = (3 4 5 | 18 7 6 | 9 2)

O crossover só acontece se houver o cruzamento e o critério de seleção nao beneficiar um dos pais (ou os dois).

Mutação ----------

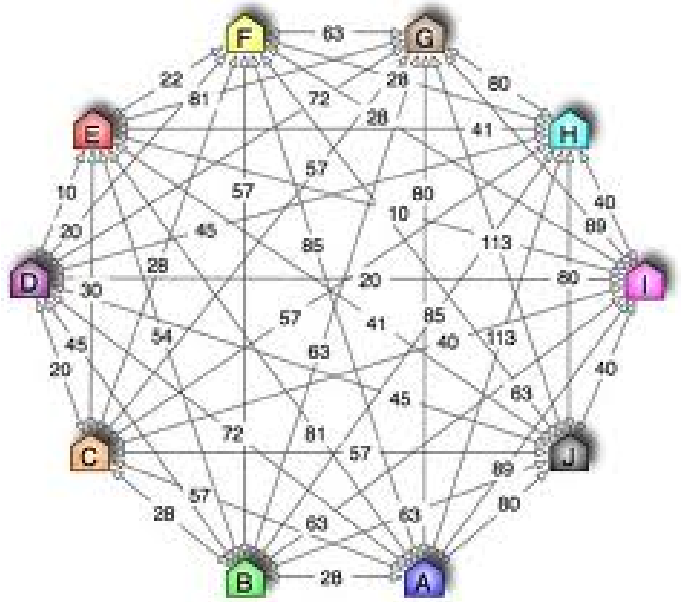
Reciprocal Exchange
A mutação de Troca Recíproca sorteia duas cidades e inverte sua posição.

Exemplo:
(1 2 3 **4** 5 6 **7** 8 9) → (1 2 3 **7** 5 6 **4** 8 9)

Função de avaliação ----------

A cada iteração é feita a soma das distâncias entre as cidades adjacentes de cada indivíduo e os respectivos resultados são organizados em ordem crescente. Quanto menor o resultado, menor a distância final percorrida pelo caixeiro viajante e mais apto está o cromossomo de ser o ideal.

A cada iteração ocorre, também, a renovação dos cromossomos, com a aplicação das chances de cruzamento, mutação, favorecimento de seleção e utilização ou não utilização do elitismo, deendendo da simulação.



Código Implementado

```
import java.util.*;
import java.io.*;

public class AlgoritmoGenetico {

	public static int NUMERO_CIDADES = 10;
	public static int NUMERO_POPULACAO = 200;
	public static int NUMERO_ITERACOES = 150;	// maximo de iteracoes
	public static int NUMERO_PRESERVADOS = 4;	// numero de cromossomos preservador no elitismo
	public static int NUMERO_SELECIONADOS = 4;	// critério de seleção,mais aptos tem maior chance
							// de sobreviverem intactos
	public static int TAXA_CRUZAMENTO = 70;		// taxa de cruzamento (em 100)
	public static int TAXA_MUTACAO = 5;		// taxa de mutação (em 1000)
	public static int TAXA_SELECAO = 70;		// taxa de selecao (em 100)

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

		// definicoes iniciais
		boolean mostrarEvolucao = false;
		boolean elitismo = false;
		int opcao = 49; //1 em ASCII
		int[][] mapa = new int [NUMERO_CIDADES][NUMERO_CIDADES];

		do {
		//limpa tela
			for (int i = 0; i < 24; i++)
				System.out.println("\n");

		//criar matriz mapa randomica
			mapa = gerarMapa();

		//exibicao da matriz mapa
			System.out.println("Mapa:\n");
			for( int[] a : mapa )
			    System.out.println( Arrays.toString( a ));

		//randomizar um novo mapa ou continuar com o mesmo
			System.out.println("\nTecle 1 para randomizar um novo mapa ou qualquer outro " +
									"valor para continuar o programa:");
			opcao = (int)System.in.read();
			new Scanner(System.in).nextLine();

		}while (opcao == 49);


		String[] cidades = { "A", "B", "C", "D", "E", "F", "G", "H", "I", "J" };

		int[][] cromossomos = new int[NUMERO_POPULACAO][NUMERO_CIDADES];
```

```
int[] resultados = new int[NUMERO_POPULACAO];
float media = 0;
int melhor = 0, pior = 0;
int MELHOR, contador = 0; //usados para o criterio de parada

//loop para rodar o programa quantas vezes o usuario desejar
do {
//loop para rodar o programa sem e com elitismo
for (int k = 0; k < 2; k++) {

gerarCromossomosAleatoriamente(cromossomos);
calcularResultado(cromossomos, resultados, mapa);
media = calcularMedia(resultados);
melhor = calcularMelhor (resultados);
pior = calcularPior (resultados);
ordenar(cromossomos, resultados);

MELHOR = melhor;

System.out.println("\nTecle 1 para mostrar a evolucao ou qualquer outro valor para " +
                                                                    "continuar o programa: ");
opcao = (int)System.in.read();
new Scanner(System.in).nextLine();

if (opcao == 49)
        mostrarEvolucao = true;

if (mostrarEvolucao)
        imprimir(cromossomos, resultados, media, melhor, pior, cidades);

for (int i = 0; i < NUMERO_ITERACOES; i++) {
        renovarCromossomos(cromossomos, elitismo);
        calcularResultado(cromossomos, resultados, mapa);
        media = calcularMedia(resultados);
        melhor = calcularMelhor (resultados);
        pior = calcularPior (resultados);
        ordenar(cromossomos, resultados);

        if (mostrarEvolucao) {
                System.out.println("Geracao: " + (i + 1));
                imprimir(cromossomos, resultados, media, melhor, pior, cidades);
        }

        if(melhor == MELHOR)
                contador++;
        else
        {
                contador = 0;
                MELHOR = melhor;
        }

        if (contador == 50)
                i = NUMERO_ITERACOES;

}

// mostrando resultado encontrado
```

```
            System.out.println("Simulacão " + (k+1) + " : ");
            resultado(cromossomos, resultados, cidades);

            elitismo = !elitismo;
            }

            System.out.println("\nTecle 1 para refazer a evolucao utilizando o mesmo mapa ou " +
                                                                    "qualquer outro valor para finalizar o programa: ");
            opcao = (int)System.in.read();
            new Scanner(System.in).nextLine();

            } while (opcao == 49);

}

// matriz de adjacencia e randomização de valores
private static int[][] gerarMapa() {

            int[][] mapa = new int [NUMERO_CIDADES][NUMERO_CIDADES];

            for( int i = 0 ; i < mapa.length ; i++ )
                        for ( int j = 0 ; j < mapa[i].length ; j++ )
                                    if (i == j)
                                    mapa[i][j] = 0;
                           else
                                    mapa[i][j] = 10 + new Random().nextInt(90);

            for( int i = 0 ; i < mapa.length ; i++ )
                        for ( int j = 0 ; j < mapa[i].length ; j++ )
                           mapa [i][j] = mapa [j][i];

            return mapa;
}

private static void resultado(int[][] cromossomos, int[] resultados, String[] cidades) {
            int i=0;
            for (int j = 0; j < NUMERO_CIDADES; j++) {
                        System.out.print(cidades[cromossomos[i][j]] + " => ");
            }
            System.out.print(cidades[cromossomos[i][0]] + " ");
            System.out.println(" Resultado: " + resultados[i]);

}

public static void renovarCromossomos(int[][] cromossomos, boolean elitismo) {

            //selecao --> pai apto irá sobreviver intacto e não gerará descendente

            int i, j, k, aux, n_iguais;
            boolean existente = false;
            boolean cromossomoValido;
            int descendentes, cruzamento, mutacao, selecao;

            int[][] cromossomosAux = new int[NUMERO_POPULACAO][NUMERO_CIDADES];

            //vetor para dizer qual pai já foi utilizado
            boolean [] paisUsados = new boolean[NUMERO_POPULACAO];
```

```
for (i=0; i<NUMERO_POPULACAO;i++)
	paisUsados[i] = false;

//se o elitismo nao for utilizado, a população toda terá uma chance de ser renovada
if (elitismo)
	descendentes = NUMERO_POPULACAO - NUMERO_PRESERVADOS;
else
	descendentes = NUMERO_POPULACAO;

int numFilho = NUMERO_POPULACAO-descendentes;

//substituindo
for (k = 0; k < descendentes; k+=2) {

		int[] cromossomoTmp1 = resetaCromossomo();
		int[] cromossomoTmp2 = resetaCromossomo();

		// pegando 2 pais aleatoriamente e ainda nao usados
		int pai1;
		int pai2;

		do {
			pai1 = new Random().nextInt(NUMERO_POPULACAO);
		} while (paisUsados[pai1] != false);
		paisUsados[pai1] = true;

		n_iguais = 0;
		cromossomoValido = true;

		do {
			pai2 = new Random().nextInt(NUMERO_POPULACAO);

			for (i = 0; i < NUMERO_CIDADES; i++)
				if (cromossomos[pai2][i] == cromossomos[pai1][i])
					n_iguais++;

			if (n_iguais == NUMERO_CIDADES)
				cromossomoValido = false;
			else
				cromossomoValido = true;

		} while ((paisUsados[pai2] != false) || (cromossomoValido == false));
		paisUsados[pai2] = true;

		//se o cruzamento ocorrer (chance de 70%)

		cruzamento = new Random().nextInt(100);
		if (cruzamento < TAXA_CRUZAMENTO)
		{
		//atribuicao de valores para os filhos

		//cromossomos mais aptos a sobreviver terão chancess maiores de continuarem
		// intactos na próxima iteração (critério de seleção)

		selecao = new Random().nextInt(100);
```

```
//Filho 1

//selecao
if (pai1 < NUMERO_SELECIONADOS && selecao < TAXA_SELECAO)
	cromossomosAux [numFilho++] = cromossomos[pai1];
else
{
//pegando uma sequência do pai 2
for (j = 3; j < 7; j++)
	cromossomoTmp1[j] = cromossomos[pai2][j];

//pegando restante do pai 1
for (j = 0; j < 10; j++)
	if (j < 3 || j > 6) {
		for (i = 3; i < 7; i++)
			if (cromossomos[pai1][j] == cromossomoTmp1[i])
				existente = true;

	if (existente == false)
		cromossomoTmp1[j] = cromossomos[pai1][j];
	else
		cromossomoTmp1[j] = valorValidoNoCromossomo(cromossomoTmp1);
	}

//verifica a ocorrência de mutação
mutacao = new Random().nextInt(1000);
if (mutacao < TAXA_MUTACAO)
{
	i = new Random().nextInt(10);
	j = new Random().nextInt(10);
	while (i == j)
		j = new Random().nextInt(10);

	aux = cromossomoTmp1[i];
	cromossomoTmp1[i] = cromossomoTmp1[j];
	cromossomoTmp1[j] = aux;
}

cromossomosAux [numFilho++] = cromossomoTmp1;
}

//Filho 2

//selecao
if (pai2 < NUMERO_SELECIONADOS && selecao < TAXA_SELECAO)
	cromossomosAux [numFilho++] = cromossomos[pai2];
else
{
//pegando uma sequência do pai 1
for (j = 3; j < 7; j++)
	cromossomoTmp2[j] = cromossomos[pai1][j];

//pegando o restante do pai 2
for (j = 0; j < 10; j++)
if (j < 3 || j > 6) {
	for (i = 3; i < 7; i++)
		if (cromossomos[pai2][j] == cromossomoTmp2[i])
```

```
                                        existente = true;

                        if (existente == false)
                                cromossomoTmp2[j] = cromossomos[pai2][j];
                        else
                                cromossomoTmp2[j] = valorValidoNoCromossomo(cromossomoTmp2);

        }

        //verifica a ocorrência de mutação
        mutacao = new Random().nextInt(1000);
        if (mutacao < TAXA_MUTACAO)
        {
                i = new Random().nextInt(10);
                j = new Random().nextInt(10);
                while (i == j)
                        j = new Random().nextInt(10);

                aux = cromossomoTmp2[i];
                cromossomoTmp2[i] = cromossomoTmp2[j];
                cromossomoTmp2[j] = aux;
        }

        cromossomosAux[numFilho++] = cromossomoTmp2;
        }

        }
        //se o cruzamento nao ocorrer uma cópia dos pais é feita
        else
        {
                //verifica a ocorrência de mutação no cromossomo 1
                mutacao = new Random().nextInt(1000);
                if (mutacao < TAXA_MUTACAO)
                {
                        i = new Random().nextInt(10);
                        j = new Random().nextInt(10);
                        while (i == j)
                                j = new Random().nextInt(10);

                        aux = cromossomoTmp1[i];
                        cromossomoTmp1[i] = cromossomoTmp1[j];
                        cromossomoTmp1[j] = aux;
                }
                cromossomosAux[numFilho++] = cromossomos[pai1];

                //verifica a ocorrência de mutação no cromossomo 2
                mutacao = new Random().nextInt(1000);
                if (mutacao < TAXA_MUTACAO)
                {
                        i = new Random().nextInt(10);
                        j = new Random().nextInt(10);
                        while (i == j)
                                j = new Random().nextInt(10);

                        aux = cromossomoTmp2[i];
                        cromossomoTmp2[i] = cromossomoTmp2[j];
                        cromossomoTmp2[j] = aux;
```

```
                                        }
                                        cromossomosAux[numFilho++] = cromossomos[pai2];
                                }

        }

        //se o elitismo for utilizado, uma parte da população será mantida
        if(elitismo)
                k = NUMERO_POPULACAO - descendentes;
        else
                k = 0;


        do {
                cromossomos[k] = cromossomosAux[k];
                k++;
        } while (k < NUMERO_POPULACAO);

}

// inicializando cromossomos aleatoriamente
private static int[][] gerarCromossomosAleatoriamente(int[][] cromossomos) {

        int[] cromossomosTmp = new int[NUMERO_CIDADES];

        int i, j;
        for (i = 0; i < NUMERO_POPULACAO; i++) {
                boolean crom_valido = false;
                while (!crom_valido) {
                        crom_valido = true;
                        cromossomosTmp = resetaCromossomo();

                        // gerando cromossomo - ok
                        for (j = 0; j < NUMERO_CIDADES; j++) {
                                cromossomosTmp[j] = valorValidoNoCromossomo(cromossomosTmp);
                        }
                        crom_valido = cromossomoValido(cromossomosTmp, cromossomos);
                }
                cromossomos[i] = cromossomosTmp;
        }
        return cromossomos;
}

//reseta cromossomo criando um vetor de -1
private static int[] resetaCromossomo() {
        int[] cromossomo = new int[NUMERO_CIDADES];
        int i;
        for (i = 0; i < NUMERO_CIDADES; i++)
                cromossomo[i] = -1;


        return cromossomo;
}

//não permite que uma cidade seja repetida. Retorna uma cidade randômica ainda não utilizada.
private static int valorValidoNoCromossomo(int[] cromossomosTmp) {
        int crom_temp;
        boolean valido;


        do {
```

```
                crom_temp = new Random().nextInt(NUMERO_CIDADES);
                valido = true;
                for (int i = 0; i < NUMERO_CIDADES; i++) {
                        if (cromossomosTmp[i] == crom_temp)
                                valido = false;
                }
        } while (!valido);

        return crom_temp;
}

//verifica se um cromossomo filho é válido. Este só será válido se for diferente
//de todos os cromossomos da geração anterior. Retorna se é válido ou não.
private static boolean cromossomoValido(int[] cromossomosTmp, int[][] cromossomos) {
        int j, i;
        boolean crom_valido = true;

        for (j = 0; j < NUMERO_POPULACAO; j++) {
                int n_iguais = 0;
                for (i = 0; i < NUMERO_CIDADES; i++)
                        if (cromossomosTmp[i] == cromossomos[j][i])
                                n_iguais++;

                if (n_iguais == NUMERO_CIDADES)
                        crom_valido = false;
        }

        return crom_valido;
}

//imprimindo os valores
private static void imprimir(int[][] cromossomos, int[] resultados, float media, int melhor, int pior, String[] cidades) {
        int i, i2;
        for (i = 0; i < NUMERO_POPULACAO; i++) {
                for (i2 = 0; i2 < NUMERO_CIDADES; i2++)
                        System.out.print(cidades[cromossomos[i][i2]] + " => ");

                System.out.print(cidades[cromossomos[i][0]] + " ");
                System.out.println(" Resultados: " + resultados[i]);
        }

        System.out.println("\nMelhor fitness da populacao: " + melhor);
        System.out.println("Pior fitness da populacao: " + pior);
        System.out.println("Media fitness da populacao: " + media + "\n\n");
}

// calculando o resultado para cada cromossomo
private static void calcularResultado(int[][] cromossomos, int[] resultados, int[][] mapa) {
        int i, j;

        for (i = 0; i < NUMERO_POPULACAO; i++) {
                int resTmp = 0;
                for (j = 0; j < NUMERO_CIDADES - 1; j++)
                        resTmp += mapa[cromossomos[i][j]][cromossomos[i][j + 1]];

                resTmp+=mapa[cromossomos[i][0]][cromossomos[i][j]];
                resultados[i] = resTmp;
        }
```

```
}

//calcular média fitness
private static float calcularMedia(int[] resultados) {

	float media = 0;

	for (int i = 0; i < NUMERO_POPULACAO; i++)
		media += resultados[i];
		media = media/ NUMERO_POPULACAO;

	return media;
}

//calcular melhor fitness
private static int calcularMelhor(int[] resultados) {

	int melhor = resultados[0];

	for (int i = 1; i < NUMERO_POPULACAO; i++)
		if (resultados[i] < melhor)
			melhor = resultados[i];

	return melhor;
}

//calcular pior fitness
private static int calcularPior(int[] resultados) {

	int pior = resultados[0];

	for (int i = 1; i < NUMERO_POPULACAO; i++)
		if (resultados[i] > pior)
			pior = resultados[i];

	return pior;
}

// ordenando
private static void ordenar(int[][] cromossomos, int[] resultados) {
	int i, j;
	for (i = 0; i < NUMERO_POPULACAO; i++)
		for (j = i; j < NUMERO_POPULACAO; j++)
			if (resultados[i] > resultados[j]) {
				int Aux;
				int[] Aux2 = new int[NUMERO_POPULACAO];

				//Ordenando os resultados
				Aux = resultados[i];
				resultados[i] = resultados[j];
				resultados[j] = Aux;

				//Colocando os cromossomos com seus respectivos resultados
				Aux2 = cromossomos[i];
				cromossomos[i] = cromossomos[j];
```

```
                                        cromossomos[j] = Aux2;
                                    }
                }

        }
```

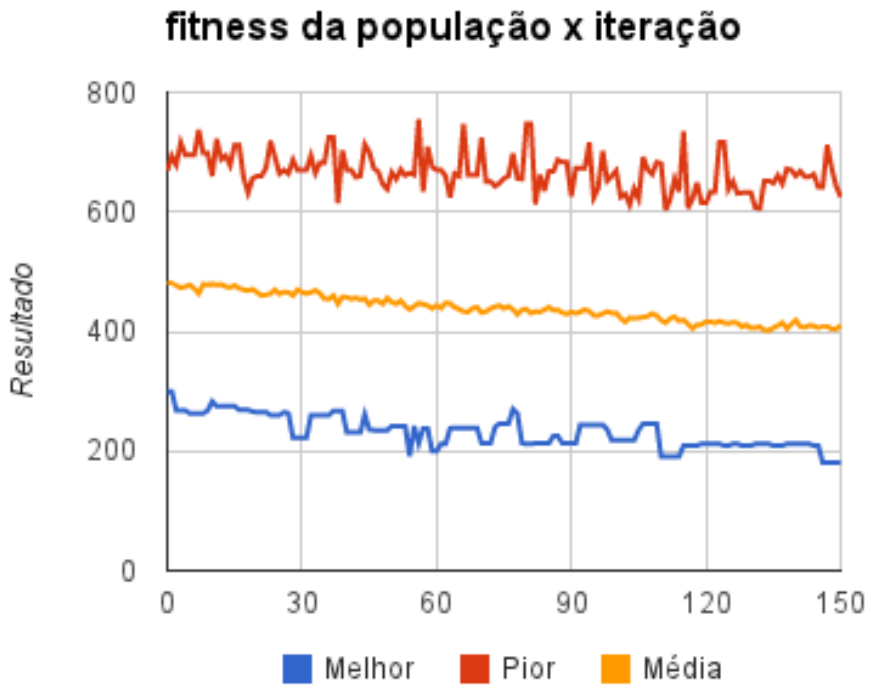
Simulação

Mapa gerado em certa execução do programa:

[0, 21, 66, 76, 13, 56, 28, 85, 64, 14]
[21, 0, 69, 57, 51, 69, 83, 13, 25, 62]
[66, 69, 0, 21, 20, 90, 56, 19, 63, 77]
[76, 57, 21, 0, 83, 39, 83, 19, 94, 17]
[13, 51, 20, 83, 0, 76, 39, 70, 10, 60]
[56, 69, 90, 39, 76, 0, 21, 22, 61, 54]
[28, 83, 56, 83, 39, 21, 0, 49, 22, 25]
[85, 13, 19, 19, 70, 22, 49, 0, 36, 38]
[64, 25, 63, 94, 10, 61, 22, 36, 0, 58]
[14, 62, 77, 17, 60, 54, 25, 38, 58, 0]

Gráfico de avaliação da população (Simulação 1) ----------

G => I => E => C => D => J => A => B => H => F => G Resultado: 181

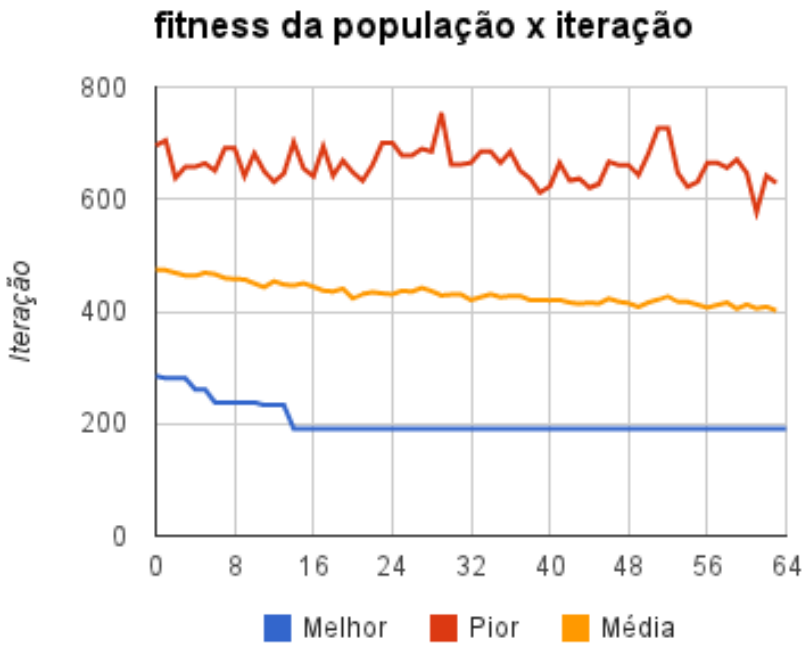


Outras simulações com o mesmo mapa e sem elitismo:

Geração 150: D => F => G => J => A => E => I => B => H => C => D Resultado: 200
Geração 150: J => D => C => E => I => B => H => F => G => A => J Resultado: 191
Geração 150:: I => H => B => A => J => G => F => D => C => E => I Resultado: 220

Gráfico de avaliação da população (Simulação 2) ----------

E => I => B => H => F => G => A => J => D => C => E Resultado: 191



Outras simulações com o mesmo mapa e com elitismo:
Geração 56: C => H => B => A => J => D => F => G => I => E => C Resultado: 196
Geração 68: D => C => E => A => B => I => H => F => G => J => D Resultado: 221
Geração 57: H => D => C => E => I => B => A => J => G => F => H Resultado: 198

Análise dos Gráficos Gerados

Através dos gráficos gerados através dos dados de melhor, pior e média de fitness da população a cada iteração, podemos observar que, apesar de ambas as simulações chegarem à resultados satisfatórios, a simulação 1 é muito menos eficiente (média de 2,5x a mais de iterações utilizadas), além de ser menos eficaz já que corre o risco de substituir o melhor resultado encontrado por algum que não seja tão bom assim. Já a simulação 2, com a utilização do elitismo, pode aumentar rapidamente o desempenho do AG, porque previne a perda da melhor solução já encontrada.