Universidade Tecnológica Federal
do Paraná

Bacharelado em Ciência da Computação

Inteligência Artificial

Problema do Caixeiro Viajante com Algoritmo Genético

Authors:
Higor Celante
Brendow Isidoro

Supervisor:
Dr. Rodrigo Hübner

November 7, 2018

Contents

1 Introdução

O Problema do Caixeiro Viajante (TSP - Travelling Salesman Problem) é um dos problemas de otimização combinatória mais estudados e conhecidos e se dá da seguinte forma:

Suponha que um caixeiro viajante tenha que visitar n cidades, passando por todas as cidades, começando e terminando numa mesma cidade, qual o melhor caminho? O problema de otimização é NP-difícil, enquanto o problema de decisão relacionado é NP-completo.

O objetivo deste trabalho é criar um Algoritmo Genético que encontre a melhor solução para o problema.

2 Algoritmo Genético

Algoritmos Genéticos foram descobertos por John Henry Holland (1975), com o objetivo inicial de estudar os fenômenos relacionados à adaptação das espécies e da seleção natural que ocorre na natureza (Darwin 1859), bem como desenvolver uma maneira de incorporar estes conceitos aos computadores (Mitchell 1997). São algoritmos de otimização global, baseados nos mecanismos de seleção natural e da genética.

3 Implementação

A linguagem para implementação escolhida foi Python. E foram utilizadas as bibliotecas *sys*, *numpy*, *math*, *random*, *copy*, *time* e *matplotlib*.

O programa inicia na chamada de *start(auxList, 500)*, onde o valor inteiro atribuído é o número de gerações.

3.1 Primeira Geração

Na primeira geração é criada a primeira população, no qual cada valor do cromossomo será definido (aleatoriamente) por 1 nó (Cidade) da base de dados:

```
genFirstPop(matriz, n):
... pop = []
... m = len(matriz)
... for i in range (n):
..... pop.append(random.sample(range(0, m), n))
... return pop
```

3.2 Nova Geração

Na função que cria a nova geração é passado por parâmetro 10 informações:

1. **pop** = A população atual.
2. **listFit** = A lista de todos os fitness dessa população.
3. **matriz** = A matriz de adjacência
4. **cRate** = Porcentagem de cruzamentos
5. **mRate** = Porcentagem de mutações
6. **mType** = Tipo de mutação a ser usado (1 para a primeira função de mutação, outro número qualquer para a segunda)
7. **alter** = Define se o crossover será ordenado ou alternativo (True para alternativo, False para ordenado)

8. **elit** = Define se será usado elitismo (bool)

9. **graph** = Define se vai plotar no gráfico (bool)

```
genNewPop(pop,listFit,matriz,cRate,mRate,mType,alter,elit,graph):
...
... ordedFit = selection(listFit)
... bestSolution = listFit[ordedFit[0]]
... cNum = ((len(pop)*cRate)/100)
... while i < cNum :
......newPop.append(crossOverAlt(pop[aux], pop))
......newPop.append(crossOverOrd(pop[aux], pop))
...
...return newPop
```

3.3 Fitness

O *Fitness* é a soma das distâncias de um vértice do cromossomo para o próximo, começando e terminando no mesmo ponto, passando por todos os pontos.

```
def fitness(pop, matriz):
... for i in range (len(pop)):
..... for j in range (len(pop[i])-1):
..... psum += getDist(pop[i][j], pop[i][j+1], matriz)
..... psum += getDist(pop[i][j], pop[i][0], matriz)
..... listFit.append(psum)
..... psum = 0
... return listFit
```

3.4 Crossover

3.4.1 Crossover Ordenado

O *Crossover* Ordenado consiste em randomizar um número de elementos a serem passados dos pais para o filho, ordenando esse elementos e inserindo nos indices selecionados para o *crossover*.

```
crossOverOrd(p1, p2):
... r = copy.copy(p1)
... p = random.sample(range(0, len(p1)), random.randint(1, len(p1)-1))
... p.sort()
... for i in p:
..... s.append(p1[i])
... for i in p2:
..... if i in s:
..... p_ord.append(s.index(i))
... for i in range (len(s)):
..... r[p[i]] = s[p_ord[i]]
... return r
```

3.4.2 Crossover Alternativo

O *Crossover* Alternativo randomiza o índice de corte, copia para o cromossomo filho o cromossomo do pai (p1) até o ponto de corte, busca no segundo pai (p2) os elementos que ainda não estão no filho e os adiciona até preencher o cromossomo.

```
crossOverAlt(p1, p2):
... corte = random.randrange (0, len(p1)-1)
... r = copy.copy(p1[0:corte])
... pos = 0
... while len(r) ¡ len(p1):
..... if (p1[pos] not in r[0:len(r)]):
......... r.append(p1[pos])
......... pos += 1 ..... elif (p2[pos] not in r[0:len(r)]):
......... r.append(p2[pos])
......... pos += 1
..... else:
......... prox = encontraprox(p1, p2, r)
......... r.append(prox)
......... pos += 1
... return r
```

3.5 Mutações

3.5.1 Mutação 1

A mutação 1 seleciona um índice aleatório, assim invertendo o valor do elemento, com o próximo a direita. Caso seja selecionado a última posição do vetor, troca-se com a primeira.

```
mutation1 (chromossome):
... n = len (chromossome) - 1
... gene = random.randint(0, n)
... if gene == n:
..... aux = chromossome[gene]
..... chromossome[gene] = chromossome[0]
..... chromossome[0] = aux
... else:
..... aux = chromossome[gene]
..... chromossome[gene] = chromossome[gene+1]
..... chromossome[gene+1] = aux
... return chromossome
```

3.5.2 Mutação 2

A mutação 2 seleciona dois índices aleatórios, invertendo os valores na posição do cromossomo.

```
mutation2 (chromossome):
... n = len (chromossome) - 1
... gene = random.randint(0, n)
... rand = random.randint(0, n)
... while rand == gene:
..... rand = random.randint(0, n)
... aux = chromossome[gene]
... chromossome[gene] = chromossome[rand]
... chromossome[rand] = aux
... return chromossome
```

3.6 Elitismo

No elitismo a função "encontrakpior" retorna os "qtd" melhores fitness numa lista de fitness ordenada do melhor para o pior.

```
elitismo(pop, listfit, qtd):
... sortedFit = selection(listfit)
... for i in range(qtd):
..... newpop.append(pop[sortedFit[i]])
... return newpop
```

4 Execução

Basta executar o arquivo "main.py" com o python 2.7 e passar como parâmetro o nome da base de dados escolhida. Exemplo:

#user: python main.py a280tsp.txt

A execução irá retornar o plot dos resultados alcançados e retornar o melhor indivíduo encontrado. Exemplo:

#user: Melhor Caminho = [1,4,6,7,9,8,0]

5 Teste

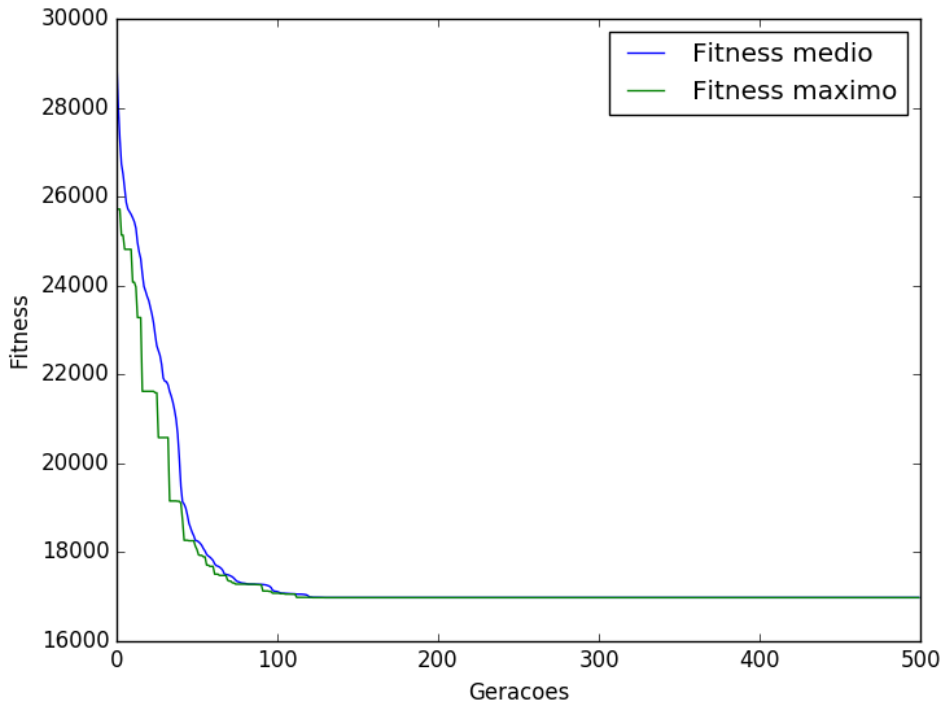
A base de dados demonstrada será a berlin52.tsp

Fitness médio - Se refere à média dos fitness de cada geração

Fitness máximo - Se refere ao melhor fitness de cada geração

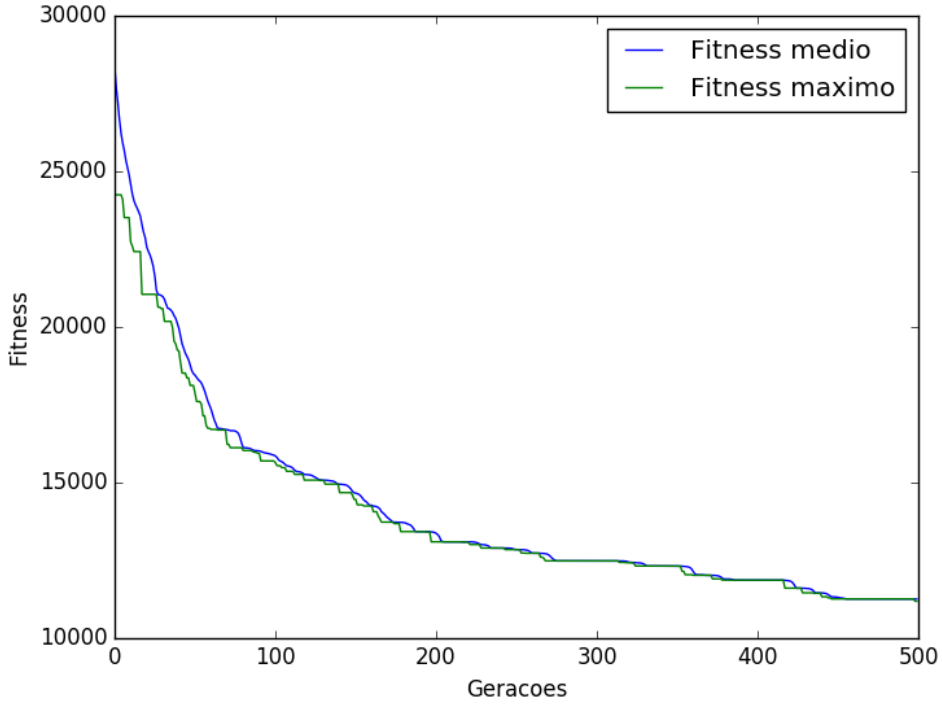
5.1 Testes com Elitismo

5.1.1 Cross-over ordenado com mutação 1



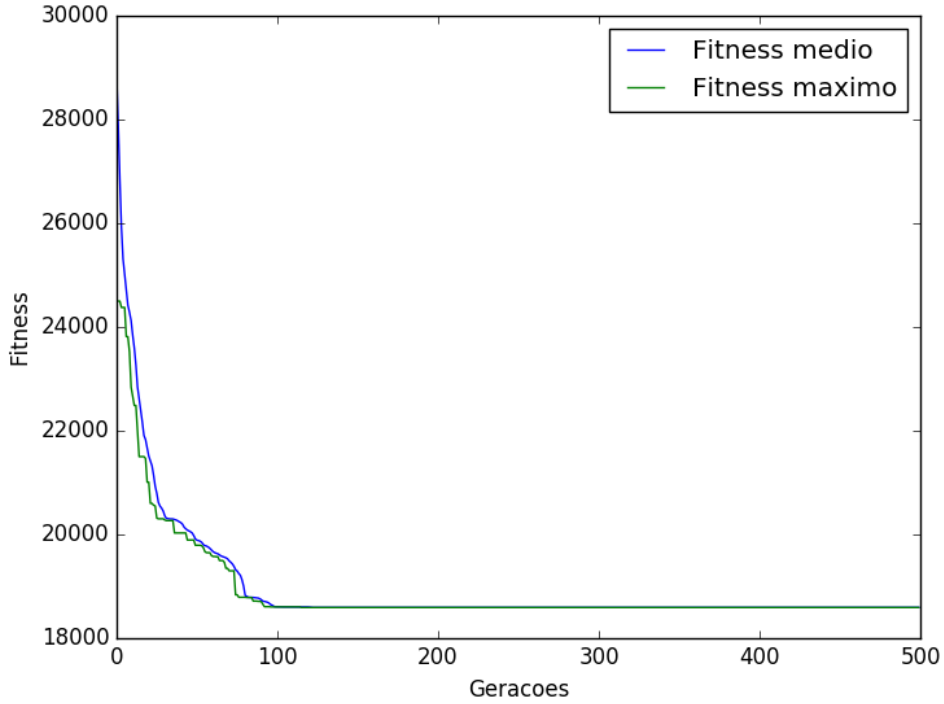
Média dos Resultados = 16980.903420234947

5.1.2 Cross-over ordenado com mutação 2



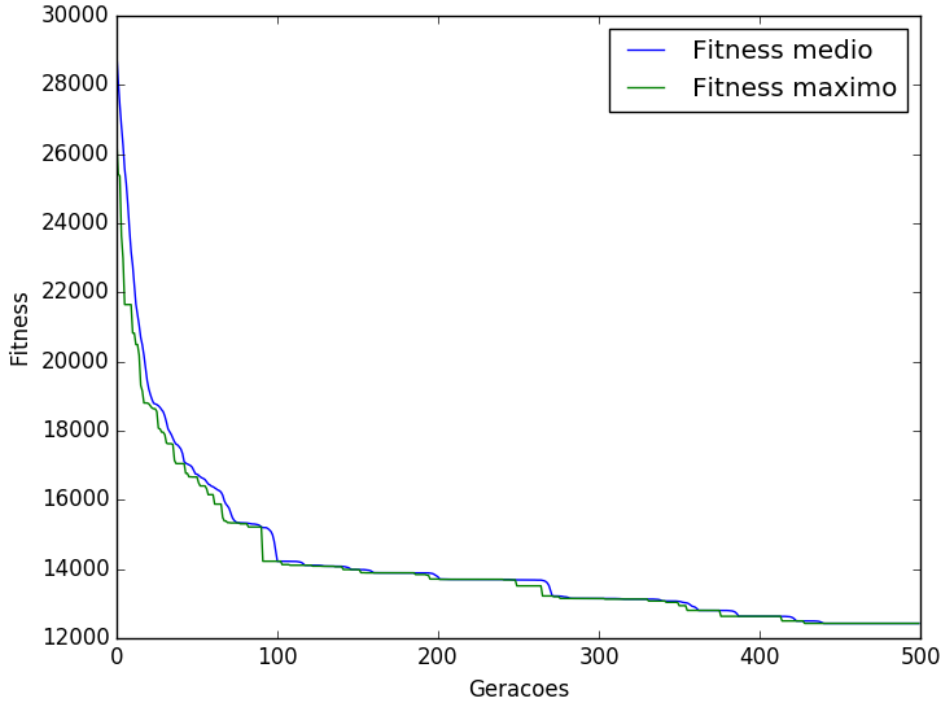
Média dos Resultados = 11258.678750176141

5.1.3 Cross-over alternativo com mutação 1



Média dos Resultados = 18592.437090404223

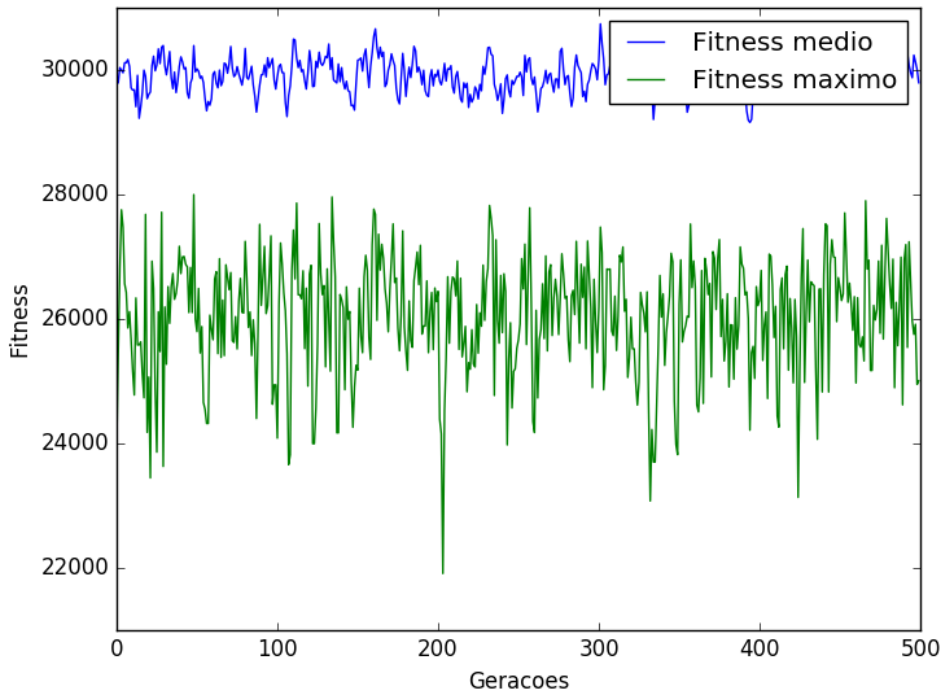
5.1.4 Cross-over alternativo com mutação 2



Média dos Resultados = 29720.17811291862

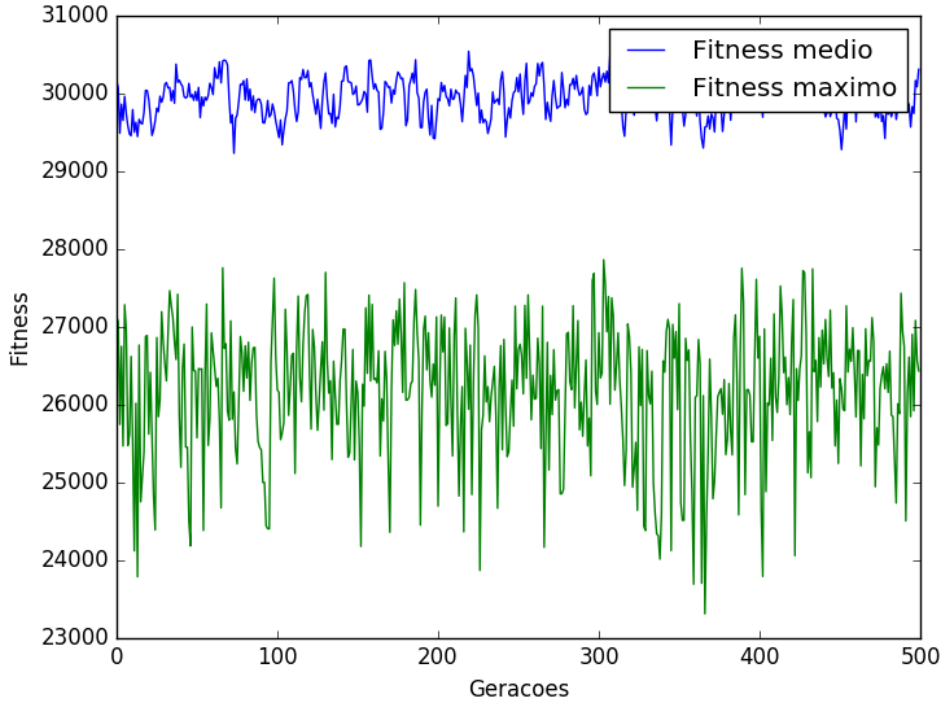
5.2 Testes sem Elitismo

5.2.1 Cross-over ordenado com mutação 1



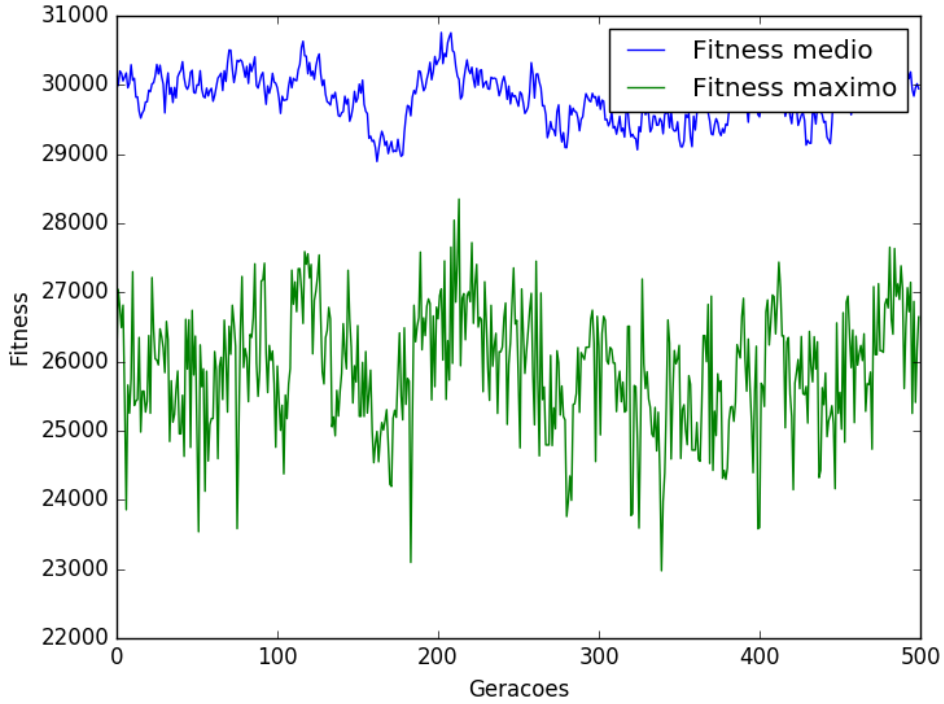
Média dos Resultados = 29796.28620185845

5.2.2 Cross-over ordenado com mutação 2



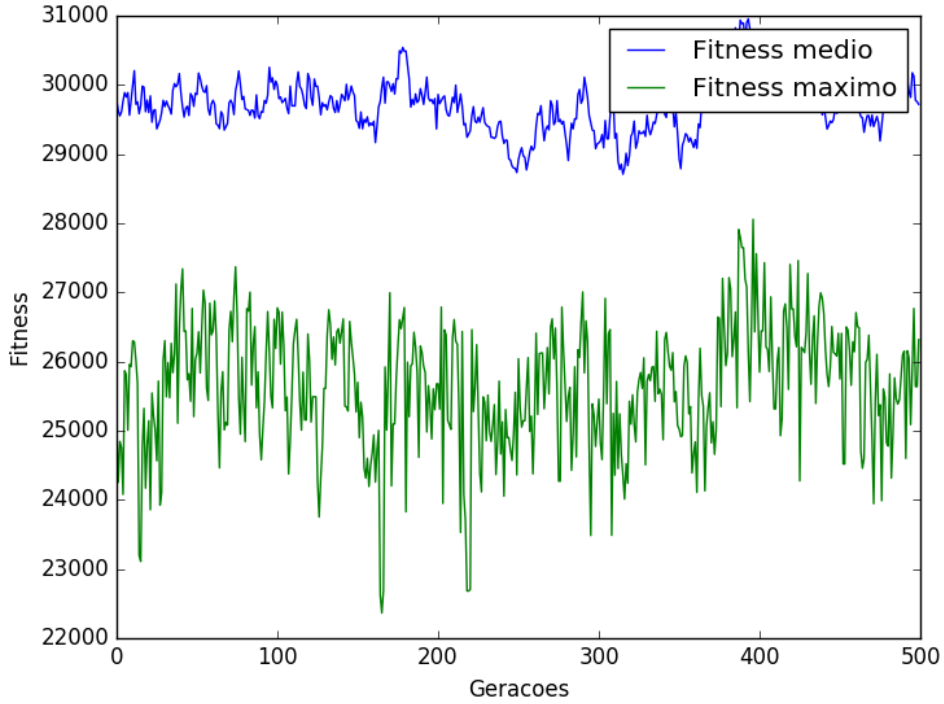
Média dos Resultados = 30311.735910200732

5.2.3 Cross-over alternativo com mutação 1



Média dos Resultados = 29945.097622357433

5.2.4 Cross-over alternativo com mutação 2



Média de Resultados = 29720.17811291862

6 Conclusões

Concluindo a análise dos gráficos e resultados, percebe-se que o melhor foi alcançado pelo teste com a mutação 2, com crossover ordenado e elitismo. Era esperado esse resultado por propriedades do algoritmo genético, no qual, quando possuímos o elitismo, o resultado será mais próximo do ideal. Os casos sem elitismo, acabam gerando valores muito distantes do ideal, resultando valores mais aleatórios.