Inteligência Computacional

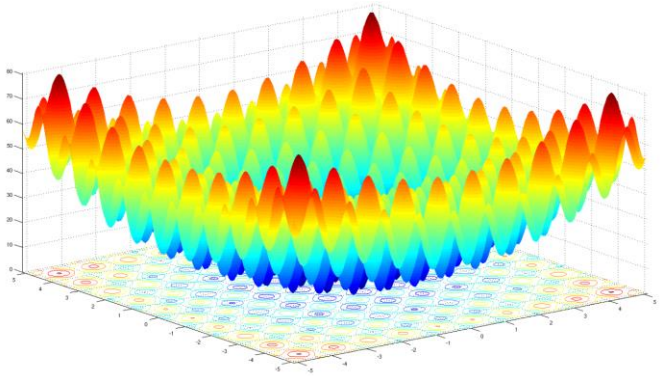
Implementação de Algoritmo Genético para minimizar função

Michel Monteiro Zerbinati

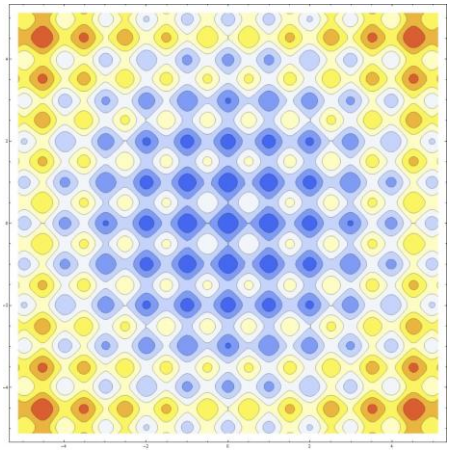
Função a ser minimizada

$$f(x, y) = 20 + x\,2 + y\,2 - 10[\cos(2\pi x) + \cos(2\pi y)]$$

Gráficos da função



3D



Contorno

Estratégia utilizada

Para este trabalho foi desenvolvido um algoritmo genérico parametrizado, o qual é possível escolher entre diversos tipos de seleção, de crossover e de mutação, além de outros parâmetros configuráveis, como por exemplo, taxa de crossover, de mutação, quantidade de gerações, indivíduos por geração, elitismo, entre outros

Abaixo exibirei brevemente quais são os principais tipos de seleção, crossover e mutação implementados, o qual após serão realizados alguns experimentos e análises para que possamos chegar a uma conclusão final.

Tipo de Seleção

Seleção por Roleta (Roulete Wheel)

Neste tipo de seleção, os pais são selecionados de acordo com a proporção de seu fitness, ou seja, indivíduos com fitness maior tem melhores chances de serem selecionados.

No nosso caso, onde desejamos minimizar a função, foi necessário um tratamento/normalização do fitness dos indivíduos para que este tipo de seleção fosse utilizada. Na função de avaliação de cada indivíduo, onde geramos o valor do fitness, o sinal do resultado foi invertido f(x) => - f(x), para que o menor valor de função fosse considerado como um fitness melhor (maior) que os resultados de função maiores. Por esse motivo, para que a roleta pudesse ser aplicada corretamente, efetuamos a seguinte normalização:

1. Foi verificado qual o menor valor de fitness (após o resultado da função ter sido invertido
2. Este menor valor foi armazenado, transformado em valor positivo, e adicionado + 1
3. Todos os fitness foram somados com o valor gerado no passo anterior

Abaixo, um exemplo com 5 indivíduos

Indivíduo	Fitness -f(x)	Fitness normalizado p/ Roleta
1	-3	13
2	-5	11
3	-7	9
4	-9	7
5	-15	1

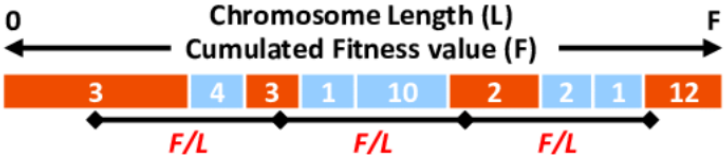
Ou seja, o valor -15 foi transformado em +15 e somado +1 , ficando +16. O valor +1 foi somado para que mesmo o pior indivíduo tivesse chance de ser selecionado, apesar desta chance ser bem pequena, comparada com os demais.

Após esta normalização, a Seleção por Roleta foi aplicada normalmente.

Ponto negativo: pode causar uma prematura convergência para um ótimo local. Tem performance ruim se um ou até mais membros da população fitness altos comparados aos outros membros

Amostragem Universal Estocástica (Stochastic Universal Sampling - SUS)

Esta estratégia é similar a Seleção por Roleta, mas tenta reduzir o risco de cair em um ótimo local. Resumidamente, os individuos são “embaralhados” mas mantém suas proporções de espaço de acordo com seu fitness. É calculada a média da soma dos fitness de acordo com a quantidade de seleções a serem realizadas, um valor delta é definido de acordo com um alfa aleatório e, a partir deste delta, é selecionado o primeiro membro dentro da população, sendo os demais sendo proporcionais com a média que foi calculada. Abaixo, uma imagem que exibe o comportamento do SUS:



Para a amostragem universal, os dados fora normalizados da mesma forma que foram para a seleção por roleta

Ranking Linear (Linear Rank Selection – LRS)

Este estratégia também é uma variação da Seleção por Roleta que tenta superar o problema da convergência prematura em ótimos locais (e não globais). Ele é baseado no ranking(ordem) dos indivíduos ao invés do fitness, onde se é ordenado do melhor para o pior, sendo o pior com o rank = 1 e o melhor indivíduo com o rank = n. Assim, de acordo com o rank, cada indivíduo tem a seguinte probabilidade de ser selecionado:

$$p(i) = \frac{rank(i)}{n * (n-1)}$$

Torneio (Tournament Selection – TOS)

O princípio do Tornei é selecionar aleatoriamente uma quantidade de indivíduos e, o melhor dentre esta disputa é escolhido. No nosso algoritmo, definimos uma taxa de seleção por torneio (por padrão 0.8), onde um número aleatório é gerado e, caso ele seja menor que esta taxa, o melhor do torneio é selecionado. Caso contrário, o pior é selecionado. Caso seja desejável que somente os melhores de cada torneio sejam selecionados, sem probabilidade alguma de o pior vencer, basta setar esta taxa para 1.

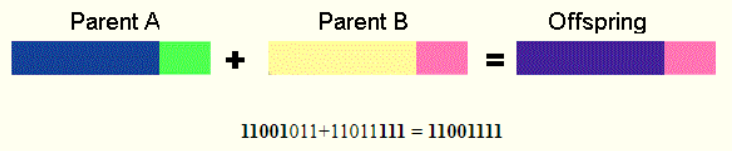
Elitismo

Também foi incluso um parâmetro de Elitismo, ou seja, onde é possível informar a quantidade dos melhores indivíduos que desejamos preservar para a próxima geração. Resumindo, caso o valor deste parâmetro seja 2, os 2 melhores individuos da geração serão clonados para a a próxima geração. Esta é uma forma de não perdermos os melhores indivíduos.

Tipo de Crossover

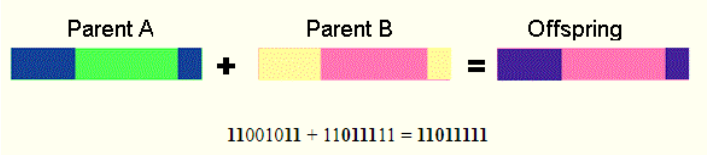
Único Ponto

Neste crossover um ponto de corte é definido aleatoriamente, onde o Filho 1 herdará a primeira parte do Pai 1 (até o corte) e a segunda parte (após o corte) do Pai 2, e o Filho 2 vice-versa, conforme imagem abaixo:



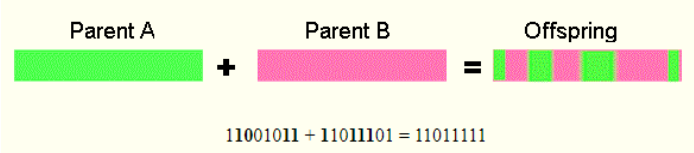
Ponto Duplo

Neste crossover, dois pontos de cortes são definidos aleatoriamente, onde o Filho 1 herdará a primeira parte do Pai 1 (até o corte 1), herdará a segunda parte (após o corte 1 e até o corte 2) do Pai 2, e a terceira parte (do corte 2 até o final) do Pai 1 novamente, e o Filho 2 vice-versa, conforme imagem abaixo:



Uniforme

Neste crossover, os genes são copiados aleatoriamente, do Pai 1 eou do Pai 2 para o Filho, conforme imagem abaixo:



Tipo de Mutação

Único Gene

É verificado se haverá mutação (caso o valor aleatório gerado seja menor que a taxa definida) e, se sim, um único gene é escohido aleatoriamente do individuo e alterado.

N Genes

É verificado se haverá mutação (caso o valor aleatório gerado seja menor que a taxa definida) e, se sim, uma quantidade aleatória de n genes é alterada. A quantidade n também é aleatoriamente definida.

Todos Genes

A taxa de mutação é verificada para cada gene do indivíduo e se for definido que mutação será efetuada, o gene é alterado.

Experimentos

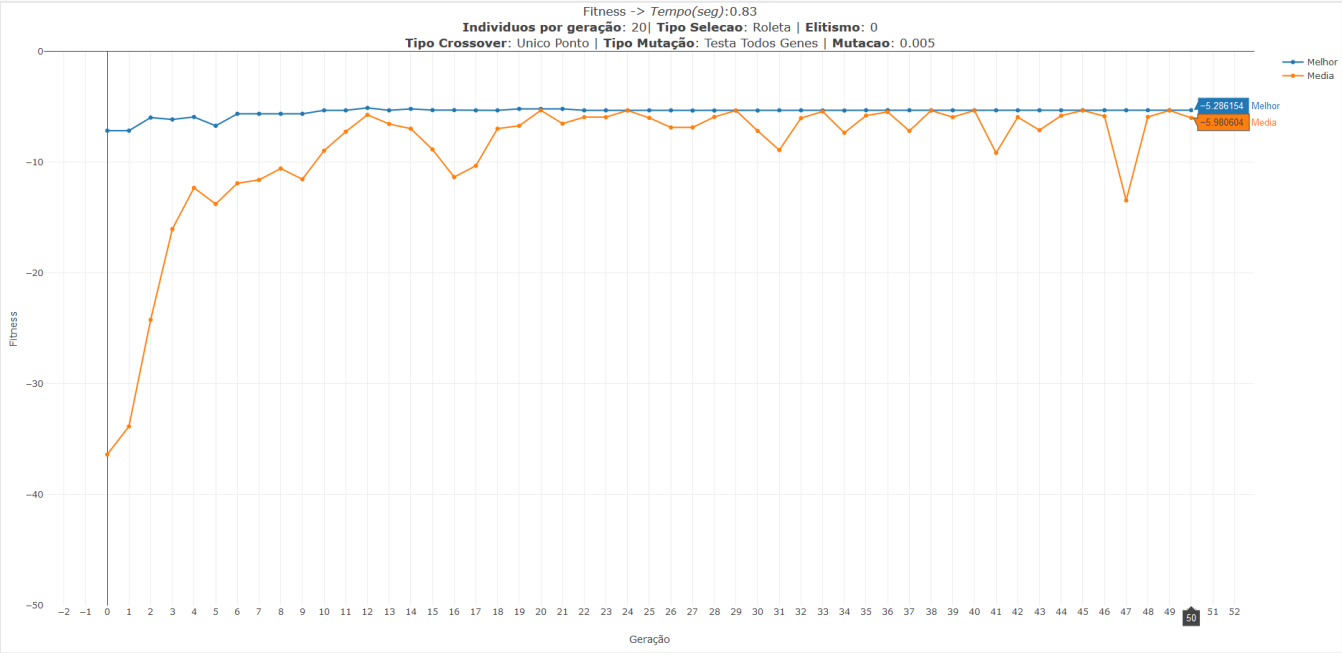
Como a quantidade de combinações possíveis é grande, de acordo com os parâmetros que foram desenvolvidos, entre os quais alguns estão citados nos itens acima, escolhi por efetuar alguns experimentos iniciais para verificar o comportamento padrão para as principais configurações, os quais estão no quadro abaixo:

Gerações	Individuos	Seleção	Crossover	Mutação
50	20	ROLETA	ÚNICO	VERIFICAÇÃO DE TODOS GENES
50	20	ESTOCASTCA	ÚNICO	VERIFICAÇÃO DE TODOS GENES
50	20	TORNEIO	ÚNICO	VERIFICAÇÃO DE TODOS GENES
50	20	RANK	ÚNICO	VERIFICAÇÃO DE TODOS GENES
50	20	ROLETA	DUPLO	VERIFICAÇÃO DE TODOS GENES
50	20	ESTOCASTCA	DUPLO	VERIFICAÇÃO DE TODOS GENES
50	20	TORNEIO	DUPLO	VERIFICAÇÃO DE TODOS GENES
50	20	RANK	DUPLO	VERIFICAÇÃO DE TODOS GENES
50	20	ROLETA	UNIFORME	VERIFICAÇÃO DE TODOS GENES
50	20	ESTOCASTCA	UNIFORME	VERIFICAÇÃO DE TODOS GENES
50	20	TORNEIO	UNIFORME	VERIFICAÇÃO DE TODOS GENES
50	20	RANK	UNIFORME	VERIFICAÇÃO DE TODOS GENES

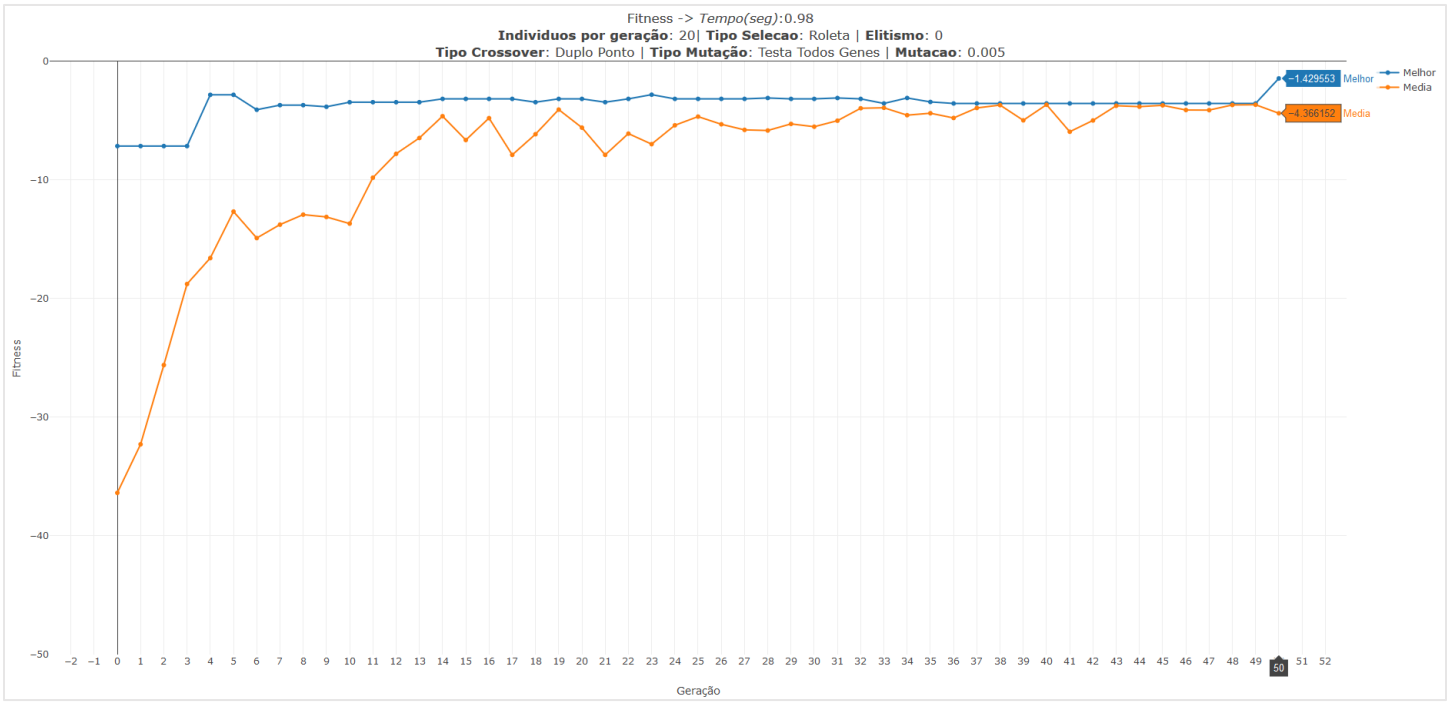
O número 50 de gerações foi escolhido inicialmente pois identifiquei que para um número entre 20 e 30 indivíduos, na maioria dos casos o melhor indivíduo é gerado antes da geração 45.

Para minimizar as diferenças iniciais, todos os experimentos acima foram iniciados com o mesmo grupo e combinações de individuos.

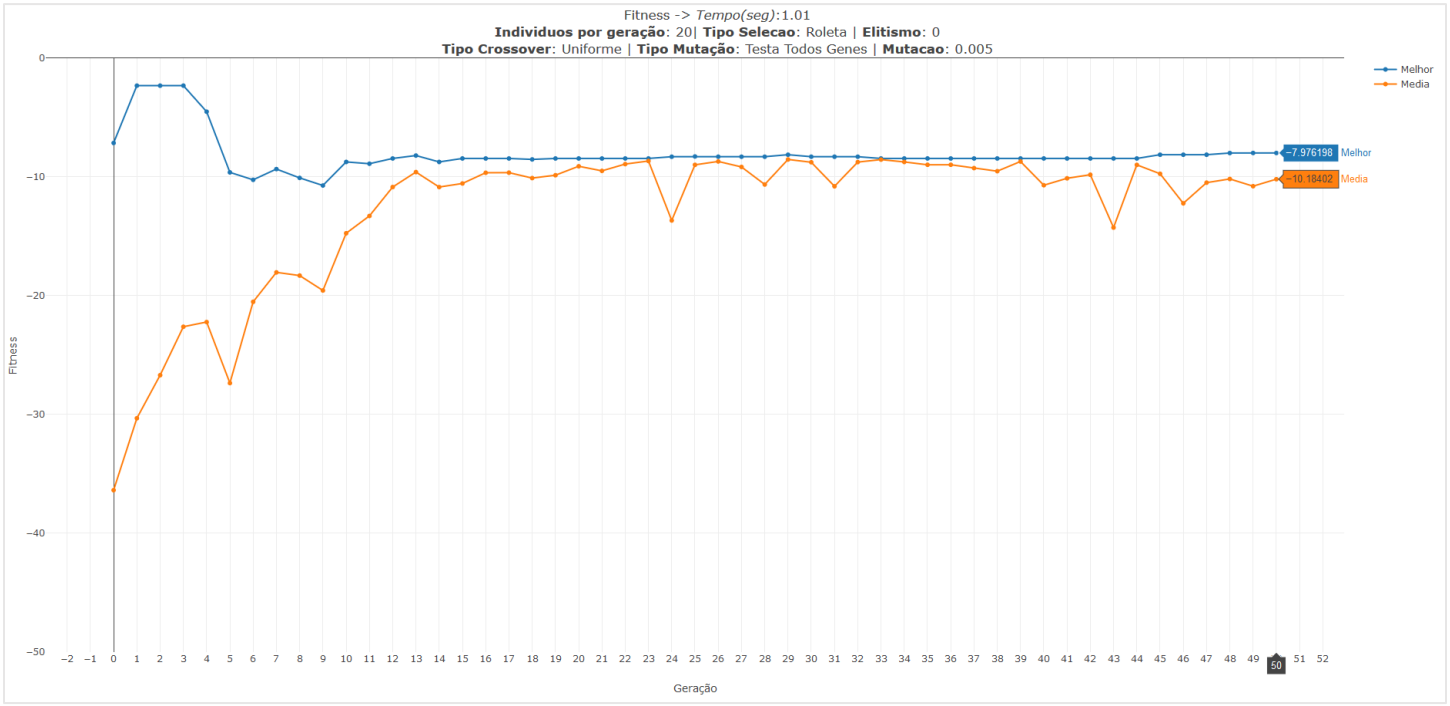
Abaixo seguem os gráficos das execuções e resultados da evolução do melhor indivíduo de cada geração de da média das gerações:



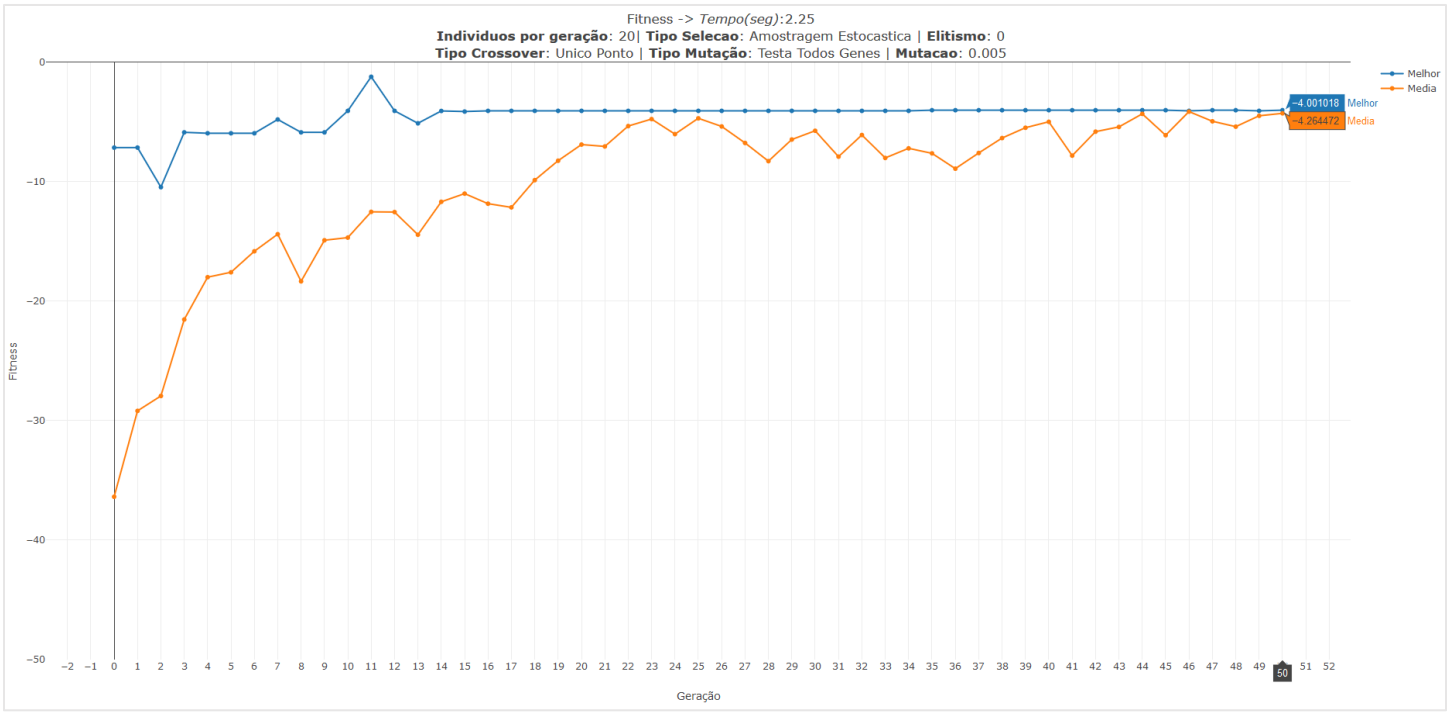
Roleta – Crossover único ponto



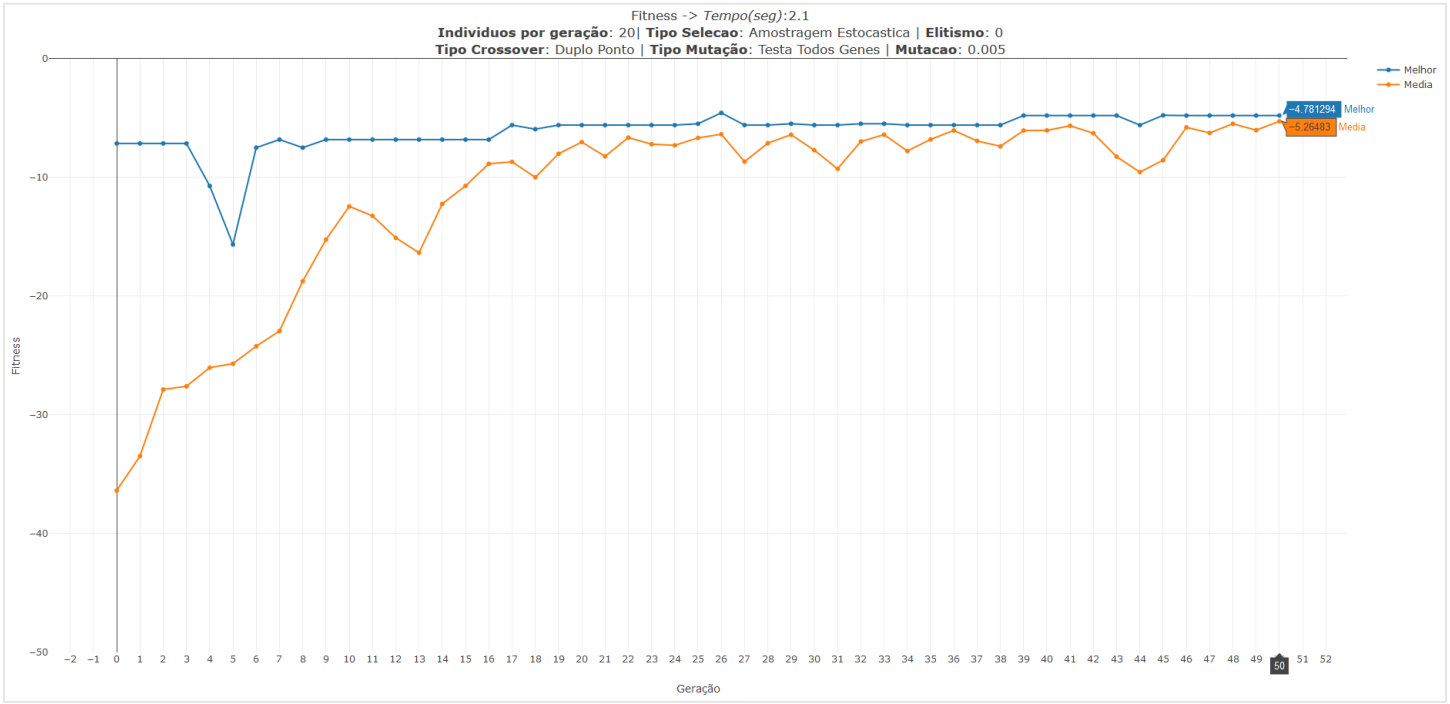
Roleta – Crossover duplo ponto



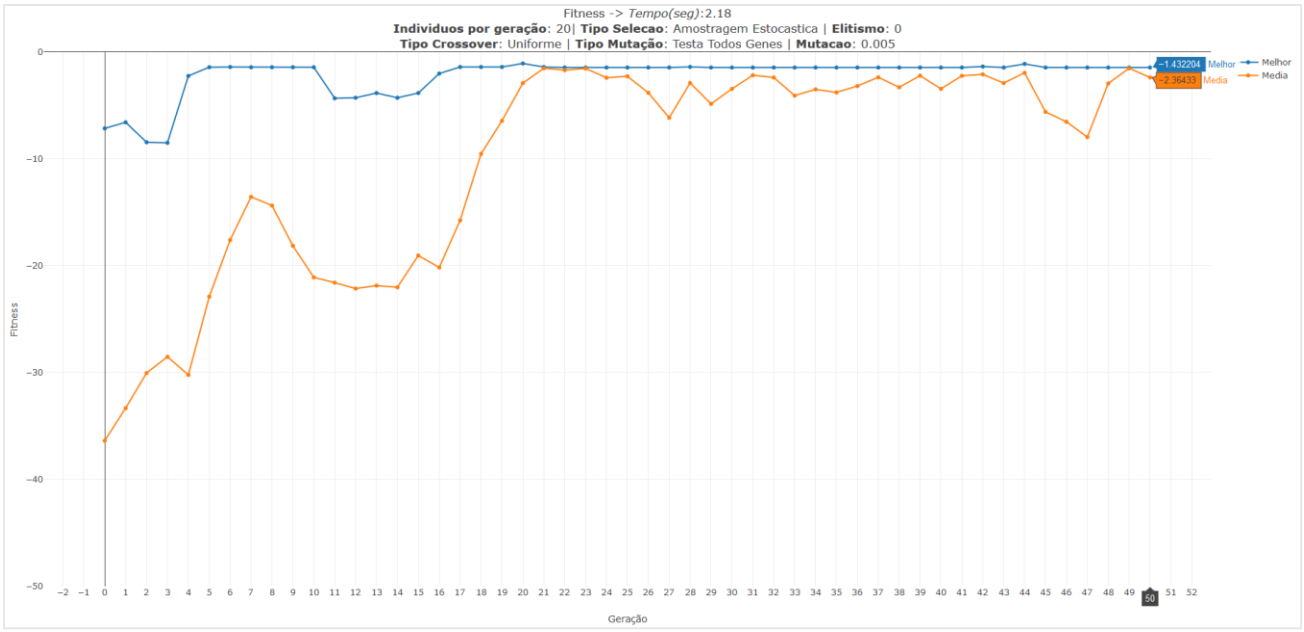
Roleta – Crossover uniforme



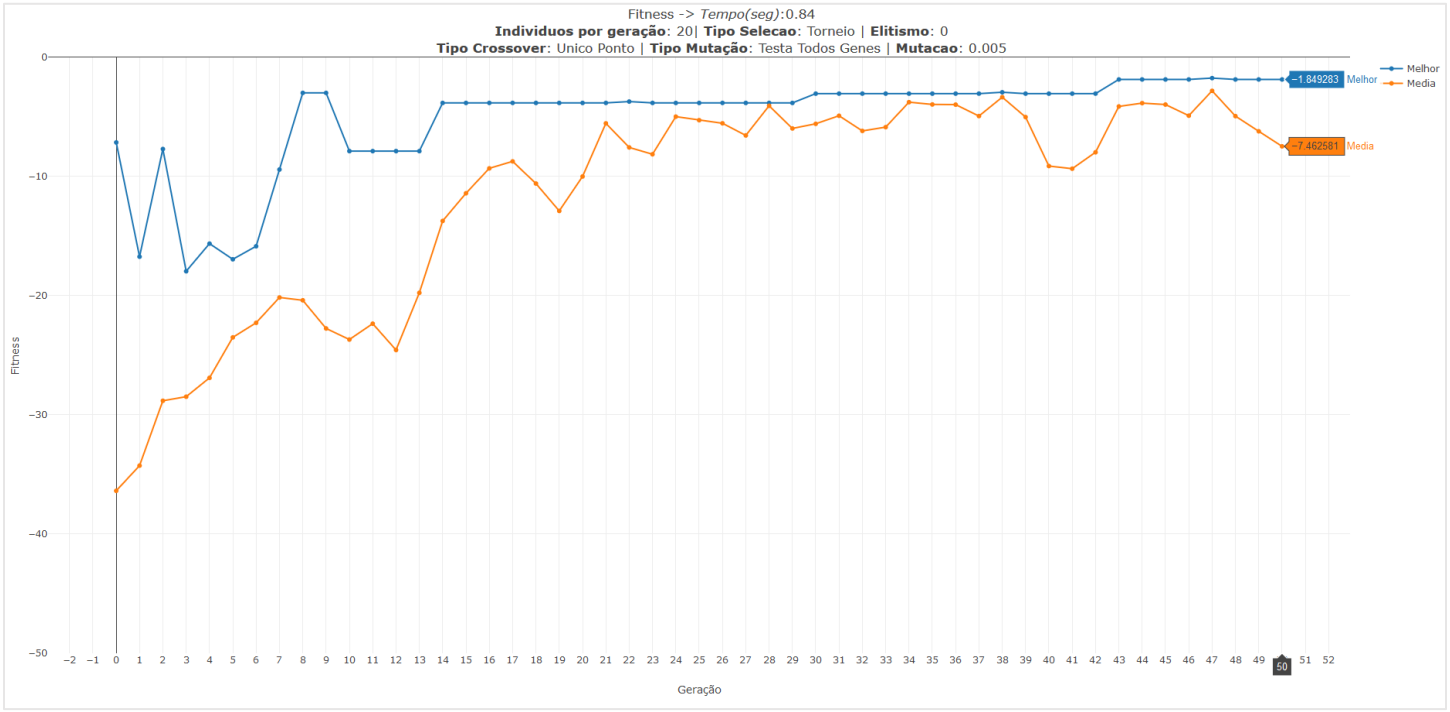
Amostragem Estocástica – Crossover único Ponto



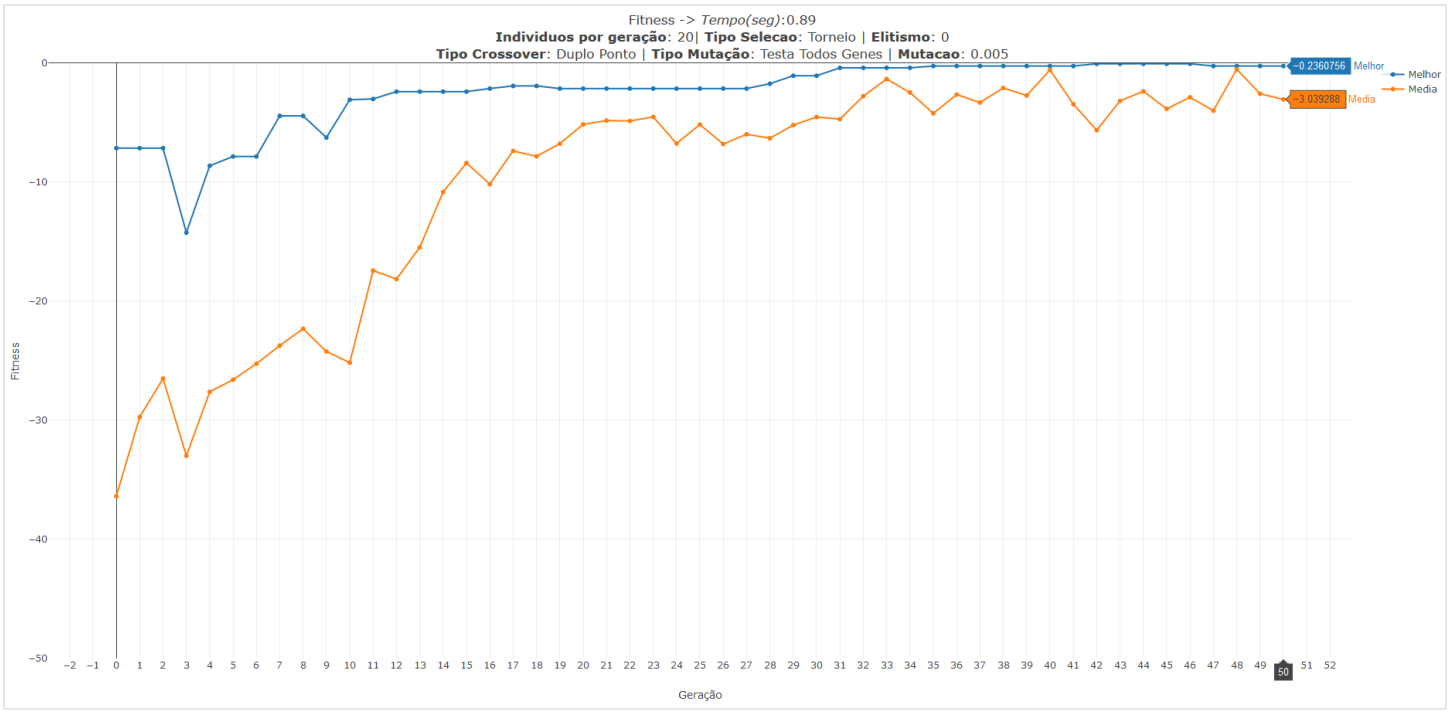
Amostragem Estocástica – Crossover duplo Ponto



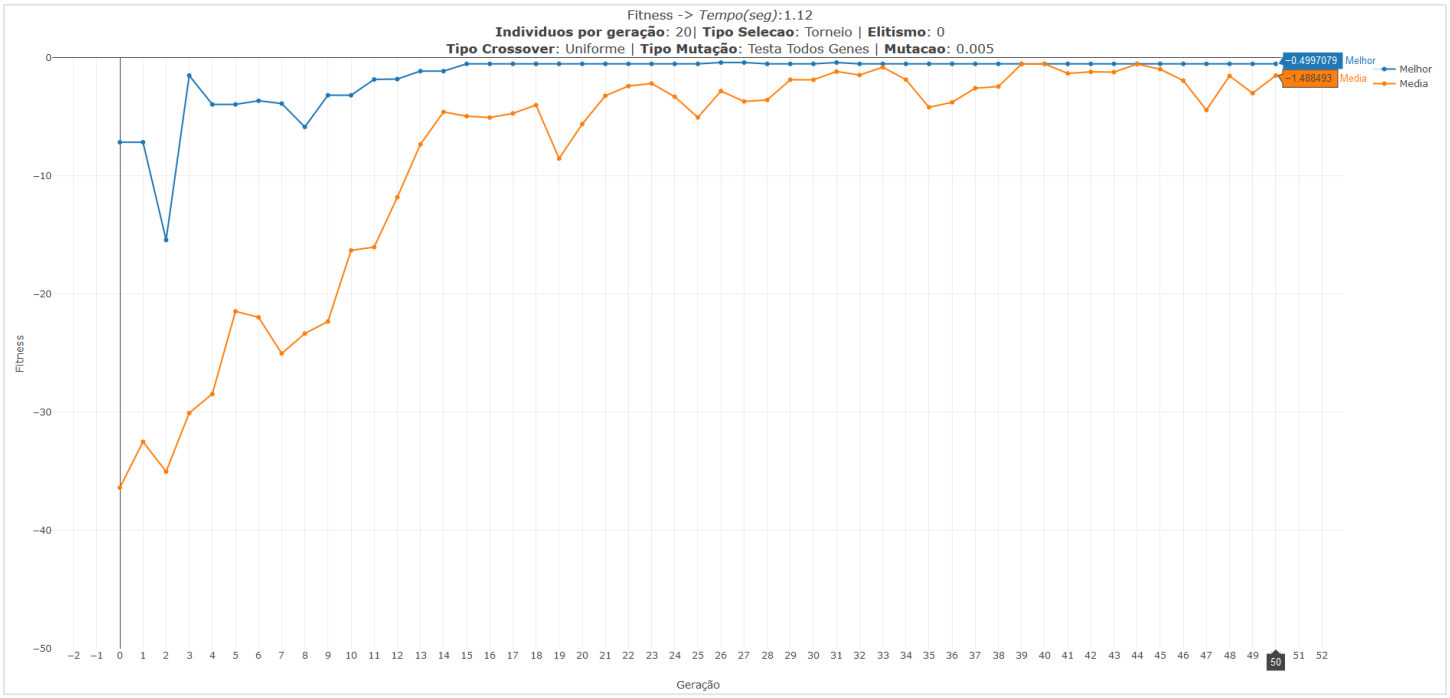
Amostragem Estocástica – Crossover uniforme



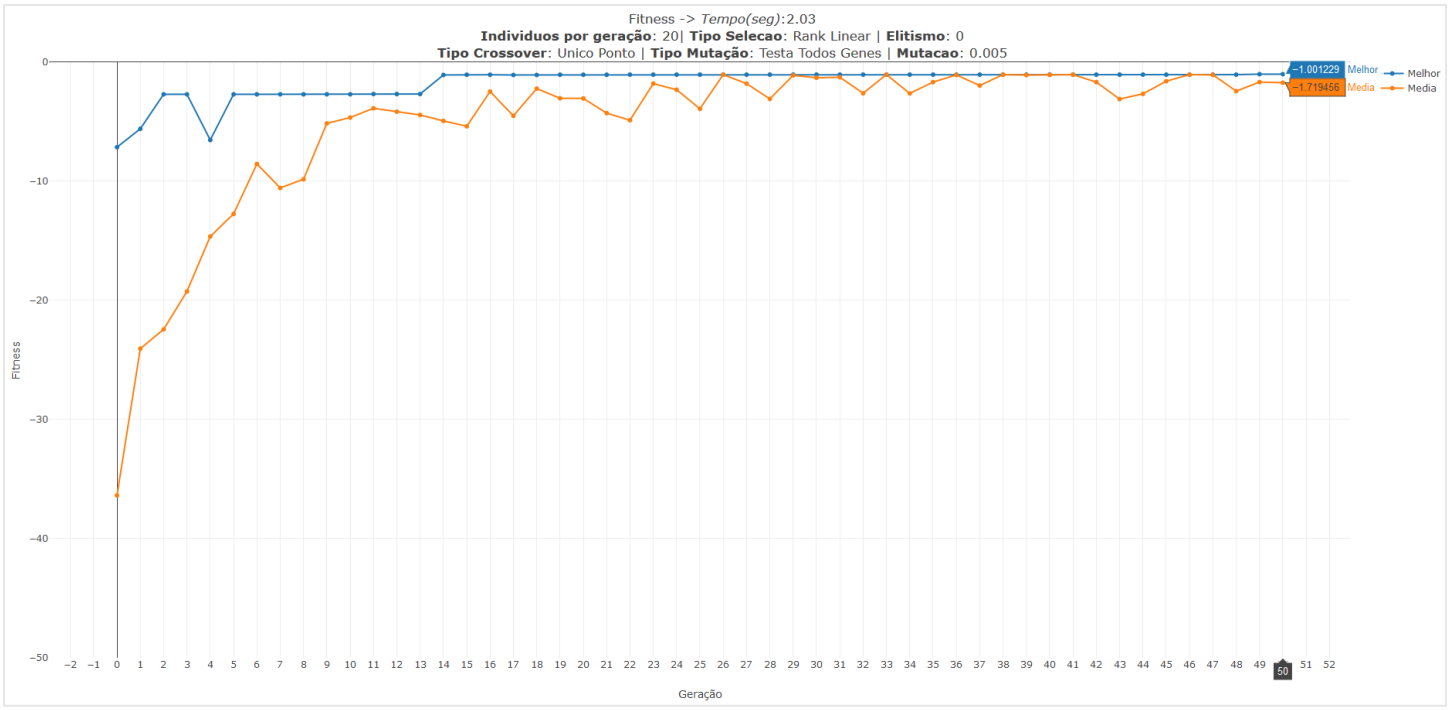
Torneio – Crossover único Ponto



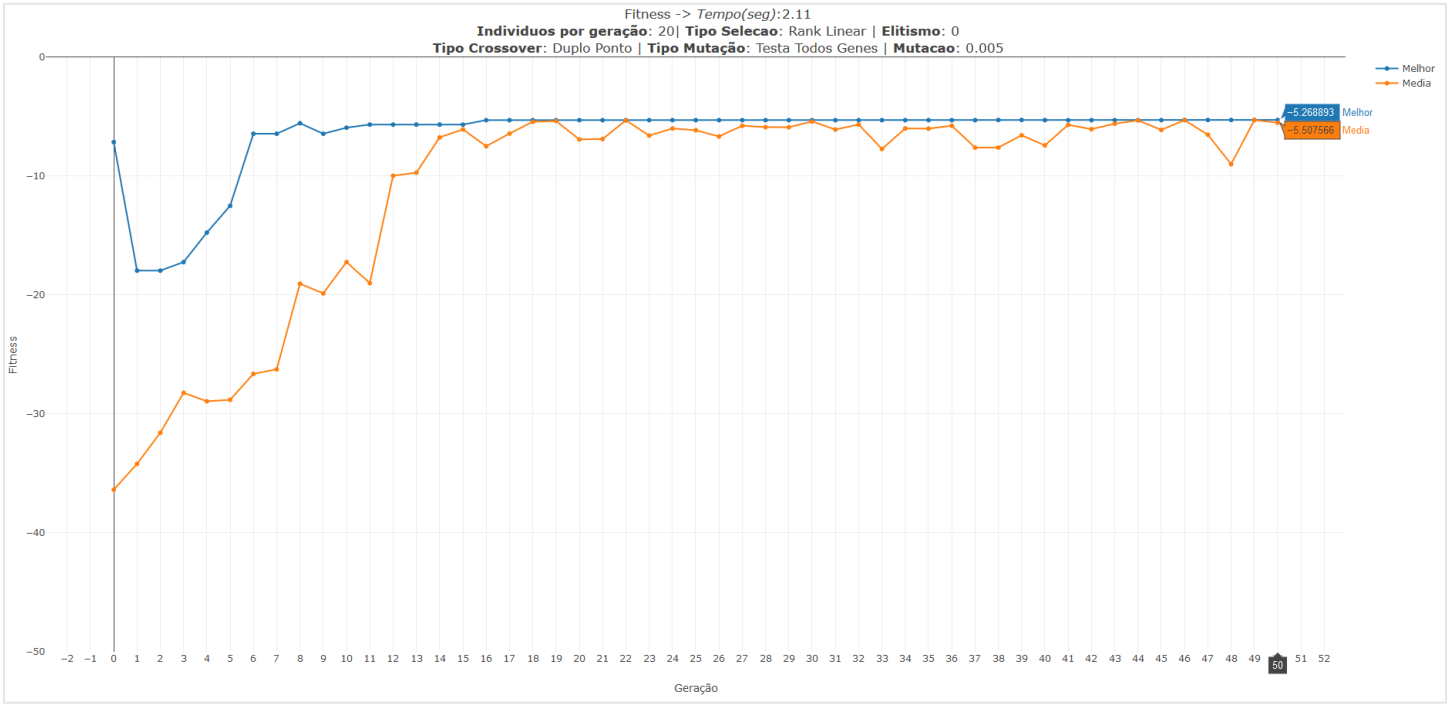
Torneio – Crossover duplo Ponto



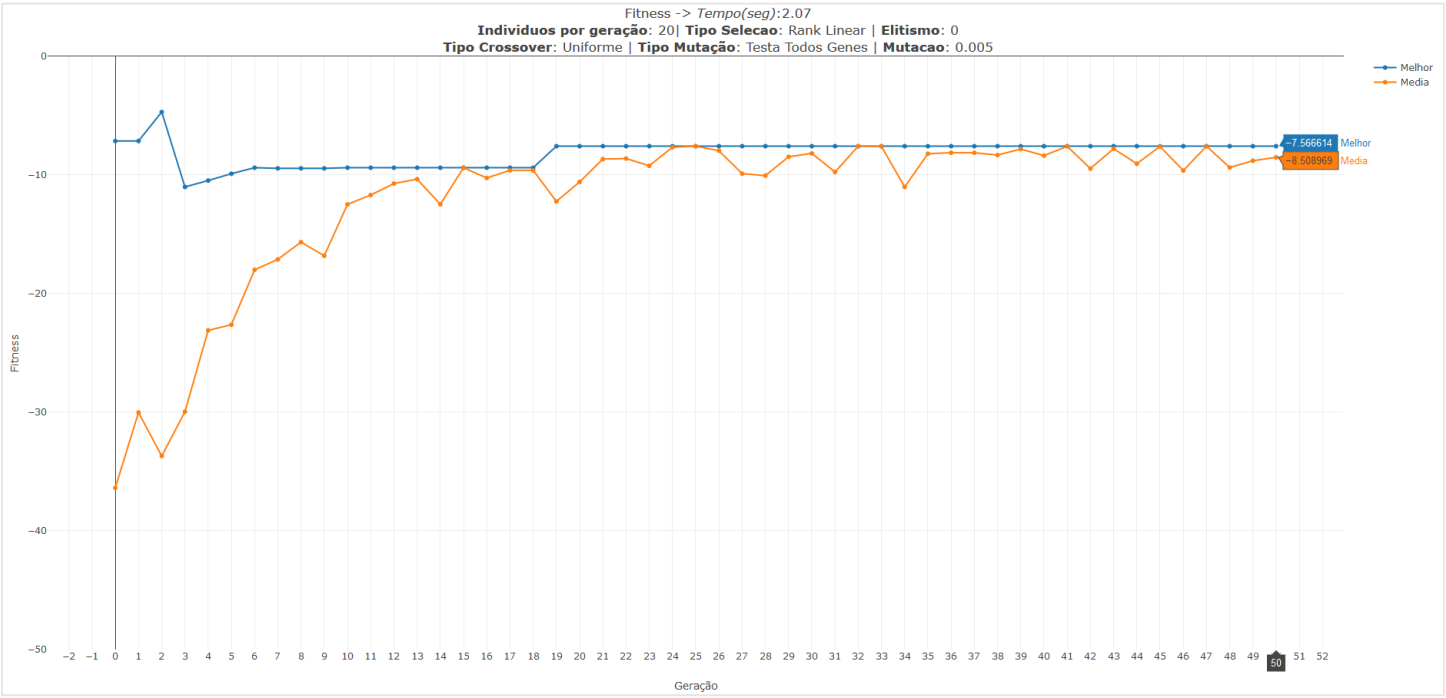
Torneio – Crossover uniforme



Rank Linear – Crossover único Ponto



Rank Linear – Crossover duplo Ponto



Rank Linear – Crossover uniforme

Abaixo, a tabela resumida com todas os resultados das execuções dos gráficos acima:

Seleção	Crossover	Mutação	Tempo	X	Y	Funcao	Melhor X	Melhor Y	Melhor Função/ Fitness	Geração Melhor Indivíduo
ROLETA	UNICO	TODOS	0.83	-1.950147	-0.99218	5.286154	-2.008798	-0.982405	5.0767104	13
ESTOCASTICA	UNICO	TODOS	2.25	-1.999022	0.004888	4.001018	0.962854	0.004888	1.2029548	12
TORNEIO	UNICO	TODOS	0.84	-0.933529	0.024438	1.849283	-0.933529	0.004888	1.735771	48
RANK	UNICO	TODOS	2.03	-0.99218	-0.004888	1.001229	-0.99218	-0.004888	1.0012288	49
ROLETA	DUPLO	TODOS	0.98	0.034213	-0.962854	1.429553	0.034213	-0.962854	1.4295526	50
ESTOCASTICA	DUPLO	TODOS	2.1	-1.930596	-0.024438	4.781294	-2.038123	-0.024438	4.5577126	27
TORNEIO	DUPLO	TODOS	0.89	0.004888	0.034213	0.236076	0.004888	0.014663	0.0473625	43
RANK	DUPLO	TODOS	2.11	-2.028348	0.99218	5.268893	-2.028348	0.99218	5.2688926	38
ROLETA	UNIFORME	TODOS	1.01	-1.989247	1.999022	7.976198	0.943304	-0.063539	2.3080565	2
ESTOCASTICA	UNIFORME	TODOS	2.18	1.01173	-0.043988	1.432204	1.01173	-0.004888	1.0554853	21
TORNEIO	UNIFORME	TODOS	1.12	0.024438	-0.043988	0.499708	0.004888	-0.043988	0.386196	26
RANK	UNIFORME	TODOS	2.07	1.871945	-0.99218	7.566614	-0.014663	2.047898	4.6859703	3

Além das configuraçoes acima, os AG foram executados com os seguintes parametros:

- Elitismo = 0
- Taxa de Crossover = 0.8
- Taxa de Mutação = 0.005
- Taxa de Seleção Melhor Torneio = 0.8

Ao analisar os resultamos e os gráficos acima, podemos perceber que nem sempre os melhores indivíduos são mantidos para as próximas gerações. Isso pode causar uma queda na probabildade de alcançarmos um resultado melhor, e uma das possibilidades de ajustarmos este problema é adicionando, por exemplo, uma taxa de Elitismo = 2 , onde a cada geração os 2 melhores indivíduos serão clonados para a próxima geração.

Podemos perceber também que em diversas execuções, principalmente as que utilizam o método de seleção da da roleta e do rank linear, estabilizaram nas gerações iniciais o "melhor" individuo, chegando a convergir com a média do fitness da população, o que pode indicar que um minimo local foi atingido. Para solucionarmos isto poderíamos experimentar desenvolver um algoritmo mais dinâmico que, ao perceber esta convergência e estabilidade por um determinado tempo e local, incluiria uma aumento momentâneo da taxa de mutação, para que houvesse uma probabilidade de que novas possibilidades e combinações de genes dos indivíduos fosse gerados (aumentando a diversidade), permitindo assim que este possível minimo local fosse superado.

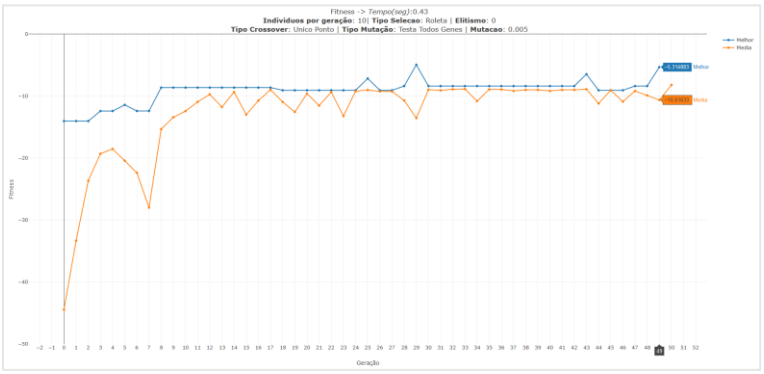
Como dito anteriormente que existem diversas combinações possíveis de execução para serem feitas com os parâmetros desenvolvidos, ao analisar as execuções acima percebemos que o tipo de seleção por Torneio, apesar de ainda não otimizado, atingiu melhores resultados no geral, se comparado aos outros tipos de seleção, tanto no valor do indivíduo final (Função = 0.23) quanto no valor do melhor indivíduo entre todas gerações (Função = 0.04), e que o tipo de seleção Roleta com corte de crossover em um ponto, foi um dos piores.

Dado este cenário, selecionei ambas as configurações citadas no parágrafo anterior e, a seguir, executarei elas com as seguintes combinações de parâmetros:

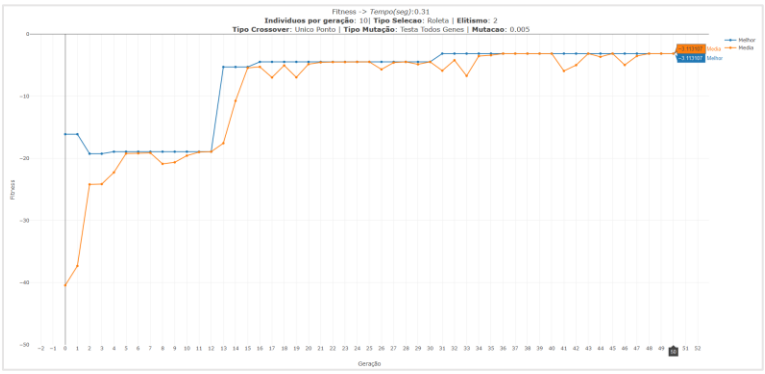
Gerações	Individuos	Seleção	Crossover
50	10	ROLETA	UNICO
50	20	ROLETA	UNICO
50	30	ROLETA	UNICO
50	40	ROLETA	UNICO
50	50	ROLETA	UNICO
50	60	ROLETA	UNICO
50	70	ROLETA	UNICO
50	80	ROLETA	UNICO
50	90	ROLETA	UNICO
50	100	ROLETA	UNICO
50	10	TORNEIO	DUPLO
50	20	TORNEIO	DUPLO
50	30	TORNEIO	DUPLO
50	40	TORNEIO	DUPLO
50	50	TORNEIO	DUPLO
50	60	TORNEIO	DUPLO
50	70	TORNEIO	DUPLO
50	80	TORNEIO	DUPLO
50	90	TORNEIO	DUPLO
50	100	TORNEIO	DUPLO

Abaixo seguem os resultados das execuções da tabela acima. Com a finalidade de adicionar mais um experimento, para cada item da tabela axima, també adicionei uma execução com o Elitismo adicionado ao algoritmo. Isto propiciou uma oportunidade de comparação entre a diferença da execução mantendo alguns dos melhores indivíduos para a próxima geração (com elitismo) e a execução sem o elitismo. Seguem abaixo os gráficos:
*(**obs**: os gráficos estão negativos pois, para manter o fitness maior sendo o mais importante, mesmo tentando minimizar a função, alteramos o sinal do resultado da função. Então, o valor mais perto do 0, mesmo que negativo, é o melhor fitness)*

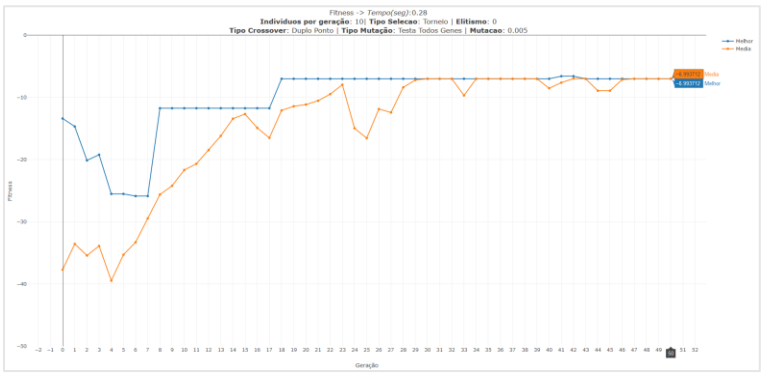
Experimento com População inicial = 10 indivíduos



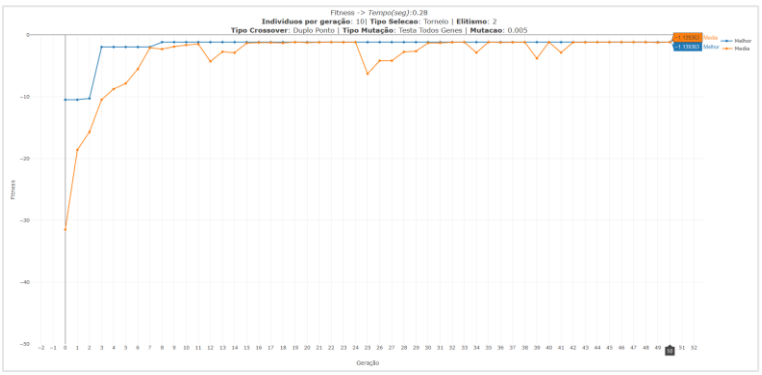
Roleta – Único Ponto – Sem Elitismo



Roleta – Único Ponto – Elitismo = 2

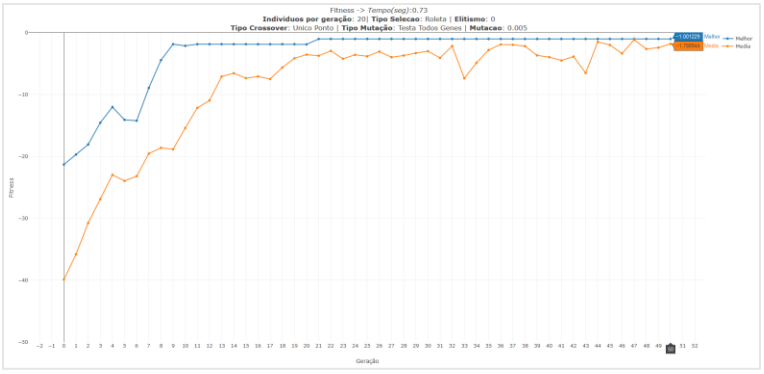


Torneio – Duplo Ponto – Sem Elitismo

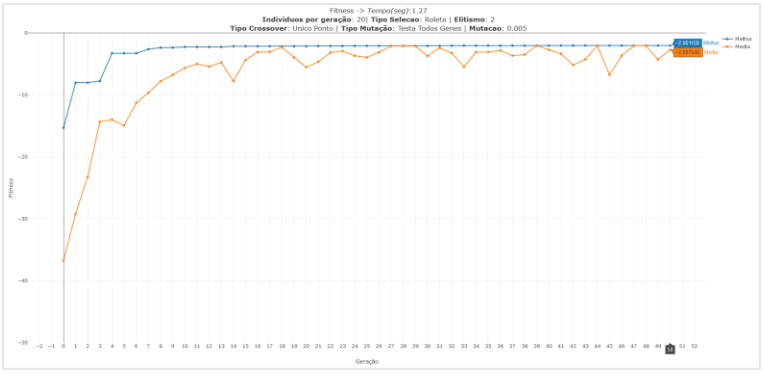


Torneio – Duplo Ponto – Elitismo = 2

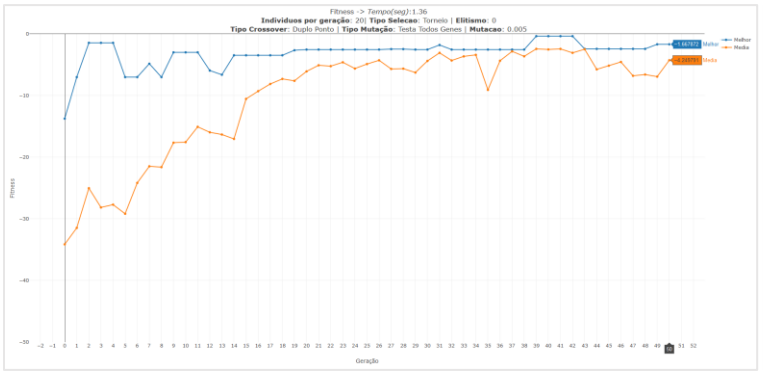
Experimento com População inicial = 20 indivíduos



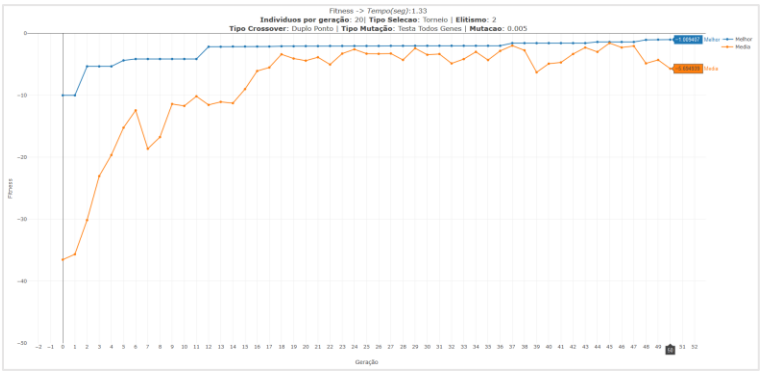
Roleta – Único Ponto – Sem Elitismo



Roleta – Único Ponto – Elitismo = 2

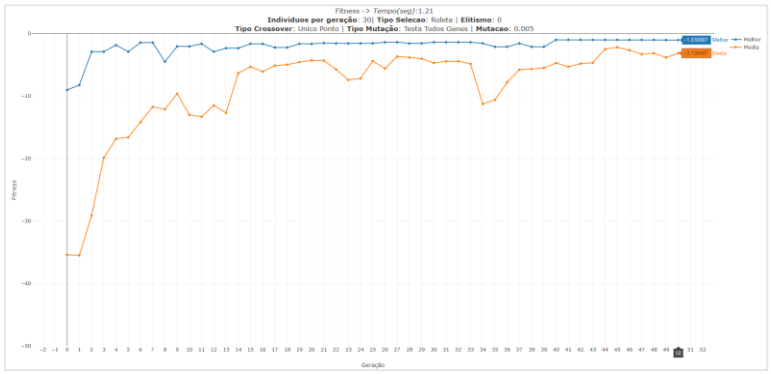


Torneio – Duplo Ponto – Sem Elitismo

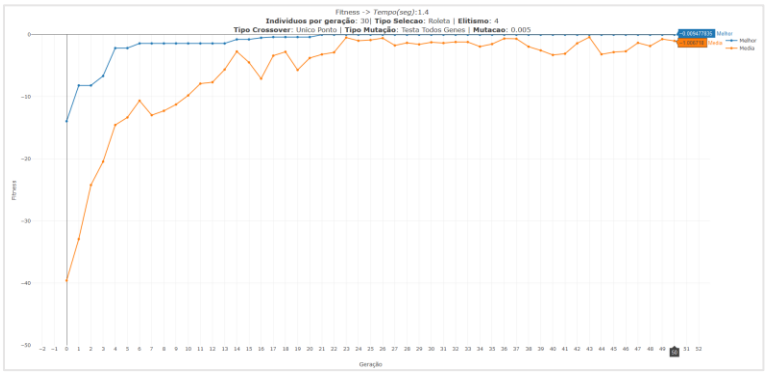


Torneio – Duplo Ponto – Elitismo = 2

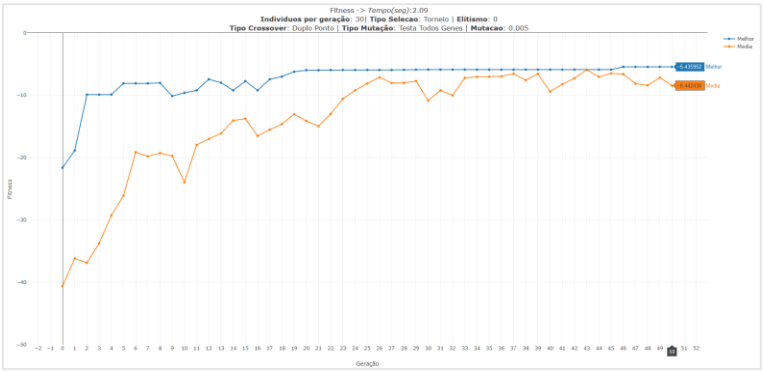
Experimento com População inicial = 30 indivíduos



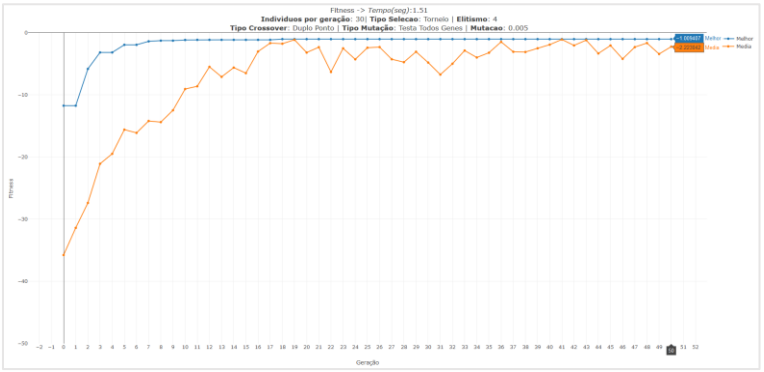
Roleta – Único Ponto – Sem Elitismo



Roleta – Único Ponto – Elitismo = 4

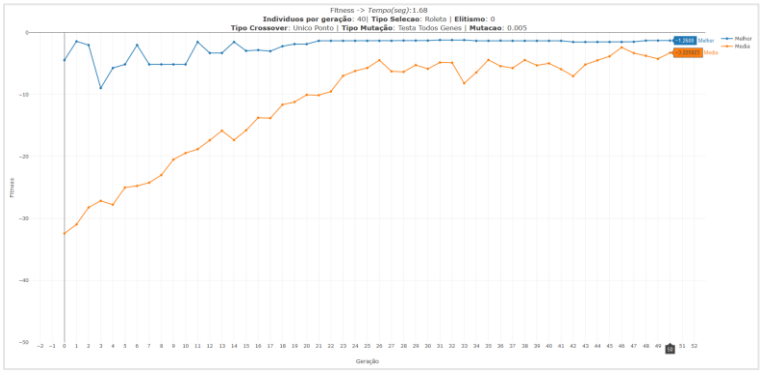


Torneio – Duplo Ponto – Sem Elitismo

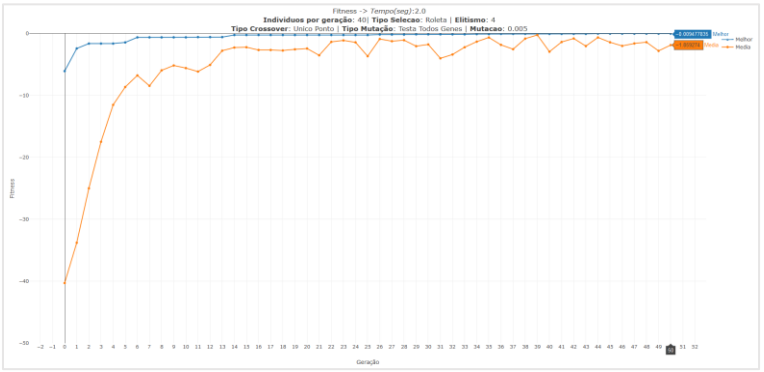


Torneio – Duplo Ponto – Elitismo = 4

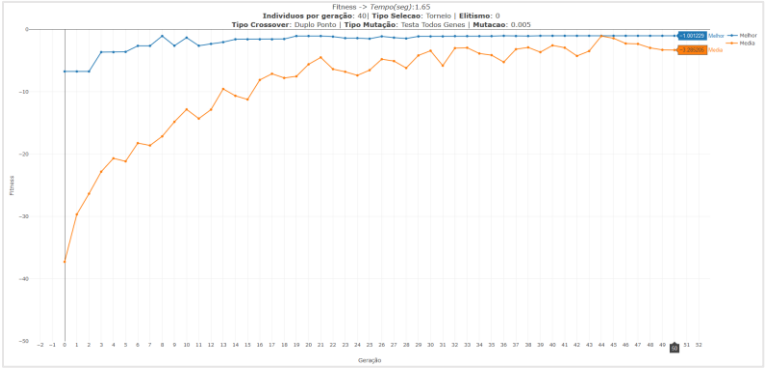
Experimento com População inicial = 40 indivíduos



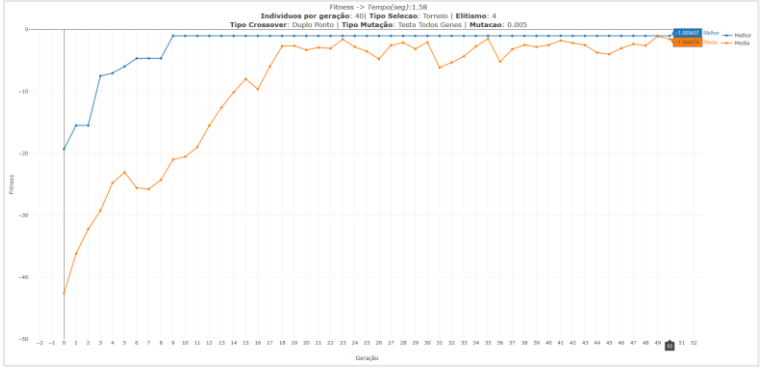
Roleta – Único Ponto – Sem Elitismo



Roleta – Único Ponto – Elitismo = 4

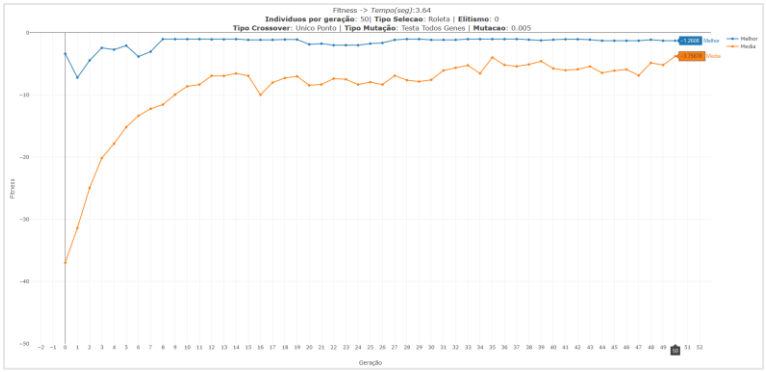


Torneio – Duplo Ponto – Sem Elitismo

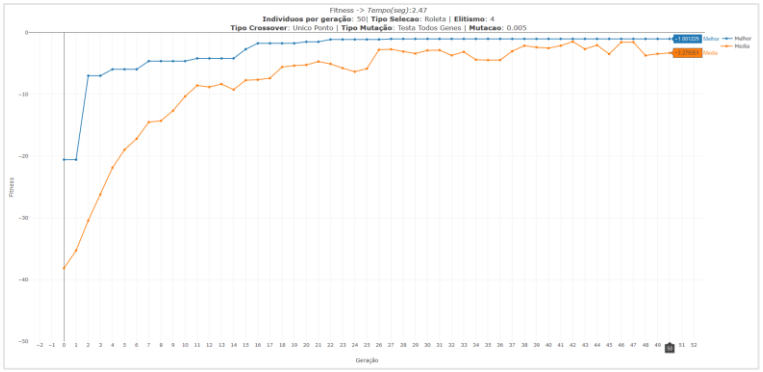


Torneio – Duplo Ponto – Elitismo = 4

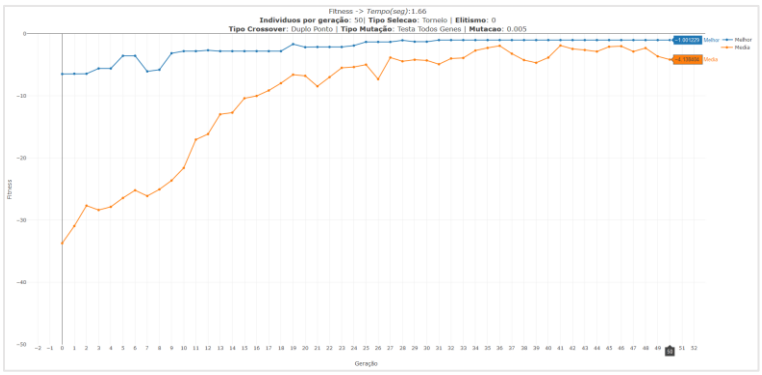
Experimento com População inicial = 50 indivíduos



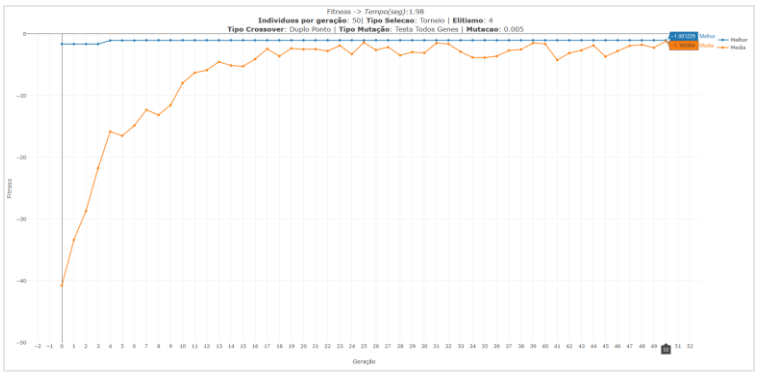
Roleta – Único Ponto – Sem Elitismo



Roleta – Único Ponto – Elitismo = 4

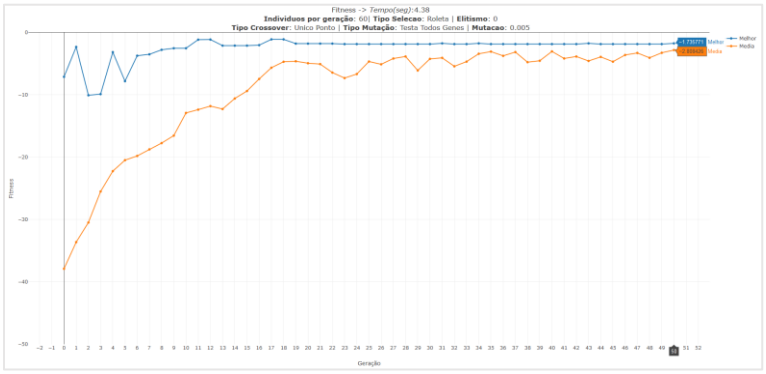


Torneio – Duplo Ponto – Sem Elitismo

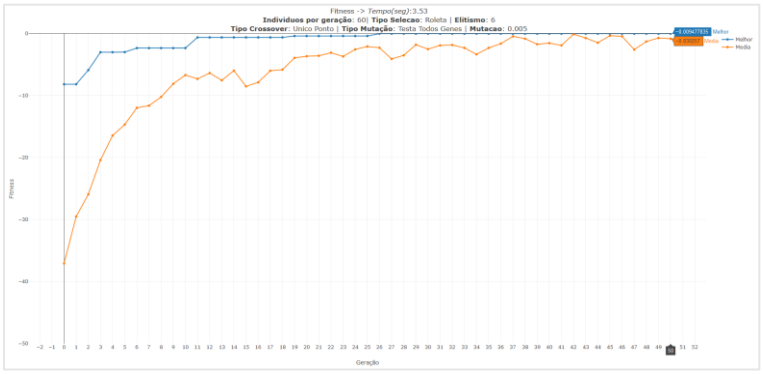


Torneio – Duplo Ponto – Elitismo = 4

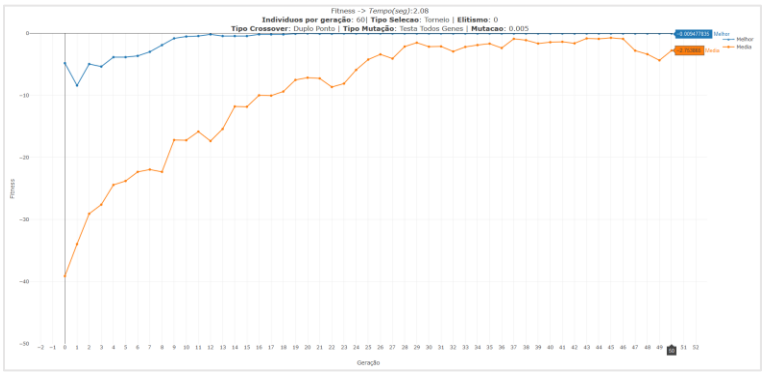
Experimento com População inicial = 60 indivíduos



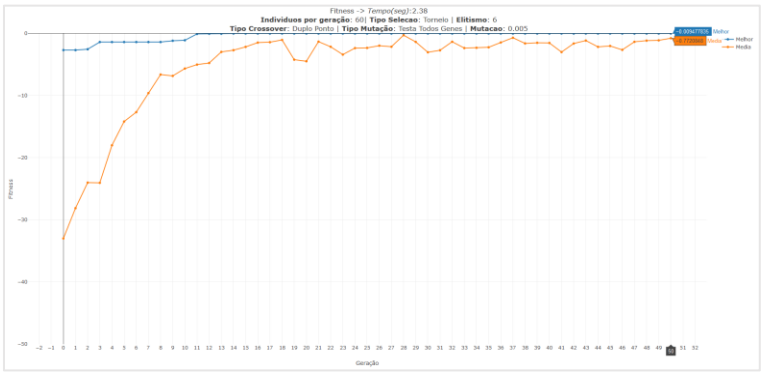
Roleta – Único Ponto – Sem Elitismo



Roleta – Único Ponto – Elitismo = 6

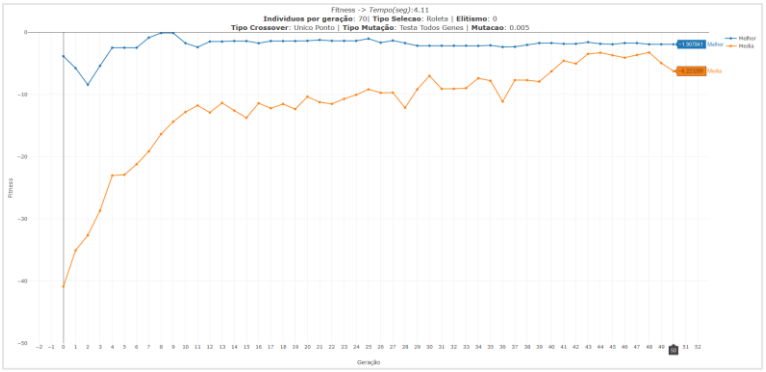


Torneio – Duplo Ponto – Sem Elitismo

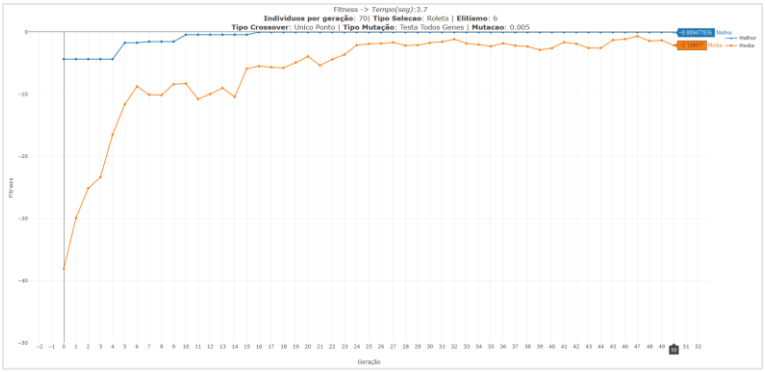


Torneio – Duplo Ponto – Elitismo = 6

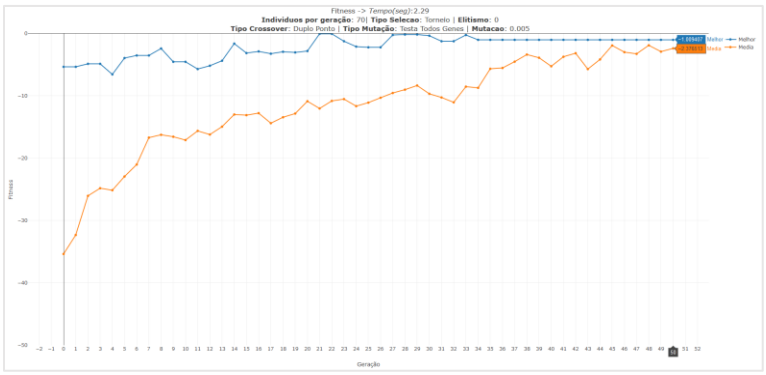
Experimento com População inicial = 70 indivíduos



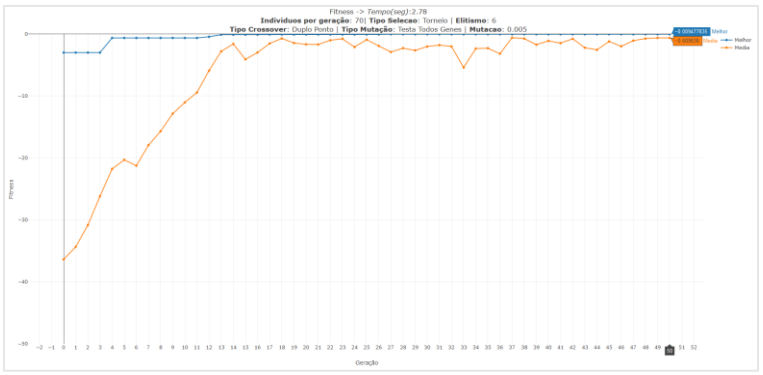
Roleta – Único Ponto – Sem Elitismo



Roleta – Único Ponto – Elitismo = 6

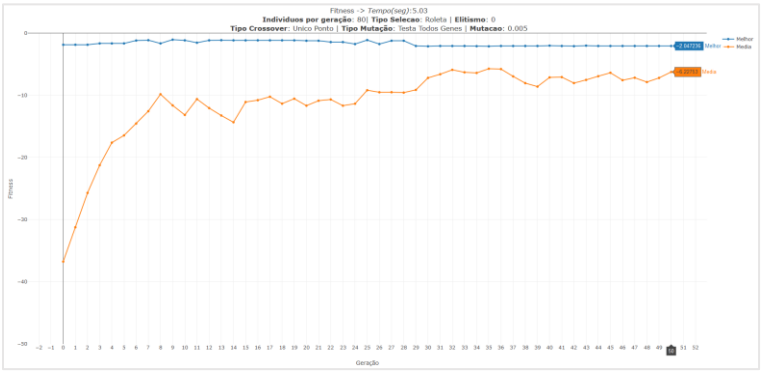


Torneio – Duplo Ponto – Sem Elitismo

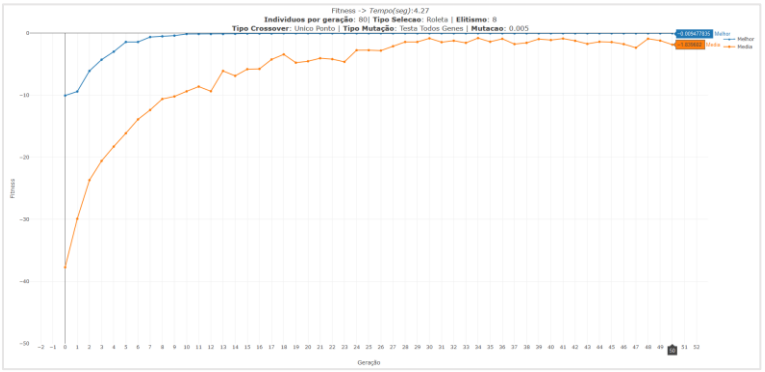


Torneio – Duplo Ponto – Elitismo = 6

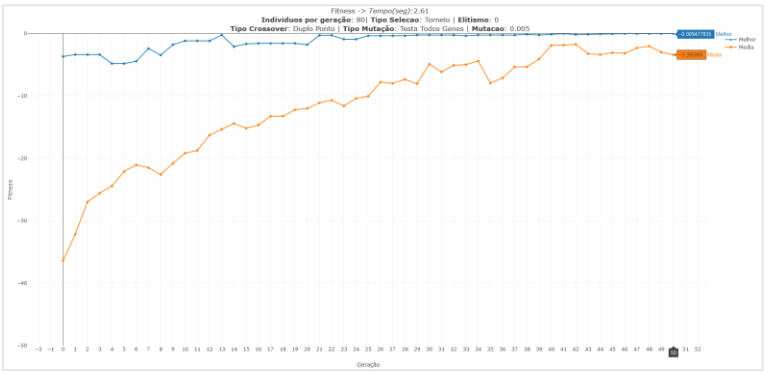
Experimento com População inicial = 80 indivíduos



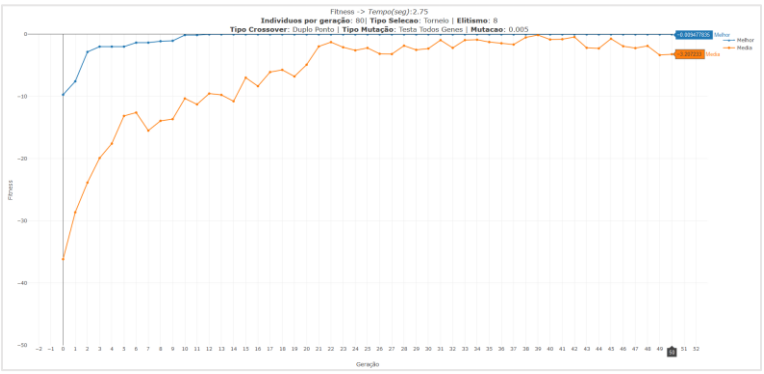
Roleta – Único Ponto – Sem Elitismo



Roleta – Único Ponto – Elitismo = 8

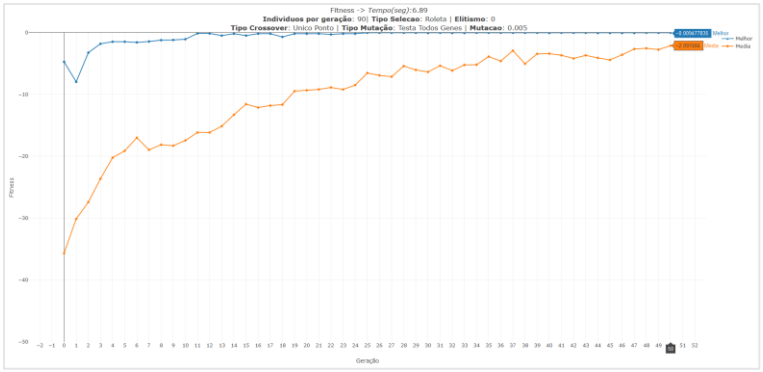


Torneio – Duplo Ponto – Sem Elitismo

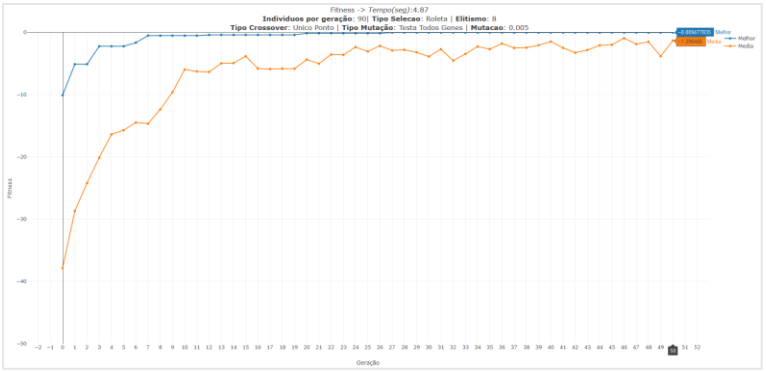


Torneio – Duplo Ponto – Elitismo = 8

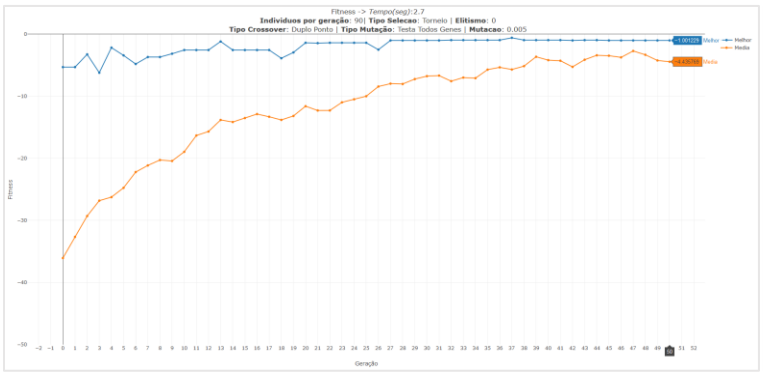
Experimento com População inicial = 90 indivíduos



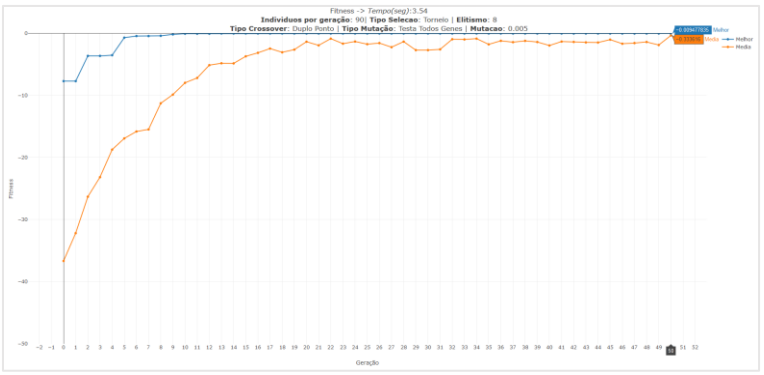
Roleta – Único Ponto – Sem Elitismo



Roleta – Único Ponto – Elitismo = 8

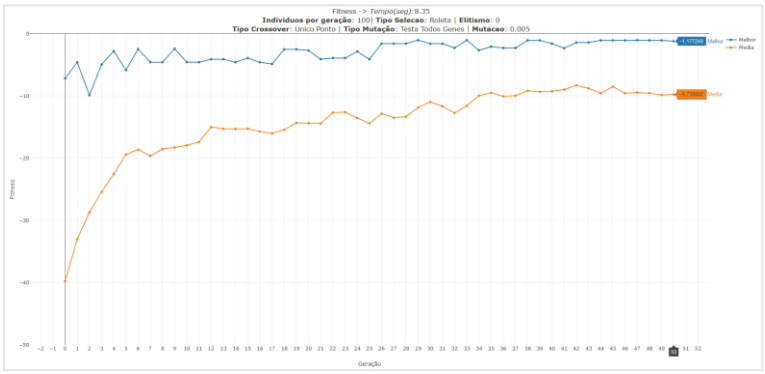


Torneio – Duplo Ponto – Sem Elitismo

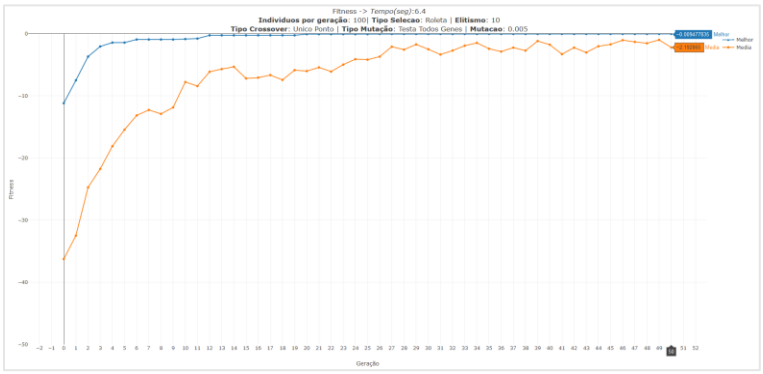


Torneio – Duplo Ponto – Elitismo = 8

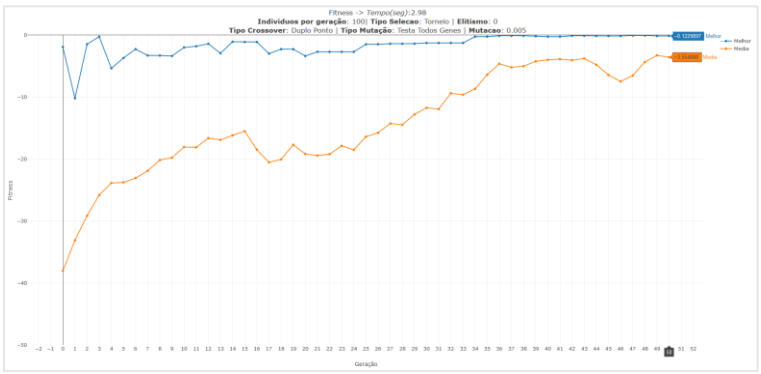
Experimento com População inicial = 100 indivíduos



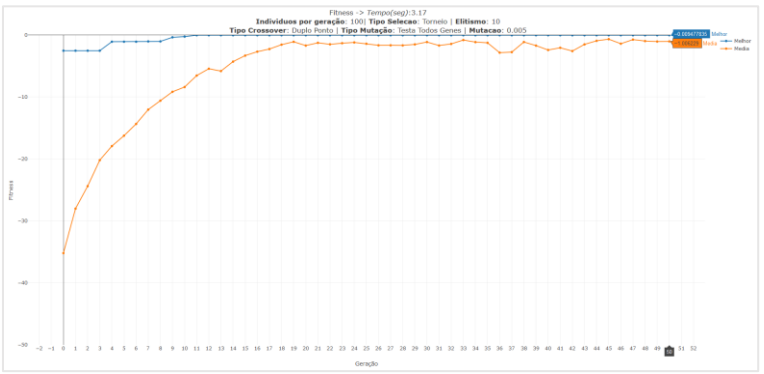
Roleta – Único Ponto – Sem Elitismo



Roleta – Único Ponto – Elitismo = 10



Torneio – Duplo Ponto – Sem Elitismo



Torneio – Duplo Ponto – Elitismo = 10

Após a execução e análise dos gráficos acima, pude concluir que:

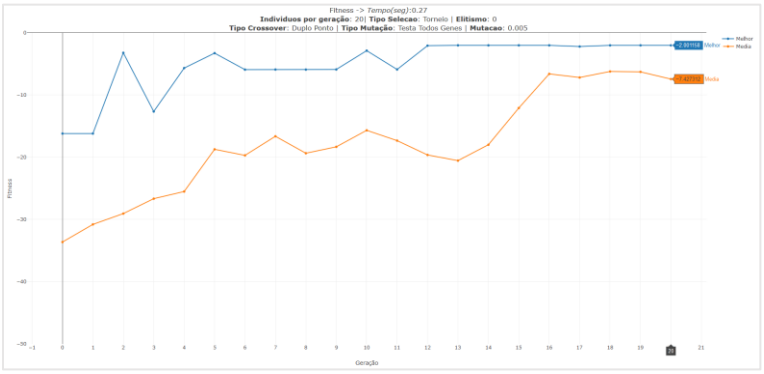
- Quanto maior a população inicial, probabilidade de um individuo perto do ótimo global “nascer” é maior, pois também essa população é gerada com mais diversidade
- Quanto maior a população, a probabilidade de ocorrer uma mutação é maior (pois mais indivíduos/genes são testados para a mutação), possibilitando um maior o aumento da diversidade
- É importante aplicarmos uma forma de mantermos os melhores indivíduos da população “vivos” para a próxima geração. Identificamos isso pois, nas execuções sem elitismos, em algumas gerações um bom indivíduo foi alcançado, mas não conseguiram passar seus “genes” de uma forma otimizada para as seguintes gerações. Isso fez com que, em diversas execuções, o melhor indivíduo da última geração não tenah sido o melhor indivíduo considerando todas as gerações. No nosso caso, procuramos usar uma média de Elitismo na faixa de 10%, o que nas nossa análises, fez com que as execuções com elitismo performassem acima das sem elitismo.
- Algoritmos Genéticos que iniciam com baixa população geralmente caem em um mínimo local em suas gerações iniciais. Isso também ocorre devido a baixa diversidade, e ao cair no minimo local, o melhor indivíduo estabiliza, juntamente com a média dos demais indivíduos da população a medida que as gerações passam. Ou seja, a população vai ficando mais homogênea
- A quantidade de gerações não influencia muito na qualidade da solução do problema, principalmente para populações pequenas, pois na maior parte das vezes, como citado no item anterior, após um mínimo local ser atingido e a diversidade da população com o passar das gerações é diminuir, a solução estabiliza e não obtem melhora
- Quanto menor a quantidade de população inicial, mais rápido um mínimo local (as vezes, com sorte, global) é atingido, não importando a quantidade de gerações.
- Quando foi usado elitismo, o percentual de atingimento do minimo global foi maior do que as vezes que o elitismo não foi usado. Isso parte do princípio que o melhor indivíduo tem maior probabilidade de passar seus genes para os demais ao longo das gerações e, mesmo que em um geração ele não crie um bom descendente, nas próximas ele novamente terá outras oportunidades, pois não existe a probabilidade do melhor indivíduo da população ser descartado

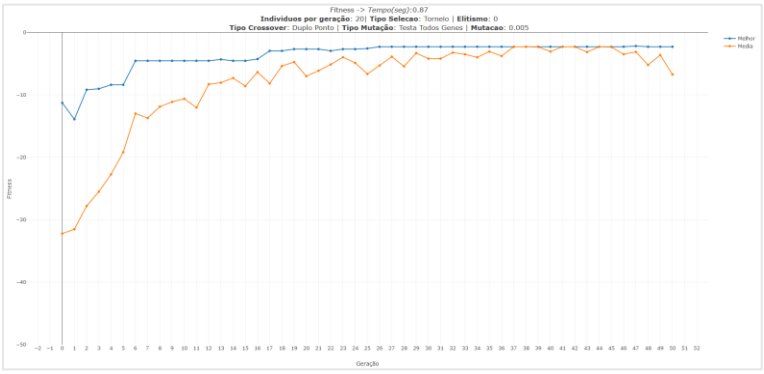
Abaixo, fiz umas execuções adicionais, para exibir que, mesmo com o aumento das gerações, não existe melhoria na qualidade para populações iniciais com poucos indivíduos e, para as que iniciam com alto número, a probabilidade de alcançar um minimo global é bem maior.

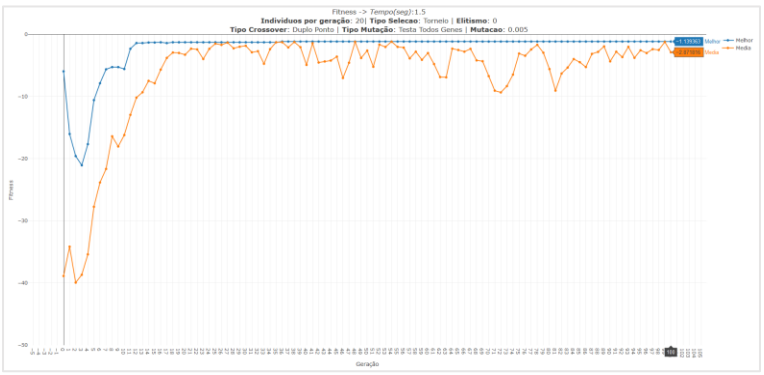
Esses são os cenários, e foram executados somente com o Torneio – Duplo Ponto:

Gerações	Individuos
20	20
20	50
20	100
50	20
50	50
50	100
100	20
100	50
100	100

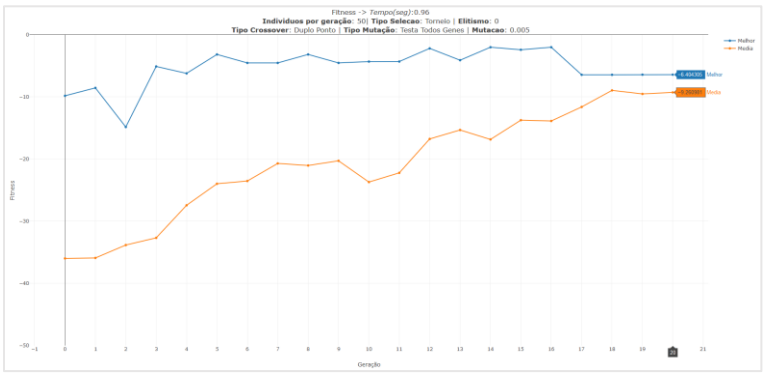
Experimento com População inicial = 20 indivíduos e Gerações (20, 50 e 100)

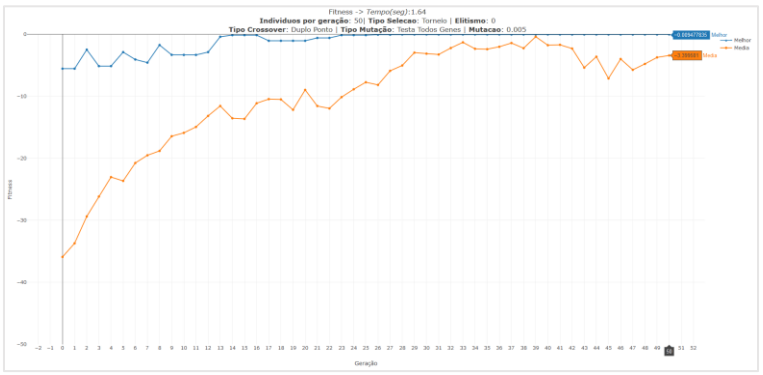


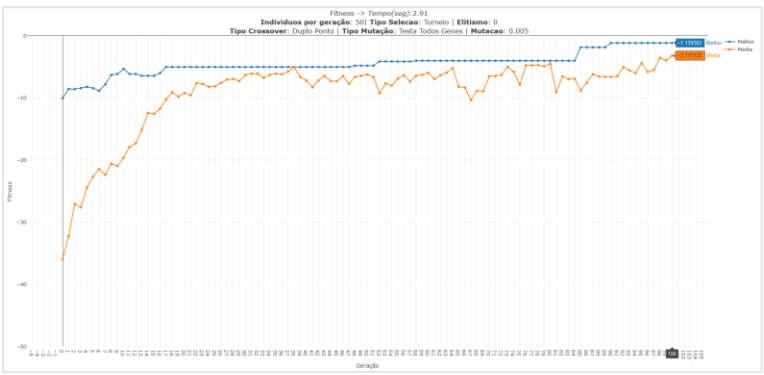




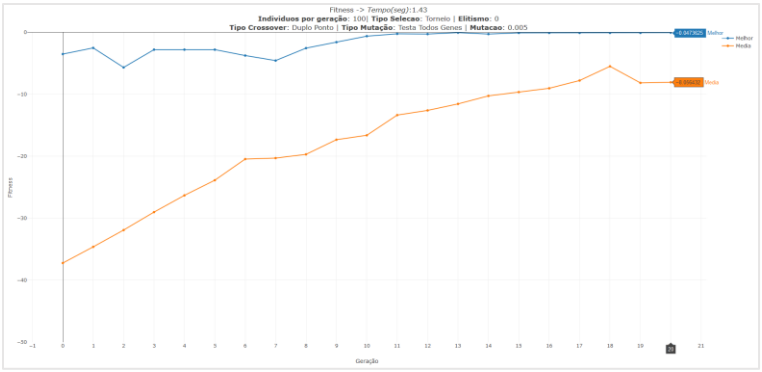
Experimento com População inicial = 50 indivíduos e Gerações (20, 50 e 100)

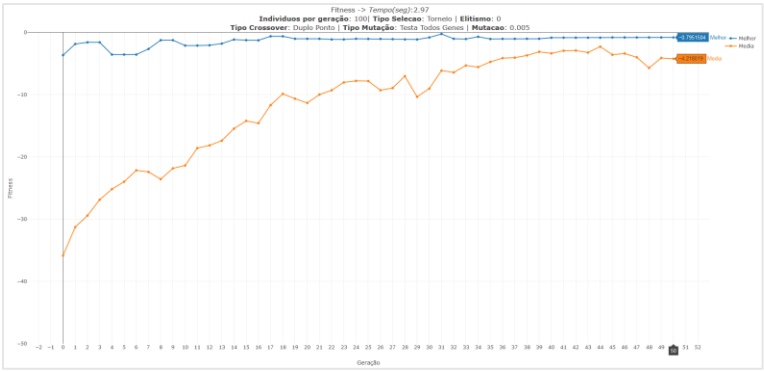


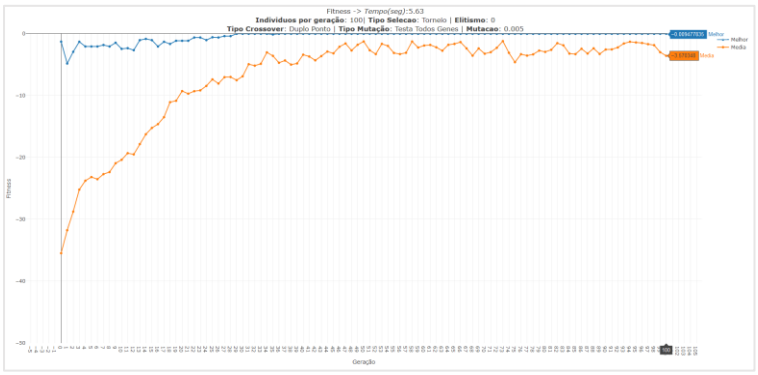




Experimento com População inicial = 100 indivíduos e Gerações (20, 50 e 100)







Nas execuções acima, pudemos confirmar que:

- Populações iniciais baixas convergem para o mínimo local mais próximo da população, geralmente entre as gerações 10/20, isso para populações de 20 indivíduos. Ou seja, mesmo com uma grande quantidade de gerações, o resultado não apresenta melhora.
- As execuções com 50 indivíduos por população tiveram uma melhor performance que os de 20, principalmente por iniciarem com uma diversidade maior. Isso permite que os individuos se complementem no crossover/seleção, aumentando a probabilidade de alcançarem um minimo global, e diminuindo a chance de permanecerem em um minimo local. Foi possível perceber que as estas execuções com 50 individuos de certa forma ficaram mais estáveis a partir da geração 20/25.
- Para as execuções de 100 indivíduos com a população inicial, todas as 3 execuções alcançaram o minimo global em algum momento, sendo 2 finalizando com o minimo global e, umas das execuções (a de 50 gerações), atingiu o minimo global na geração 31 mas, devido a não ter o elitismo aplicado, este indivíduo for descartado não sendo alcançado novamente até o final da execução. Com isso, concluimos que a diversidade é extremamente importante para que o Algoritmo Genético não fique preso em um minimo local, principalmente nas gerações iniciais (mas não há garantia), e também que é importante tentarmos preservar uma quantidade de melhores indivíduos por geração, para que ele não seja descartado, tendo o cuidado de não transformar a população em uma população homogênea.

Conclusão

Com os experimentos acima pudemos ter uma análise bem produtiva de como funciona o algoritmo genético e como os parâmetros possíveis podem influenciar na qualidade dos resultados. Foi possíve perceber que a diversidade é um fato muito importante, o qual se deve implementar formas de preservá-la (e em alguns cenários aumentá-la) para que seja miminizada a probabilidade de uma execução cacir em um minimo local. A quantidade de indivíduos por população também é bem importante, o que fazer com que o aumento de gerações não impacte na qualidade do resultado para pequenas populações.

Código desenvolvido para os experimentos acima (também está no repositório)

```
import numpy as np
import random
from operator import attrgetter
import math as math
import time as time
from copy import deepcopy
from enum import Enum


class TIPO_SELECAO(Enum):
    ROLETA = 1
    AMOSTRAGEM_ESTOCASTICA = 2
    TORNEIO = 3
    RANK_LINEAR = 4

class TIPO_CROSSOVER(Enum):
    UNICO_PONTO = 1
    DUPLO_PONTO = 2
    UNIFORME = 3
    ARITMETICO = 4

class TIPO_MUTACAO(Enum):
    GENE_UNICO = 1 # Se houver, escolhe 1 gene e efetua a mutação
    N_GENES = 2 # Se houver, escolhe N genes e efetua a mutação
    TODOS_GENES = 3 # Percorre cada gene e, se houver mutaçãp, efetua a mutação neste gene

class Individuo():
    def __init__(self, qtde_bits_por_entrada, geracao=0):
        self.xy = np.random.randint(2, size=qtde_bits_por_entrada*2)
        self.x = self.xy[0:qtde_bits_por_entrada]
        self.y = self.xy[qtde_bits_por_entrada::]
        self.fitness = -10000
        self.geracao = geracao
        self.qtde_bits_por_entrada = qtde_bits_por_entrada

    def getX(self):
        return self.xy[0:self.qtde_bits_por_entrada]

    def getY(self):
        return self.xy[self.qtde_bits_por_entrada::]

    def atualizarXY(self):
        self.x = self.xy[0:self.qtde_bits_por_entrada]
        self.y = self.xy[self.qtde_bits_por_entrada::]

class AlgoritmoGenetico():

    def __init__(self,
```

```
                    limite_ini,
                    limite_fim,
                    geracoes,
                    individuos_por_geracao,
                    qtde_bits_por_entrada,
                    tipo_selecao=TIPO_SELECAO.ROLETA, # Qual o tipo de seleção aplicada
                    qtde_individuo_elitismo=0,
                    crossover_por_entrada=False, # True ou False. Indica se o crossover será feito
por entrada (ex.: POR ENTRADA => pai1.x com pai2.x e pai1.y com pai2.y) ou no cromossomo inteiro
(ex.: INTEIRO => pai1.xy com pai2.xy)
                    tipo_crossover=TIPO_CROSSOVER.UNICO_PONTO,
                    taxa_crossover=0.8,
                    taxa_selecao_melhor_torneio=0.8,
                    tipo_mutacao=TIPO_MUTACAO.GENE_UNICO,
                    taxa_mutacao=0.01,
                    populacao_inicial = []):
        self.q = self.calcular_q(limite_ini, limite_fim, qtde_bits_por_entrada)
        self.individuos_por_geracao = individuos_por_geracao
        self.tipo_selecao = tipo_selecao
        self.qtde_individuo_elitismo = qtde_individuo_elitismo
        self.crossover_por_entrada = crossover_por_entrada
        self.tipo_crossover = tipo_crossover
        self.taxa_crossover = taxa_crossover
        self.geracoes = geracoes
        self.qtde_bits = qtde_bits_por_entrada
        self.numero_geracao = 1
        self.tipo_mutacao = tipo_mutacao
        self.taxa_mutacao = taxa_mutacao
        self.taxa_selecao_melhor = taxa_selecao_melhor_torneio
        self.fitnesses = []
        self.media_fitnesses = []
        self.populacao_inicial = populacao_inicial
        # Coluna 1: x1
        # Coluna 2: x2
        # Coluna 3: fitness do Individuo
        self.populacao = []
        self.melhor_individuo = Individuo(qtde_bits_por_entrada)

    def calcular_q(self, min_funcao, max_funcao, qtde_bits_por_entrada):
        self.q = (max_funcao - min_funcao) / (2**qtde_bits_por_entrada - 1)
        self.ajuste = min_funcao
        return self.q

    def calcular_valor_decimal_q(self, binario):
        valor = 0
        exp = 0
        for i in reversed(range(len(binario))):
            valor = valor + (binario[i] * 2**exp) * self.q
            exp = exp + 1
        return valor + self.ajuste

    # Gera a população inicial, de forma aleatória, com a quantidade de individuos pré-definida
    def gerar_populacao_inicial(self):
```

```
        if len(self.populacao_inicial) <= 0:
            for i in range(self.individuos_por_geracao):
                self.populacao.append(Individuo(self.qtde_bits, self.numero_geracao))
        else:
            for i in range(self.individuos_por_geracao):
                self.populacao.append(self.populacao_inicial[i])

    # Avalia a população, gerando o fitness
    def avaliar_populacao(self):
        for i in range(self.individuos_por_geracao):
            self.populacao[i].fitness = self.calcular_funcao(self.populacao[i])

    # Calculo da função que se deseja minimizar
    def calcular_funcao(self, individuo):
        x = self.calcular_valor_decimal_q(individuo.getX())
        y = self.calcular_valor_decimal_q(individuo.getY())
        # valor_funcao = 3*x1**2 + 2*x2
        # valor_funcao = math.sin(x1) + math.cos(x2)
        # valor_funcao = (x1 - 1)**2 + (x2 - 2)**2 - 1
        # rastring
        valor_funcao = 20 + x**2 + y**2 -10*(math.cos(2*math.pi*x) + math.cos(2*math.pi*y))

        return -valor_funcao # Coloca o negativo para "inverter" o sinal, pois queremos o minimizar
a função

    # Ordena a lista de Individuos de acordo com o fitness, do maior para o menor
    def ordenar_populacao(self):
        self.populacao = sorted(self.populacao,
                                key = lambda populacao: populacao.fitness,
                                reverse = True)

        self.fitnesses.append(self.populacao[0].fitness)
        media = np.mean([c.fitness for c in self.populacao])
        self.media_fitnesses.append(media)
        if self.populacao[0].fitness > self.melhor_individuo.fitness:
            self.melhor_individuo = self.populacao[0]

    # Imprime o melhor e pior indivíduo da geração
    def imprime_melhor_pior_da_geracao(self):
        melhor_fitness = self.populacao[0].fitness
        pior_fitness = self.populacao[-1].fitness
        x = self.calcular_valor_decimal_q(self.populacao[0].getX())
        y = self.calcular_valor_decimal_q(self.populacao[0].getY())
        x_pior = self.calcular_valor_decimal_q(self.populacao[-1].getX())
        y_pior = self.calcular_valor_decimal_q(self.populacao[-1].getY())

        if (self.numero_geracao >= self.geracoes):
            print("\nGERAÇÃO %s" % (self.numero_geracao))
            print("Melhor: Fitness: %s x: %s y: %s" % (melhor_fitness,
                                                       x,
                                                       y))
            print("Pior: Fitness: %s x: %s y: %s" % (pior_fitness,
                                                     x_pior,
```

```
                                                                y_pior))

    # soma do fitness para utilizar na selção por roleta/amostragem estática
    def soma_total_fitness(self):
        soma = 0
        valor_ajuste = 0
        # Caso tenha numero negativo, ajusta para a soma ser a partir do 1, que para que seja
possível efetuar a Roleta
        # É somado +1 para que, mesmo o último colocado, tenha chance de ser selecionado na roleta
(apesar se ser pequena).
        # Resumindo, estamos ajustando o eixo dos valores para a partir de 0. Isso só é utilizado
se a Roleta for utilizada para seleção
        if(self.populacao[-1] < 0):
            valor_ajuste = -self.populacao[-1].fitness + 1

        for individuo in self.populacao:
            soma += individuo.fitness + valor_ajuste
        return soma


    # Seleciona os pais para o crossover
    def selecionar_pais(self):
        if self.tipo_selecao == TIPO_SELECAO.ROLETA:
            return self.seleciona_pai_roleta()
        elif self.tipo_selecao == TIPO_SELECAO.AMOSTRAGEM_ESTOCASTICA:
            return self.seleciona_pai_amostragem_estocastica()
        elif self.tipo_selecao == TIPO_SELECAO.TORNEIO:
            return self.seleciona_pai_torneio()
        elif self.tipo_selecao == TIPO_SELECAO.RANK_LINEAR:
            return self.seleciona_pai_rank_linear()

    def calcular_soma_fitness(self, populacao_soma):
        soma_avaliacao = 0
        valor_ajuste = 0
        # Caso tenha numero negativo, ajusta para a soma ser a partir do 1, que para que seja
possível efetuar a Roleta
        # É somado +1 para que, mesmo o último colocado, tenha chance de ser selecionado na roleta
(apesar se ser pequena).
        # Resumindo, estamos ajustando o eixo dos valores para a partir de 0. Isso só é utilizado
se a Roleta for utilizada para seleção
        if(populacao_soma[-1].fitness < 0):
            valor_ajuste = -populacao_soma[-1].fitness + 1

        for individuo in populacao_soma:
            soma_avaliacao += individuo.fitness + valor_ajuste

        return soma_avaliacao, valor_ajuste

    # Efetua seleção dos pais via Roleta
    def seleciona_pai_roleta(self):
        soma_avaliacao, valor_ajuste = self.calcular_soma_fitness(self.populacao)

        # Seleciona o PAI 1
```

```
        pai1 = -1
        valor_sorteado = random.random() * soma_avaliacao
        soma = 0
        i = 0
        while i < len(self.populacao) and soma < valor_sorteado:
            soma += self.populacao[i].fitness + valor_ajuste
            pai1 += 1
            i += 1

        # Seleciona o PAI 2
        pai2 = -1
        valor_sorteado = random.random() * soma_avaliacao
        soma = 0
        i = 0
        while i < len(self.populacao) and soma < valor_sorteado:
            soma += self.populacao[i].fitness + valor_ajuste
            pai2 += 1
            i += 1

        # Retorna os pais selecionados pela roleta
        return self.populacao[pai1], self.populacao[pai2]


    def seleciona_pai_amostragem_estocastica(self):
        soma_avaliacao, valor_ajuste = self.calcular_soma_fitness(self.populacao)
        n = 2 # Quantidade de individuos selecionados
        # Calculo a média
        media = soma_avaliacao / n
        alfa = random.random()
        delta = alfa * media

        populacao_embaralhada = deepcopy(self.populacao)
        random.shuffle(populacao_embaralhada)

        i = 0
        j = 0
        soma = populacao_embaralhada[0].fitness + valor_ajuste
        index_pais = []
        while i < n:
            if (delta < soma):
                index_pais.append(j)
                delta = delta + media
                i = i + 1
            else:
                j = j + 1
                soma = soma + populacao_embaralhada[j].fitness + valor_ajuste

        pai1 = populacao_embaralhada[index_pais[0]]
        pai2 = populacao_embaralhada[index_pais[1]]

        return pai1, pai2
```

```
    # Como existem fitness negativos, o Torneio pode ser realizado.
    # A seleção normal, que usa a somatória, não pode ser usada sem ter o fitness tratado(faço
isso). O torneio pode.
    def seleciona_pai_torneio(self):
        # Selecionando PAI 1
        # sorteia 2 individuos da população
        pais_1 = random.sample(self.populacao, 2)
        r = random.uniform(0, 1)

        # Se r for menor que a taxa de seleção, o melhor fitness é selecionado
        if r <= self.taxa_selecao_melhor:
            pai1 = max(pais_1, key=attrgetter('fitness'))
        else:
            pai1 = min(pais_1, key=attrgetter('fitness'))

        ######################################################

        # Selecionando PAI 2
        # sorteia 2 individuos da população
        pais_2 = random.sample(self.populacao, 2)
        r = random.uniform(0, 1)

        # Se r for menor que a taxa de seleção, o melhor fitness é selecionado
        if r <= self.taxa_selecao_melhor:
            pai2 = max(pais_2, key=attrgetter('fitness'))
        else:
            pai2 = min(pais_2, key=attrgetter('fitness'))

        # Retorna os pais selecionados através do torneio
        return pai1, pai2

    # Efetua seleção dos pais via Roleta
    def seleciona_pai_rank_linear(self):
        populacao_rank_linear = deepcopy(self.populacao)
        fitness_linear = len(populacao_rank_linear)
        for i in range(len(populacao_rank_linear)):
            populacao_rank_linear[i].fitness = fitness_linear
            fitness_linear = fitness_linear - 1

        soma_avaliacao, valor_ajuste = self.calcular_soma_fitness(populacao_rank_linear)

        # Seleciona o PAI 1
        pai1 = -1
        valor_sorteado = random.random() * soma_avaliacao
        soma = 0
        i = 0
        while i < len(populacao_rank_linear) and soma < valor_sorteado:
            soma += populacao_rank_linear[i].fitness + valor_ajuste
            pai1 += 1
            i += 1

        # Seleciona o PAI 2
        pai2 = -1
```

```
        valor_sorteado = random.random() * soma_avaliacao
        soma = 0
        i = 0
        while i < len(populacao_rank_linear) and soma < valor_sorteado:
            soma += populacao_rank_linear[i].fitness + valor_ajuste
            pai2 += 1
            i += 1

        # Retorna os pais selecionados pela roleta
        return populacao_rank_linear[pai1], populacao_rank_linear[pai2]

    # Efetua o crossover de 2 pais, gerando 2 filhos
    def efetuar_crossover(self, pai1, pai2):
        # Se for menor que a taxa de crossover, efetua o crossover
        if random.random() < self.taxa_crossover:
            if self.tipo_crossover == TIPO_CROSSOVER.UNICO_PONTO:
                return self.efetuar_crossover_unico_ponto(pai1, pai2)
            elif self.tipo_crossover == TIPO_CROSSOVER.DUPLO_PONTO:
                return self.efetuar_crossover_duplo_ponto(pai1, pai2)
            elif self.tipo_crossover == TIPO_CROSSOVER.UNIFORME:
                return self.efetuar_crossover_uniforme(pai1, pai2)
            elif self.tipo_crossover == TIPO_CROSSOVER.ARITMETICO:
                return ""
        else: # se não, retorna o próprio pai1 e pai2 (clone)
            return pai1, pai2


    # Crossover de Unico Ponto
    def efetuar_crossover_unico_ponto(self, pai1, pai2):
        if (self.crossover_por_entrada): # Crossover de X com X (0 a 10) e de Y com Y (10 a 20)
            # Calcula um local aleatório de corte
            total_bits_x = self.qtde_bits
            local_corte = round(random.random()  * total_bits_x)

            # Filho 1
            # X do filho 1
            filho1 = Individuo(self.qtde_bits, self.numero_geracao)
            x1 = []
            x1 = np.append(pai1.x[0:local_corte] , pai2.x[local_corte:: ])
            filho1.x = x1
            # Y do filho 1
            y1 = []
            y1 = np.append(pai1.y[0:local_corte] , pai2.y[local_corte:: ])
            filho1.y = y1
            filho1.xy = np.append(x1, y1)

            # Filho 2
            # X do filho 2
            filho2 = Individuo(self.qtde_bits, self.numero_geracao)
            x2 = []
            x2 = np.append(pai2.x[0:local_corte] , pai1.x[local_corte:: ])
            filho2.x = x2
            # Y do filho 2
```

```
            y2 = []
            y2 = np.append(pai2.y[0:local_corte] , pai1.y[local_corte:: ])
            filho2.y = y2
            filho2.xy = np.append(x2, y2)

        else: # Crossover em X e Y como um cromossomo s'o
            # Calcula um local aleatório de corte
            total_bits = len(pai1.xy)
            local_corte = round(random.random()  * total_bits)

            filho1 = Individuo(self.qtde_bits, self.numero_geracao)
            xy1 = []
            xy1 = np.append(pai1.xy[0:local_corte] , pai2.xy[local_corte:: ])
            filho1.xy = xy1
            filho1.x = xy1[0:self.qtde_bits]
            filho1.y = xy1[self.qtde_bits::]

            filho2 = Individuo(self.qtde_bits, self.numero_geracao)
            xy2 = []
            xy2 = np.append(pai2.xy[0:local_corte] , pai1.xy[local_corte:: ])
            filho2.xy = xy2
            filho2.x = xy2[0:self.qtde_bits]
            filho2.y = xy2[self.qtde_bits::]

        novos_filhos = []
        novos_filhos.append(filho1)
        novos_filhos.append(filho2)

        return novos_filhos

    # Efetua o crossover de duplo ponto
    def efetuar_crossover_duplo_ponto(self, pai1, pai2):

        if (self.crossover_por_entrada): # Crossover de X com X (0 a 10) e de Y com Y (10 a 20)
            # Calcula um local aleatório de corte
            total_bits_x = self.qtde_bits
            local_corte_1 = round(random.random()  * total_bits_x)
            local_corte_2 = round(random.random()  * total_bits_x)

            if (local_corte_1 > local_corte_2):
                corte_ini = local_corte_2
                corte_fim = local_corte_1
            else:
                corte_ini = local_corte_1
                corte_fim = local_corte_2

            # Filho 1
            # X do filho 1
            filho1 = Individuo(self.qtde_bits, self.numero_geracao)
            x1 = []
            x1 = np.append(pai1.x[0:corte_ini] , pai2.x[corte_ini:corte_fim])
            x1 = np.append(x1, pai1.x[corte_fim::])
            filho1.x = x1
```

```
            # Y do filho 1
            y1 = []
            y1 = np.append(pai1.y[0:corte_ini] , pai2.y[corte_ini:corte_fim])
            y1 = np.append(y1, pai1.y[corte_fim::])
            filho1.y = y1
            filho1.xy = np.append(x1, y1)

            # Filho 2
            # X do filho 2
            filho2 = Individuo(self.qtde_bits, self.numero_geracao)
            x2 = []
            x2 = np.append(pai2.x[0:corte_ini] , pai1.x[corte_ini:corte_fim])
            x2 = np.append(x2, pai2.x[corte_fim::])
            filho2.x = x2
            # Y do filho 2
            y2 = []
            y2 = np.append(pai2.y[0:corte_ini] , pai1.y[corte_ini:corte_fim])
            y2 = np.append(y2, pai2.y[corte_fim::])
            filho2.y = y2
            filho2.xy = np.append(x2, y2)

        else: # Crossover em X e Y como um cromossomo s'o
            # Calcula um local aleatório de corte
            total_bits = len(pai1.xy)
            rdn_1 = random.random()
            local_corte_1 = round(rdn_1  * (total_bits-1))
            rdn_2 = random.random()
            local_corte_2 = round(rdn_2  * (total_bits-1))

            if (local_corte_1 > local_corte_2):
                corte_ini = local_corte_2
                corte_fim = local_corte_1
            else:
                corte_ini = local_corte_1
                corte_fim = local_corte_2

            filho1 = Individuo(self.qtde_bits, self.numero_geracao)
            xy1 = []
            xy1 = np.append(pai1.xy[0:corte_ini] , pai2.xy[corte_ini:corte_fim ])
            xy1 = np.append(xy1, pai1.xy[corte_fim::])
            filho1.xy = xy1
            filho1.x = xy1[0:self.qtde_bits]
            filho1.y = xy1[self.qtde_bits::]

            filho2 = Individuo(self.qtde_bits, self.numero_geracao)
            xy2 = []
            xy2 = np.append(pai2.xy[0:corte_ini] , pai1.xy[corte_ini:corte_fim])
            xy2 = np.append(xy2, pai2.xy[corte_fim::])
            filho2.xy = xy2
            filho2.x = xy2[0:self.qtde_bits]
            filho2.y = xy2[self.qtde_bits::]

        novos_filhos = []
```

```
        novos_filhos.append(filho1)
        novos_filhos.append(filho2)

        return novos_filhos

    # Efetua o crossover de duplo ponto
    def efetuar_crossover_uniforme(self, pai1, pai2):

        filho1 = Individuo(self.qtde_bits, self.numero_geracao)
        filho2 = Individuo(self.qtde_bits, self.numero_geracao)

        for i in range(len(pai1.xy)):
            if random.random() < 0.5:
                filho1.xy[i] = pai1.xy[i]
            else:
                filho1.xy[i] = pai2.xy[i]

            if random.random() < 0.5:
                filho2.xy[i] = pai1.xy[i]
            else:
                filho2.xy[i] = pai2.xy[i]


        novos_filhos = []
        novos_filhos.append(filho1)
        novos_filhos.append(filho2)

        return novos_filhos


    # baseado em uma taxa de mutação, verifica se um numero randomindo gerado é menor q a taxa. Se
for, efetua a mutação. Se não, não.
    def efetuar_mutacao(self, novos_filhos):
        if self.tipo_mutacao == TIPO_MUTACAO.TODOS_GENES: # verifica em cada gene e, se a taxa for
ok, ajusta o gene
            for filho in novos_filhos:
                for j in range(len(filho.xy)): # iterage entre os 20 bits
                    # Verifica se efetua a mudança no gene
                    if random.random() < self.taxa_mutacao:
                        if filho.xy[j] == 1:
                            filho.xy[j] = 0
                        else:
                            filho.xy[j] = 1

        elif self.tipo_mutacao == TIPO_MUTACAO.GENE_UNICO: # ajusta um gene
            for filho in novos_filhos:
                if random.random() < self.taxa_mutacao:
                    i_gene = round(random.random() * len(filho.xy)-1)
                    if filho.xy[i_gene] == 1:
                        filho.xy[i_gene] = 0
                    else:
                        filho.xy[i_gene] = 1
```

```
        elif self.tipo_mutacao == TIPO_MUTACAO.N_GENES: # Ajusta n genes
            for filho in novos_filhos:
                if random.random() < self.taxa_mutacao:
                    qtd_gene  = round(random.random() * len(filho.xy)-1)
                    for i in range(qtd_gene):
                        i_gene = round(random.random() * len(filho.xy)-1)
                        if filho.xy[i_gene] == 1:
                            filho.xy[i_gene] = 0
                        else:
                            filho.xy[i_gene] = 1


        return novos_filhos

    # Execução do algoritmo genético
    def executar_algoritmo_genetico(self):
        inicio = time.time()
        # Gera a poupalação inicial, de forma aleatória
        self.gerar_populacao_inicial()
        self.avaliar_populacao()
        self.ordenar_populacao()
        self.imprime_melhor_pior_da_geracao()

        for i in range(self.geracoes):
            # soma_fitness = self.soma_total_fitness()
            nova_geracao = []
            self.numero_geracao += 1 # Acrescenta o numero da geração

            # Se algum elitismo estiver preparado, guarda os n's melhores
            if(self.qtde_individuo_elitismo > 0):
                for i in range(self.qtde_individuo_elitismo):
                    nova_geracao.append(self.populacao[i])

            # Se o Elitismo for 0, gera desde o inicio toda a geração
            # Se alguma qtde de elitismo foi definida, inicia o contador abaixo a partir deste
ponto, para manter o tamanho definido da população
            for j in range(self.qtde_individuo_elitismo, self.individuos_por_geracao, 2): # pula de
2 em 2 pois a cada iteração gera 2 filhos
                pai1, pai2 = self.selecionar_pais()

                novos_filhos = self.efetuar_crossover(pai1, pai2)
                novos_filhos = self.efetuar_mutacao(novos_filhos)
                nova_geracao.append(novos_filhos[0])
                nova_geracao.append(novos_filhos[1])

            self.populacao = list(nova_geracao)
            self.avaliar_populacao()
            self.ordenar_populacao()
            self.imprime_melhor_pior_da_geracao()


        fim = time.time()
```

```
        # Imprime melhor individuo
        print( "\n" + self.retornar_descricao_tipo_selecao() + " | " +
self.retornar_descricao_tipo_crossover())
        print("\nMELHOR INDIVIDUO: \nGeração:%s -> \nFitness: %s \nResultado Função: %s \nx1: %s
\nx2: %s \nValor x: %s \nValor y: %s" % (self.melhor_individuo.geracao,
                                                        self.melhor_individuo.fitness,
                                                        -self.melhor_individuo.fitness,

self.melhor_individuo.xy[0:self.qtde_bits],

self.melhor_individuo.xy[self.qtde_bits::],

self.calcular_valor_decimal_q(self.melhor_individuo.xy[0:self.qtde_bits]),

self.calcular_valor_decimal_q(self.melhor_individuo.xy[self.qtde_bits::])))
        self.gerar_grafico(fim-inicio)

    def gerar_grafico(self, tempo):
        import plotly
        import plotly.graph_objs as go

        tamanho = len(self.fitnesses)

        trace_melhor = go.Scatter(
            x = np.array(list(range(tamanho))),
            y = np.array(self.fitnesses),
            name= "Melhor",
            mode= "lines+markers"
        )

        trace_media = go.Scatter(
            x = np.array(list(range(tamanho))),
            y = np.array(self.media_fitnesses),
            name= "Media",
            mode= "lines+markers"
        )

        dados_normalizacao  = [trace_melhor, trace_media]


        plotly.offline.plot({
            "data": dados_normalizacao,
            "layout": go.Layout(
                title="Fitness -> <i>Tempo(seg)</i>:" + str(round(tempo,2)) + "<br><b>Individuos
por geração</b>: " + str(self.individuos_por_geracao) + "| <b>Tipo Selecao</b>: " +
self.retornar_descricao_tipo_selecao() + " | <b>Elitismo</b>: " + str(self.qtde_individuo_elitismo)
+ "<br><b>Tipo Crossover</b>: " + self.retornar_descricao_tipo_crossover() + " | <b>Tipo
Mutação</b>: " + self.retornar_descricao_tipo_mutacao()  + " | <b>Mutacao</b>: " +
str(self.taxa_mutacao) ,
                xaxis=dict(
                    title="Geração",
                    showticklabels=True,
                    dtick=1
```

```
                ),
                yaxis=dict(
                    title="Fitness",
                    range=[-50, 0]
                )
            )
        }, auto_open=True, filename= "genetico_" + self.retornar_descricao_tipo_selecao()  + "_" +
self.retornar_descricao_tipo_crossover() + "_" + str(self.geracoes) + "_" +
str(self.individuos_por_geracao)  + ".html")


    def retornar_descricao_tipo_selecao(self):
        if self.tipo_selecao == TIPO_SELECAO.ROLETA:
            return "Roleta"
        elif self.tipo_selecao == TIPO_SELECAO.AMOSTRAGEM_ESTOCASTICA:
            return "Amostragem Estocastica"
        elif self.tipo_selecao == TIPO_SELECAO.TORNEIO:
            return "Torneio"
        elif self.tipo_selecao == TIPO_SELECAO.RANK_LINEAR:
            return "Rank Linear"

    def retornar_descricao_tipo_crossover(self):
        if self.tipo_crossover == TIPO_CROSSOVER.UNICO_PONTO:
            return "Unico Ponto"
        elif self.tipo_crossover == TIPO_CROSSOVER.DUPLO_PONTO:
            return "Duplo Ponto"
        elif self.tipo_crossover == TIPO_CROSSOVER.UNIFORME:
            return "Uniforme"
        elif self.tipo_crossover == TIPO_CROSSOVER.ARITMETICO:
            return "Aritmetico"

    def retornar_descricao_tipo_mutacao(self):
        if self.tipo_mutacao == TIPO_MUTACAO.GENE_UNICO:
            return "Testa 1 Gene"
        elif self.tipo_mutacao == TIPO_MUTACAO.TODOS_GENES:
            return "Testa Todos Genes"
        elif self.tipo_mutacao == TIPO_MUTACAO.N_GENES:
            return "Testa N Genes"
```