INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO ESPÍRITO SANTO
IFES CAMPUS SERRA

INTELIGÊNCIA ARTIFICIAL

KELVIN LEHRBACK
20171BSI0448

OTIMIZAÇÃO UTILIZANDO ALGORITMO GENÉTICO

SERRA
2022

Sumário

1 - Explicação Teórica do Algoritmo

Simulando o comportamento biológico, o algoritmo genético é responsável por encontrar uma ótima solução do problema (normalmente o mínimo ou o máximo de uma função. O algoritmo é baseado na mescla de um cromossomo de um casal de pais, mesclando e alterando algumas características do indivíduo em busca do melhor resultado.

Dada uma população inicial, é verificado o quão bom são aqueles valores, em seguida é selecionado os melhores (neste trabalho foi utilizado a seleção por torneio), em seguida é criada uma nova geração com base nos pais selecionados, aplicando taxa de crossover e mutação.

1.2 - Representação do cromossomo

Utilizamos um vetor binário para representar o cromossomo do indivíduo. O cálculo da quantidade de bits para representar o vetor é dado por: A quantidade de casas decimais e o domínio da função (neste trabalho de -20 ~ +20, ou seja, 41 valores inteiros). Ou seja, para representar 41 valores inteiros precisamos 6 bits (pois 2^6=64) e como temos 3 casas decimais, o cálculo para isso é o número de casas decimais X 3, ou seja, 9,9. Sendo assim: 6 + 9,9 = 15,9, ou seja, 16 bits para representar o cromossomo.

1.3 - Crossover

O crossover será aplicado dado uma taxa, neste caso, utilizamos 60% de taxa de crossover, caso seja aplicado, dois filhos com características dos pais serão gerados através de um corte no cromossomo.

Por exemplo: pais com cromossomo 0 1 0 1 1 1 e 1 0 0 0 1 1. Sendo o corte no meio de cada pai, irão gerar filhos com: 0 1 0 0 1 1 e 1 0 0 1 1 1, pois o corte de pai_1 foi feito em 0 1 0 | 1 1 1 e o corte de pai_2 em 1 0 0 | 0 1 1, assim, gerando dois indivíduos novos.

Caso a taxa de crossover não seja aplicada, os filhos serão idênticos aos pais.

1.4 - Mutação

Assim como na genética, o cromossomo do algoritmo também pode sofrer uma mutação,dada uma taxa, neste trabalho utilizamos a taxa de 1%.

Caso a mutação seja aplicada, um bit do respectivo cromossomo terá seu valor alterado. Por exemplo: um filho com cromossomo 1 0 0 0 1 1, um bit aleatório será alterado, vamos supor que seja o terceiro bit do cromossomo então ficará: 1 0 1 0 1 1.

1.5 - Gerações

Neste trabalho utilizaremos o critério de parada de gerações.

Gerações é a quantidade de vezes que os pais geram seus filhos em busca do melhor resultado.

1.6 - Elitismo

Pela seleção ser por torneio e por acontecer crossover e mutação, pode-se perder o melhor resultado de cada geração neste processo, sendo assim, utilizamos a técnica de elitismo, que nada mais é do que salvar o cromossomo que contém o melhor resultado da respectiva geração e transferi-lo para a geração seguinte.

1.7 - Decodificação do cromossomo

Para encontrar o valor que cada cromossomo representa, utilizamos a seguinte função:

$$x = \min + (\max - \min)\frac{b_{10}}{2^l - 1}$$

Sendo:
min = o menor valor do domínio da função.
max = o maior valor do domínio da função

b10 = o cromossomo convertido na base decimal
2^L = 2 elevado a quantidade de bits para representar o cromossomo.

Após esse cálculo, é encontrado o valor de X e depois disso, podemos jogar o valor de X na função que iremos minimizar (ou maximizar, dependendo do problema).

2 - Problema proposto

Implementar o algoritmo genético para encontrar o menor valor da seguinte função:

$$f(x) = \cos(x) * x + 2$$

3 - Implementação

Linguagem utilizada: Python 3.

Sistema Operacional utilizado: Ubuntu 18.04

IDE: Vs Code.

O código está muito bem comentado e o nome das funções são de fácil compreensão.

A seguir, segue a explicação do código detalhadamente.

3.1 - Bibliotecas utilizadas

Foram utilizadas algumas bibliotecas auxiliares para a execução do código, mas não por motivos de complexidade matemática, mas sim para a saída dos resultados.

Por exemplo, a biblioteca **pathlib** possibilita a criação de diretórios. Diretórios estes que são salvos os gráficos e planilhas gerados pelo algoritmo.

A biblioteca **matplot** contém funções utilizadas para criação dos gráficos.

A biblioteca **random** contém funções para criação de números aleatórios.

Enquanto da biblioteca **math** foi utilizada a função de cosseno.

3.2 - A função a ser minimizada

Esta é a função que iremos minimizar. Em cada geração, cada cromossomo é decodificado e chama essa função para avaliar o quão bom o mesmo é.

```
#Funcao a ser minimizada
def fitness_function(x):
    return math.cos(x) * x + 2
```

3.3 - Taxa de mutação

A seguir, a função responsável por realizar ou não a taxa de mutação, dado o cromossomo e a taxa de mutação.

Caso a taxa de mutação seja aplicada, um bit aleatorio do cromossomo será alterado.

```
def mutacao(cromossomo, taxa_mutacao):
    muta = random.uniform(0, 1)
    if(muta <= taxa_mutacao):
        change_bit = random.randint(0, len(cromossomo)-1)
        if(cromossomo[change_bit]):
            cromossomo[change_bit] = 0
        else:
            cromossomo[change_bit] = 1
    return cromossomo
```

3.4 - Taxa de Crossover

Dados os pais, é realizado o crossover.

Caso a taxa de crossover seja aplicada, os filhos terão caracteristicas dos pais, caso contrário serão cópias idênticas do mesmo.

```
def crossover(pai1, pai2, taxa_crossover):
    filho1 = 0
```

```
    filho2 = 0
    cross = random.uniform(0, 1)
    if(cross <= taxa_crossover):

        half_cromossomo = int(len(pai1) / 2)
        filho1 = (pai1[:half_cromossomo] + pai2[half_cromossomo:])
        filho2= (pai1[half_cromossomo:] + pai2[:half_cromossomo])
    else:
        filho1 = pai1
        filho2 = pai2

    return filho1, filho2
```

3.5 - Geração dos filhos

Com a lista de pais selecionados, hora de gerar os novos filhos da nova geração.

```
#Criacao de novos filhos aplicando crossover e mutacao
def gera_filhos(pais, taxa_crossover, taxa_mutacao,
best_cromossomo):
    filhos = []
    filhos.append(best_cromossomo)
    #Cria uma quantidade de filhos igual a de pais
    while(len(filhos) < len(pais)):
        pos_pai1, pos_pai2 = random.randint(0, len(pais)-1),
random.randint(0, len(pais)-1)
        pai1 = pais[pos_pai1]
        pai2 = pais[pos_pai2]

        #Aplica crossover
        filho1, filho2 = crossover(pai1, pai2, taxa_crossover)

        #Aplicar taxa de mutacao
        filho1 = mutacao(filho1, taxa_mutacao)
        filho2 = mutacao(filho2, taxa_mutacao)

        filhos.append(filho1)
        if(len(filhos) < len(pais)):
            filhos.append(filho2)

    return filhos
```

3.6 - Funções para decodificação

As funções a seguir foram criadas para auxiliar na decodificação do cromossomo, sendo funções que converte binário para decimal, binário para string, array de binário que converte para decimal e por fim, a decodificação que faz uso dessas funções.

Lembrando sempre que o melhor cromossomo da geração anterior é levado adiante para a próxima geração.

```
#Criacao de novos filhos aplicando crossover e mutacao
def gera_filhos(pais, taxa_crossover, taxa_mutacao,
best_cromossomo):
    filhos = []
    filhos.append(best_cromossomo)
    #Cria uma quantidade de filhos igual a de pais
    while(len(filhos) < len(pais)):
        pos_pai1, pos_pai2 = random.randint(0, len(pais)-1),
random.randint(0, len(pais)-1)
        pai1 = pais[pos_pai1]
        pai2 = pais[pos_pai2]

        #Aplica crossover
        filho1, filho2 = crossover(pai1, pai2, taxa_crossover)

        #Aplicar taxa de mutacao
        filho1 = mutacao(filho1, taxa_mutacao)
        filho2 = mutacao(filho2, taxa_mutacao)

        filhos.append(filho1)
        if(len(filhos) < len(pais)):
            filhos.append(filho2)

    return filhos
```

3.7 - Elitismo

A função de elitismo é responsável por capturar o melhor cromossomo de cada geração, utilizando a função de fitness para avaliar o seu valor.

```
def elitismo(geracao, min, max, size_bin):
    value_elite = float('inf')
    cromossomo = []
    for value in geracao:
        x = decodificacao(value, min, max, size_bin)
        value_x = fitness_function(x)
        if(value_x < value_elite):
            cromossomo = value
            value_elite = value_x
    return cromossomo
```

3.8 - Seleção dos pais mais aptos

Utilizamos a seleção por torneio para capturar os melhores pais de cada geração.

```
#Selecao por torneio
#Retorna uma lista com uma nova geracao
def best_values(lista, min, max, qtd_bits):
    nova_geracao = []
    tam_geracao = len(lista)
    #-1 porque ja foi aplicado o elitismo
    for _ in range(tam_geracao):
        pos1, pos2 = random.randint(0, tam_geracao-1),
random.randint(0, tam_geracao-1)
        #Verifico qual melhor resultado com base no sorteio
        x1 = decodificacao(lista[pos1], min, max, qtd_bits)
        x2 = decodificacao(lista[pos2], min, max, qtd_bits)
        if(fitness_function(x1) < fitness_function(x2)):
            nova_geracao.append(lista[pos1])
        else:
            nova_geracao.append(lista[pos2])

    return nova_geracao
```

3.9 - Criação da população inicial

Para iniciarmos o algoritmo, é necessário criarmos a população inicial, dado o tamanho da população e a quantidade de bits que representará cada cromossomo.

```
#Cria uma populacao dado o tamanho e a quantidade de bits a ser representada
def populacao_inicial(tam_pop, tam_bit):
    populacao = []
    for _ in range(tam_pop):
        new_cromossomo = []
        for _ in range(tam_bit):
            new_cromossomo.append(random.randint(0,1))
        populacao.append(new_cromossomo)

    return populacao
```

3.10 - O Genético Binário

E por fim temos a execução do algoritmo em si. A cada geração é verificado o melhor cromossomo, selecionado os melhores pais, gerada uma nova geração a ter o critério de parada ser satisfeito.

É retornada uma lista com os melhores resultados de cada geração e o melhor resultado global.

```
#Tamanho da populacao / Numero de geracoes / Quantidade de bits p/ cromossomo / valores min e max
#retorna: [0] = Array dos melhores resultados de cada geracao / o melhor resultado
def genetico_binario(tam_populacao_inicial, n_geracoes, qtd_bits, min, max):
    populacao = populacao_inicial(tam_populacao_inicial, qtd_bits)
    geracao = 1
    best_cromossomo = [0]
    list_best_generation = []
    #Salva o melhor valor da populacao
    while(geracao < n_geracoes):
        best_cromossomo = elitismo(populacao.copy(), min, max,
```

```
qtd_bits)

list_best_generation.append(fitness_function(decodificacao(best_cr
omossomo, min, max, qtd_bits)))
        #Captura os pais via torneio
        pais = best_values(populacao.copy(), min, max, qtd_bits)
        #Gera uma nova populacao
        populacao = gera_filhos(pais.copy(), 0.6, 0.01,
best_cromossomo.copy())
        geracao += 1

    #Verificando a ultima geracao
    best_cromossomo = elitismo(populacao.copy(), min, max,
qtd_bits)

list_best_generation.append(fitness_function(decodificacao(best_cr
omossomo, min, max, qtd_bits)))
    return [list_best_generation,
fitness_function(decodificacao(best_cromossomo, min, max,
qtd_bits))]
```

3.11 - Salvando os gráficos

Ao término da execução do código, são salvos os gráficos que contém a média de cada iteração e os valores na iteração do melhor resultado.

```
#Funcao para salvar os resultados obtidos a partir de determinada
execucao
def save_graph(x, result, best_result, leg):
    plt.plot(x, result, label = "Media de cada iteracao")
    plt.plot(x, best_result, label = "Melhor resultado")
    plt.title(leg)
    plt.xlabel("Iteração")
    plt.ylabel("Gbest")
    plt.legend()
    plt.savefig("plot_graphs/%s.png" %leg)
    plt.close()
```

3.12 - Execução para cada geração

O trabalho inicialmente pede para executarmos 10x para 10 e 20 gerações, porém tomei a liberdade para executar com 100 gerações também. Essa função simplesmente chama o algoritmo genético variando o número de gerações. É responsável também por salvar os gráficos e informar no terminal a saída do melhor resultado e a média de cada número de geração.

```
def run_genetico_binario(qtd_execucoes, tam_populacao):
    result_10 = []
    result_20 = []
    result_100 = []

    best_result_10 = float('inf')
    best_result_20 = float('inf')
    best_result_100 = float('inf')

    array_best_result_10 = []
    array_best_result_20 = []
    array_best_result_100 = []

    media_iteration_10 = [0] * 10
    media_iteration_20 = [0] * 20
    media_iteration_100 = [0] * 100

    for _ in range(qtd_execucoes):
        res_10 = genetico_binario(tam_populacao, 10, 16, -20, 20)
        res_20 = genetico_binario(tam_populacao, 20, 16, -20, 20)
        res_100 = genetico_binario(tam_populacao, 100, 16, -20,
20)

        #[0] = Lista dos melhores de cada geracao, [1] = valor de
y
        result_10.append(res_10[1])
        result_20.append(res_20[1])
        result_100.append(res_100[1])

        if(res_10[1] < best_result_10):
            best_result_10 = res_10[1]
            array_best_result_10 = res_10[0]
```

```
        if(res_20[1] < best_result_20):
            best_result_20 = res_20[1]
            array_best_result_20 = res_20[0]

        if(res_100[1] < best_result_100):
            best_result_100 = res_100[1]
            array_best_result_100 = res_100[0]

        for i in range(10):
            media_iteration_10[i] += res_10[1]

        for i in range(20):
            media_iteration_20[i] += res_20[1]

        for i in range(100):
            media_iteration_100[i] += res_100[1]

    for i in range(10):
        media_iteration_10[i] = media_iteration_10[i] /
qtd_execucoes

    for i in range(20):
        media_iteration_20[i] = media_iteration_20[i] /
qtd_execucoes

    for i in range(100):
        media_iteration_100[i] = media_iteration_100[i] /
qtd_execucoes

    #Tirando a media dos menores valores encontrados
    media_10 = 0
    media_20 = 0
    media_100 = 0
    for i in range(qtd_execucoes):
        media_10 += result_10[i]
        media_20 += result_20[i]
        media_100 += result_100[i]

    media_10 = media_10 / 10
    media_20 = media_20 / 10
    media_100 = media_100 / 10
```

```
    print("Melhor resultado: ")
    print("10 Geracoes: %s" %best_result_10)
    print("20 Geracoes: %s" %best_result_20)
    print("100 Geracoes: %s" %best_result_100)

    print("")
    print("Media:")
    print("10 Geracoes: %s" %media_10)
    print("20 Geracoes: %s" %media_20)
    print("100 Geracoes: %s" %media_100)

    save_graph([x for x in range(10)], media_iteration_10,
array_best_result_10, "10 geracoes")
    save_graph([x for x in range(20)], media_iteration_20,
array_best_result_20, "20 geracoes")
    save_graph([x for x in range(100)], media_iteration_100,
array_best_result_100, "100 geracoes")
```

3.13 - A main

E por fim temos a função main(), que simplesmente cria a pasta para salvar os gráficos e chama o algoritmo que executa cada geração.

```
def main():
    #Criando a pasta "plot_graphs" se nao existir
    path = Path("plot_graphs")
    path.mkdir(exist_ok=True)
    run_genetico_binario(10, 10)
    print("\n-> Checar pasta: plot_graphs")
main()
```

4 - Resultados

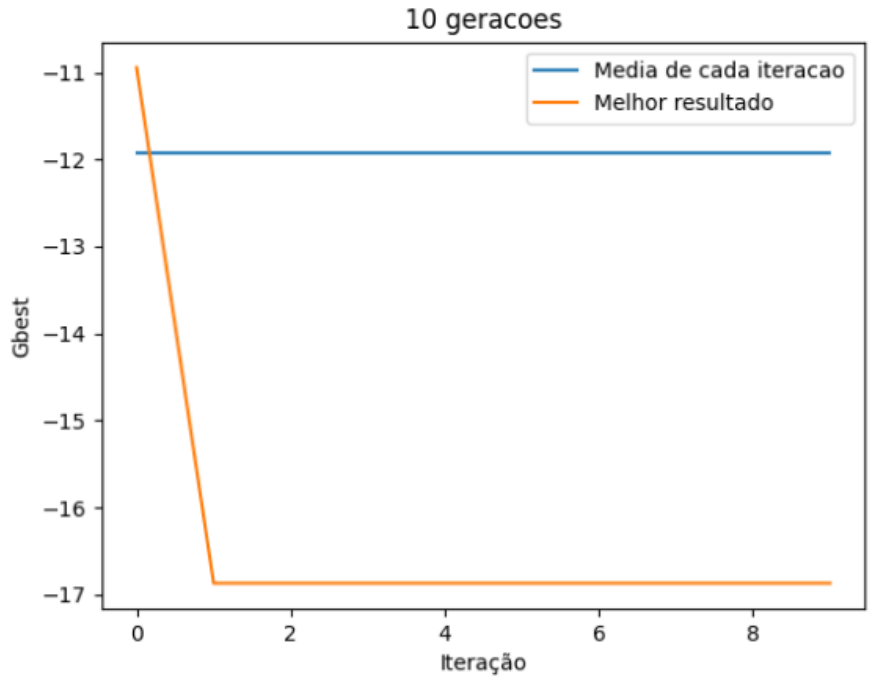
A seguir estão os resultados, vale ressaltar um dado curioso a respeito desta implementação que percebi ao longo dos testes. Originalmente a taxa de mutação é de 1%, porém, caso eu varie ela para 50%, a precisão aumenta (a média dos resultados fica melhor).

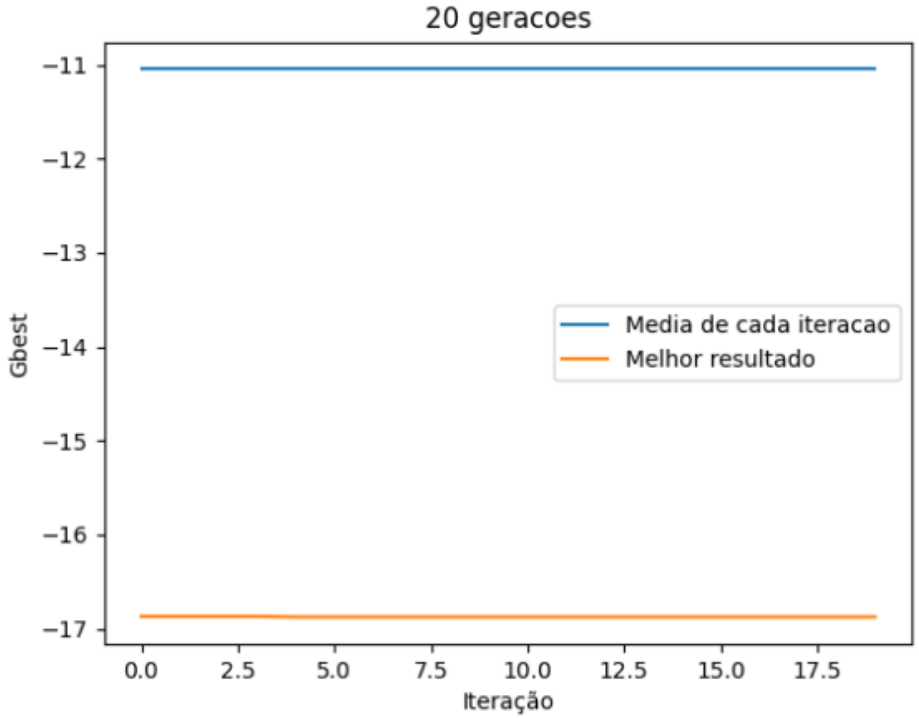
O melhor resultado não varia muito, encontrando sempre uma boa solução(com base no gabarito informado pelo professor).

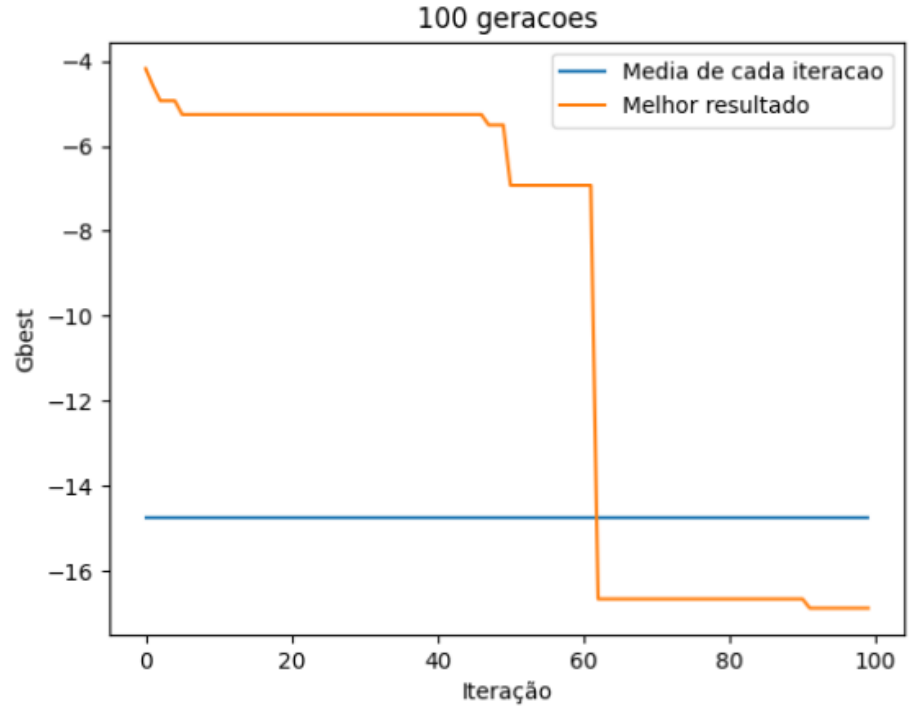
4.1 - 1% de mutação

```
Melhor resultado:
10 Geracoes: -16.871100250711166
20 Geracoes: -16.876011783166277
100 Geracoes: -16.876013451379777

Media:
10 Geracoes: -11.932914226501655
20 Geracoes: -11.046528636705649
100 Geracoes: -14.747201765237497
```



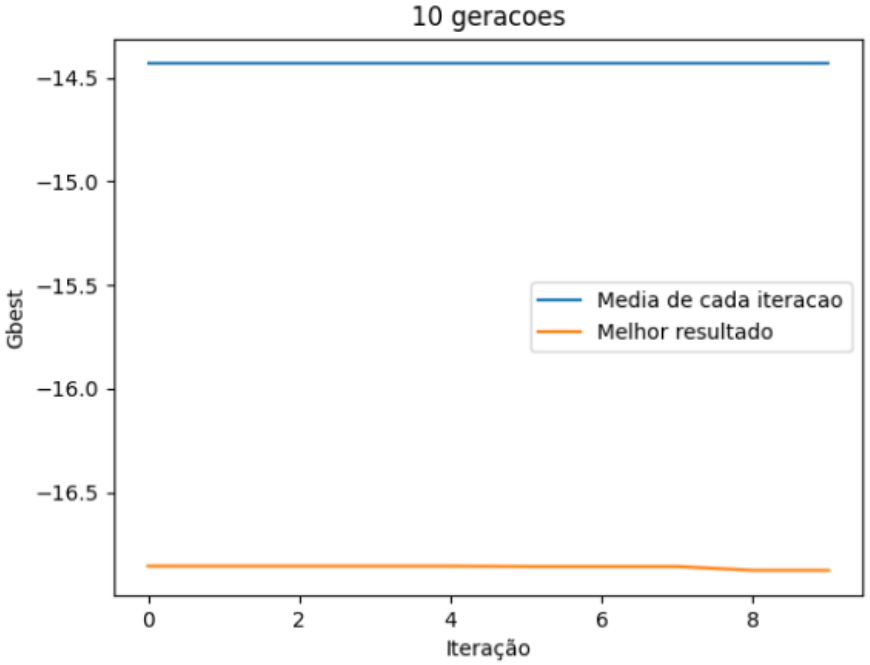

20 geracoes
Media de cada iteracao
Melhor resultado
Gbest
Iteração
−11
−12
−13
−14
−15
−16
−17
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

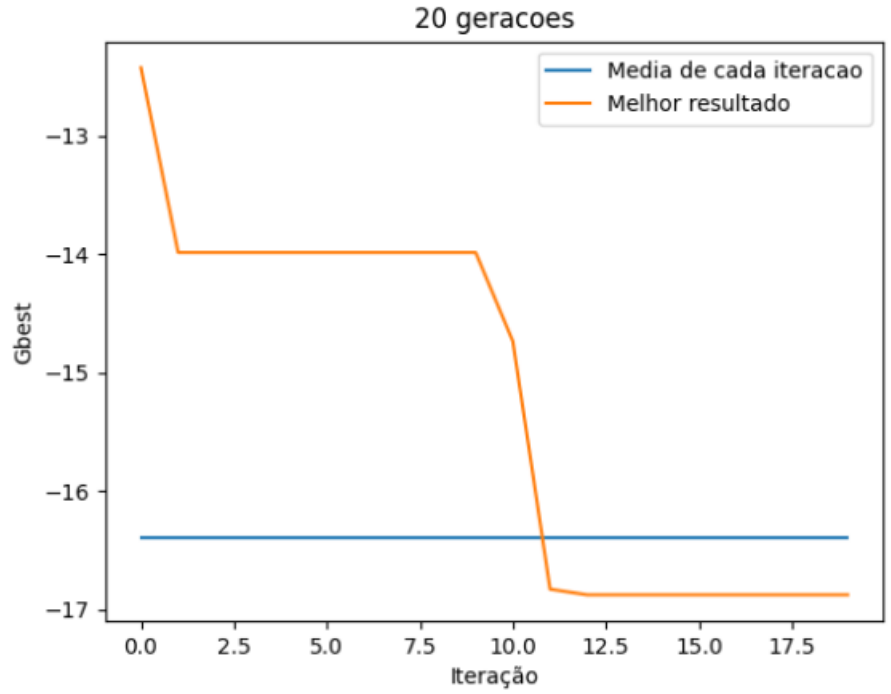

100 geracoes
Media de cada iteracao
Melhor resultado
Gbest
Iteração
−4
−6
−8
−10
−12
−14
−16
0
20
40
60
80
100

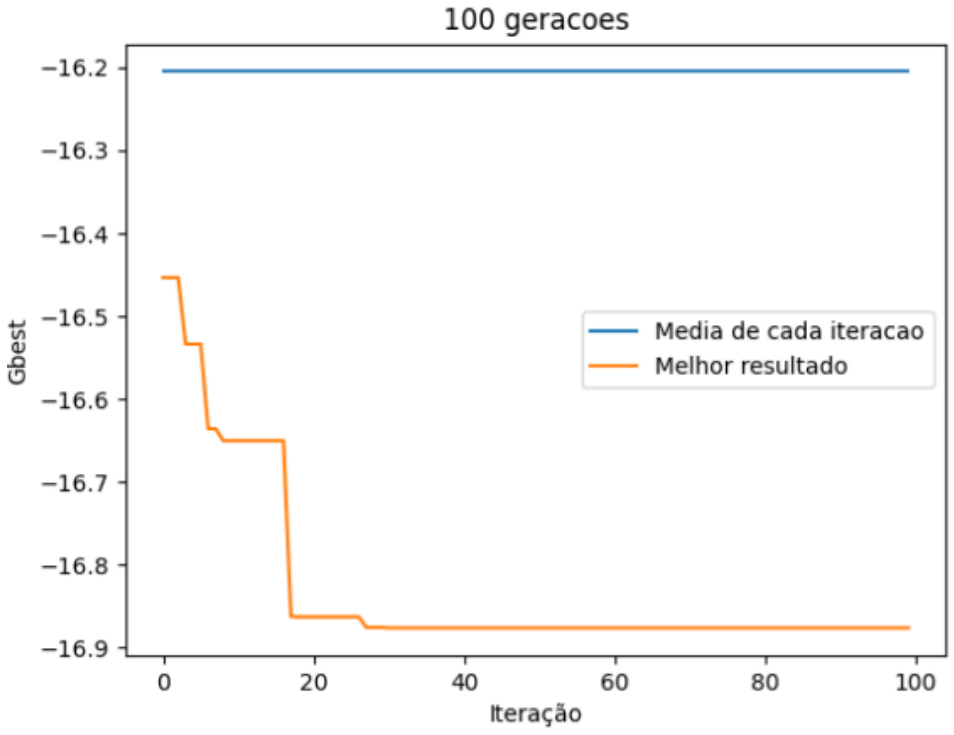
4.2 - 50% de mutação

```
Melhor resultado:
10 Geracoes: -16.87571019142799
20 Geracoes: -16.876013451379777
100 Geracoes: -16.876013451379777

Media:
10 Geracoes: -14.433616377741975
20 Geracoes: -16.396104371972363
100 Geracoes: -16.20496360628268
```







5 - Conclusão

Bom, é nítido que o algoritmo encontra sempre uma boa solução, seja com 10, 20 ou 100 gerações, encontrando valores extremamente próximos do objetivo, ou seja, o ponto mínimo da função (-16,785).

Vemos também que conforme aumentamos a quantidade de gerações, os resultados ficam mais precisos, encontrando até mesmo o ponto mínimo exato da função em alguns testes.

Vale ressaltar que foram apenas 10 rodadas para cada geração, conforme aumentamos a quantidade de rodadas, teremos um desempenho e uma média bem melhor.

O ponto curioso que ainda estou tentando assimilar é o motivo do desempenho ser bem melhor (melhorando a média dos resultados) caso a taxa de mutação aumente para 50% ao invés de apenas 1%.