Algoritmo Genético

Trabalho 3 referente a disciplina de Inteligência Artificial.

Alunos:

Gabriel Pontes Marchezi e Gabriel Viggiano Fonseca

Explicação Teórica

Algoritmos Genéticos (AG) são implementados como uma simulação de computador em que uma população de representações abstratas de solução é selecionada em busca de soluções melhores. A evolução geralmente se inicia a partir de um conjunto de soluções criado aleatoriamente e é realizada por meio de gerações. A cada geração, a adaptação de cada solução na população é avaliada, alguns indivíduos são selecionados para a próxima geração, e recombinados ou mutados para formar uma nova população. A nova população então é utilizada como entrada para a próxima iteração do algoritmo.

Algoritmos genéticos diferem dos algoritmos tradicionais de otimização em basicamente quatro aspectos:

1. Baseiam-se em uma codificação do conjunto das soluções possíveis, e não nos parâmetros da otimização em si;
2. os resultados são apresentados como uma população de soluções e não como uma solução única;
3. não necessitam de nenhum conhecimento derivado do problema, apenas de uma forma de avaliação do resultado;
4. usam transições probabilísticas e não regras determinísticas.

Problema Proposto

O problema proposto é realizar a implementação do algoritmo genético a fim de minimizar a função abaixo.

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

Vale ressaltar que para o modo de seleção, iremos utilizar o "Torneio". A taxa de Crossover será de 60%, a taxa de mutação de 1% e executaremos o algoritmo com 10 e 20 iterações.

Implementação

Para a implementação do Algoritmo Genético, foi utilizada a linguagem de programação Python, essa que vem ganhando bastante popularidade na área de de Inteligência Artificial, por ser uma linguagem de fácil aprendizado e com alto nível de produtividade. E também pelo fato de podermos gerar gráficos através da biblioteca Matplotlib, visto que para esse trabalho, os resultados são de extrema importância. Dito isso, vamos ao código:

Trechos mais Importantes

Função InicializarPopulação

Recebe como parâmetro o número de indivíduos que irá compor a população. Em loop vai criando os indivíduos, composto por um gene binário de 10 bits, e adicionando-os em um lista que representa a população.

```
#Inicializa a população com valores aleatórios
def InicializarPopulação (numPop):
    listaPop = []
    individuo = 0
    while (numPop > 0):
        individuo = random.randint(0,1024)
        listaPop.append(DecToBin(individuo))
        numPop = numPop - 1
    return listaPop
```

Funções de conversão para Decimal-Binário e Binário-Decimal

Funções auxiliares responsáveis pela conversão entre as bases binária e decimal

```
# Converte um número decimal para um número binário de 10 bits
def DecToBin(dec):
    b1n = bin(dec)[2:]
    while (len(b1n) < 10):
        b1n = '0' + b1n
    return b1n

# Convente um número binário para um número decimal
def BinToDec(b1n):
    b1n = '0b' + b1n
    dec = int(b1n,2)
    return dec
```

Função Avalia_Individuo

Faz o cálculo da pontuação do indivíduo utilizando a função f(x) = Cos(x)*x+2

```
# Avalia a aptidão do invidíduo
def Avalia_Individuo(individuo):
    x = (40/1024)*BinToDec(individuo)-20
    resultado = round(math.cos(x)*x+2,2)
    return resultado
```

Função Melhor_Pior

Recebe como parâmetro um lista de população e retorna o indivíduo com melhor e pior aptidão

```
#Recebe uma lista de individuos e retorna uma lista com os individuos com melhor e pior aptidão
def Melhor_Pior (lst_ind):
    lista = []
    melhor = lst_ind[0]
    pior = lst_ind[0]
    for individuo in lst_ind:
        if (Avalia_Individuo(individuo) < Avalia_Individuo(melhor)):
            melhor = individuo
        if(Avalia_Individuo(individuo) > Avalia_Individuo(pior)):
            pior = individuo
    lista.append(melhor)
    lista.append(pior)
    return lista
```

Função Torneio

Recebe como parâmetro uma lista de população e retorna uma lista de pais selecionados do mesmo tamanho da população recebida como parâmetro. Os possíveis pais são selecionados aleatoriamente de 2 em 2 e o que possuir a melhor aptidão é escolhido como pai

```
#seleciona dos pais através do método torneio
def Torneio(populacao):
    lista_torneio = []
    for i in range(len(populacao)):
        pai1 = populacao[random.randint(0,len(populacao)-1)]
        pai2 = populacao[random.randint(0,len(populacao)-1)]
        if Avalia_Individuo(pai1) < Avalia_Individuo(pai2):
            lista_torneio.append(pai1)
        else:
            lista_torneio.append(pai2)
    return lista_torneio
```

Função Crossover

Responsável por gerar uma nova população. Recebe como parametro a lista de pais escolhidos no torneio e se o corte dos genes dos pais será uniforme (ao meio) ou aleatório, de 2 em 2 os pais são selecionados, é aplicada uma taxa de crossover e caso a taxa seja satisfeita os filhos são gerados através da mistura dos genes dos pais, caso contrário os filhos serão cópias dos seus pais.

```python
#Aplica o Crossover com uma taxa passada por parametro de maneira uniforme ou não
def Crossover(lst_pais, taxa, uniforme):
    lst_filhos = []
    for i in range(0,len(lst_pais),2):
        if(random.randint(1,100) <= taxa):
            if(uniforme == 0):
                ponto = random.randint(0,len(lst_pais[i]))
                filho1 = lst_pais[i][:ponto] + lst_pais[i+1][ponto:]
                filho2 = lst_pais[i][ponto:] + lst_pais[i+1][:ponto]
            else:
                ponto = int(len(lst_pais[i])/2)
                filho1 = lst_pais[i][:ponto] + lst_pais[i+1][ponto:]
                filho2 = lst_pais[i][ponto:] + lst_pais[i+1][:ponto]
        else:
            filho1 = lst_pais[i]
            filho2 = lst_pais[i+1]
        lst_filhos.append(filho1)
        lst_filhos.append(filho2)
    return lst_filhos
```

Função Mutacao

Recebe como parâmetro uma taxa de mutação, percorre os genes dos indivíduos e caso a taxa satisfaça a condição, inverte o bit do indivíduo

```python
def Mutacao (lst_filhos, taxa):
    lst_mutada = []
    for filho in lst_filhos:
        b1n = ''
        for bit in filho:
            if(random.randint(1,100) <= taxa):
                if(bit == '1'):
                    b1n += '0'
                else:
                    b1n += '1'
            else:
                b1n += bit
        lst_mutada.append(b1n)
    return lst_mutada
```

Função Algo_Gen

Recebe como parâmetro a população inicial,o número de iterações, a taxa de mutação, taxa de crossover e se o corte dos genes no crossover será uniforme. A função faz apenas uma execucação do algoritmo genético, em um loop definido pelo número de iterações aplica as funções previamente definidas. Retorna uma lista contendo os indivíduos de cada iteração.

```python
def Algo_Gen(populacao, iteracoes, taxa_muta, taxa_cross, uniform_cross):
    elite = Melhor_Pior(populacao)
    ite_atual = 0
    lst_exe = []
    while(ite_atual < iteracoes):
        populacao = Torneio(populacao)
        populacao = Crossover(populacao,taxa_cross,uniform_cross)
        populacao = Mutacao(populacao,taxa_muta)
        elite_it = Melhor_Pior(populacao)
        if (Avalia_Individuo(elite_it[0]) < Avalia_Individuo(elite[0])):
            elite[0] = elite_it[0]
        cont = 0
        for individuo in populacao:
            if (individuo == elite_it[1]):
                del(populacao[cont])
                populacao.append(elite[0])
            cont += 1
        ite_atual +=1
        lst_exe.append(populacao)
    return lst_exe
```

Função principal parte 1

Recebe os dados do usuário, o algoritmo genético será executado N vezes em um loop e será gerada uma lista de execuções

```python
def main():
    numpop = int(input("Digite a tamanho da população: "))
    iteracoes = int(input("Digite o número de iterações: "))
    taxa_muta = int(input("Digite a taxa de mutação: "))
    taxa_cross = int(input("Digite a taxa de crossover: "))
    uniform_cross = int(input("O crossover terá ponto uniforme?(1 para sim, 0 para não): "))
    execucoes = int(input("Digite o número de execuções:"))

    #Executa o Algoritmo Genético N vezes
    exe_atual = 0
    lst_exes = []
    while(exe_atual < execucoes):
        populacao_inic = []
        populacao_inic = InicializarPopulação(numpop)
        resultado = Algo_Gen(populacao_inic, iteracoes, taxa_muta, taxa_cross, uniform_cross)
        lst_exes.append(resultado)
        exe_atual += 1
```

Função principal parte 2

Percorre a lista de execuções e para cada iteração de cada execução é encontrado e armazenado em uma lista o melhor resultado. Inicia a lista de dos valores das médias das iterações preenchendo com zero

```
#Encontra os melhores resultados de todas as iterações em todas as execuções
lst_melhores_exe = []
for exe in lst_exes:
    lst_melhores_ite = []
    for ite in exe:
        melhor = 1
        for ind in ite:
            if (Avalia_Individuo(ind) < melhor):
                melhor = Avalia_Individuo(ind)
        lst_melhores_ite.append(melhor)
    lst_melhores_exe.append(lst_melhores_ite)

#Inicia a lista com as médias dos resultados preenchendo com zeros
lista_medias = []
for i in range(0,iteracoes):
    lista_medias.append(0)
```

Função principal parte 3

Percorre a lista de melhores valores de cada iteração em cada execução e vai somando o valor na lista de valores médios afim de encontrar as média das iterações. Encontra e armazena os valores da execução com a melhor aptidão.

```
#Calcula a média das iterações das execuções e guarda também a execução com melhor resultado
lst_melhor_result = [['']]
melhor = 1
for exe in lst_melhores_exe:
    pos = 0
    temp = []
    cond = False
    for result in exe:
        if(result < melhor):
            cond = True
            melhor = result
        temp.append(result)
        lista_medias[pos] += result
        pos += 1
    if(cond == True):
        lst_melhor_result[0] = temp
print("melhor: " + str(melhor))
print(lst_melhor_result)
cont = 0
for i in lista_medias:
    lista_medias[cont] = lista_medias[cont]/execucoes
    cont += 1
print("Médias das iterações em "+ str(execucoes) + " execuções: ")
print(lista_medias)
```

Resultados

Gráfico de 10 execuções do algoritmo genético com 10 iterações e populações de 10 indivíduos, melhor valor: -16.876

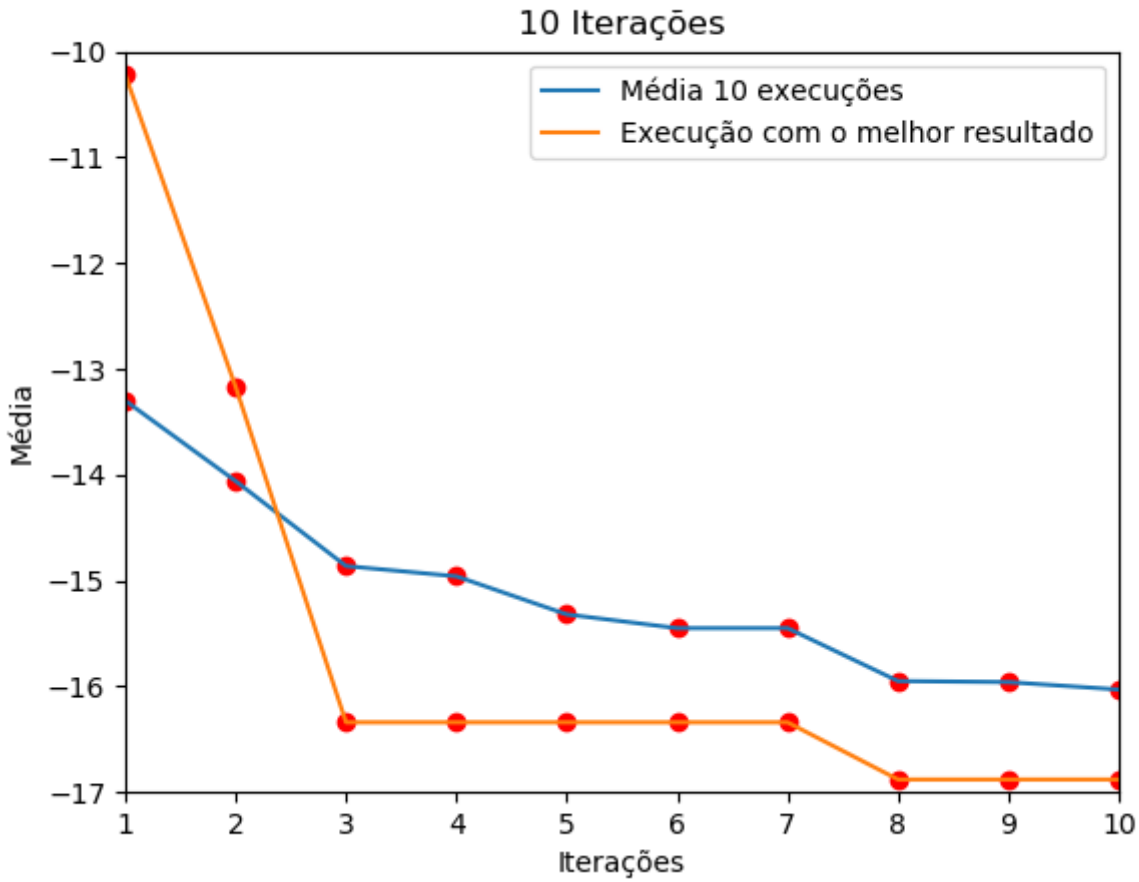
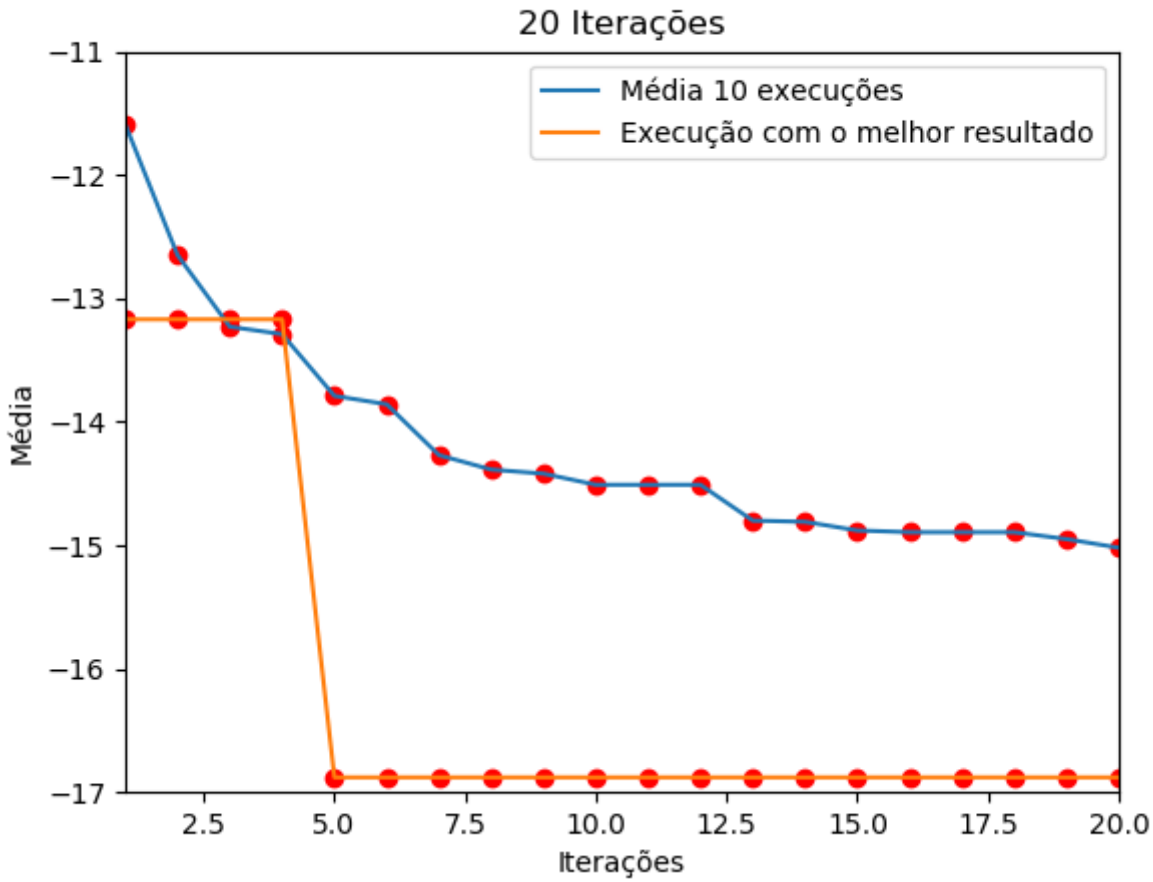


Gráfico de 10 execuções do algoritmo genético com 20 iterações e populações de 10 indivíduos, melhor valor: -16.876



Melhor média de 10 execuções com 10 Iterações

Média: -16.622

Taxa de mutação: 60%, taxa de crossover: 80%

```
Digite a tamanho da população: 10
Digite o número de iterações: 10
Digite a taxa de mutação: 60
Digite a taxa de crossover: 80
O crossover terá ponto uniforme?(1 para sim, 0 para não): 1
Digite o número de execuções:10
melhor: -16.876
[[-13.571, -13.571, -16.755, -16.755, -16.755, -16.755, -16.755, -16.876, -16.876, -16.876]]
Médias das iterações em 10 execuções:
[-13.572899999999999, -15.136000000000001, -15.829600000000003, -15.861100000000002, -15.861100000000002,
-16.264700000000005, -16.365500000000004, -16.596300000000006, -16.596300000000006, -16.622000000000003]
```

Versão 2 do trabalho

Alterações no Crossover e Mutação

Nessa versão utilizamos um crossover aritmético e uma mutação de limite

Mudanças no códigos

```python
# Crossover Aritmético
def Crossover(lst_pais,taxa):
    lst_filhos = []
    for i in range(0,len(lst_pais),2):
        if(random.randint(1,100) <= taxa):
            b = round(random.uniform(0,1),2)
            filho1 = b * lst_pais[i] + (1 - b) * lst_pais[i+1]
            filho2 = (1 - b) * lst_pais[i] + b * lst_pais[i+1]
        else:
            filho1 = lst_pais[i]
            filho2 = lst_pais[i+1]
        lst_filhos.append(filho1)
        lst_filhos.append(filho2)
    return lst_filhos

# Mutação de limites
def Mutacao (lst_filhos, taxa, ite_atual,iteracoes):
    lst_mutada = []
    for filho in lst_filhos:
        if(random.randint(1,100) <= taxa):
            r1 = round(random.uniform(0,1),2)
            if (r1 > 0.5):
                mutado = 20
            else:
                mutado = -20
        else:
            mutado = filho
        lst_mutada.append(mutado)
    return lst_mutada
```

Resultados

Gráfico de 10 execuções do algoritmo genético com 10 iterações e populações de 10 indivíduos, melhor valor: -16.876

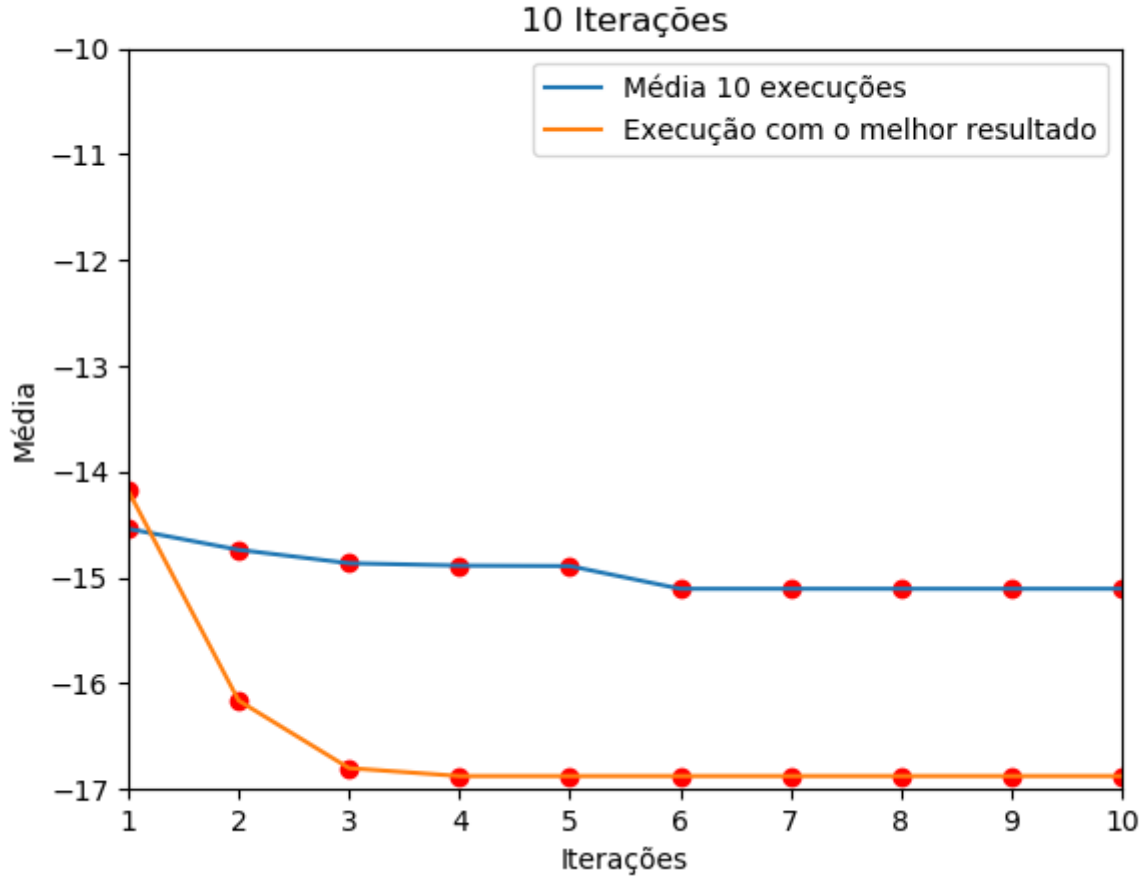
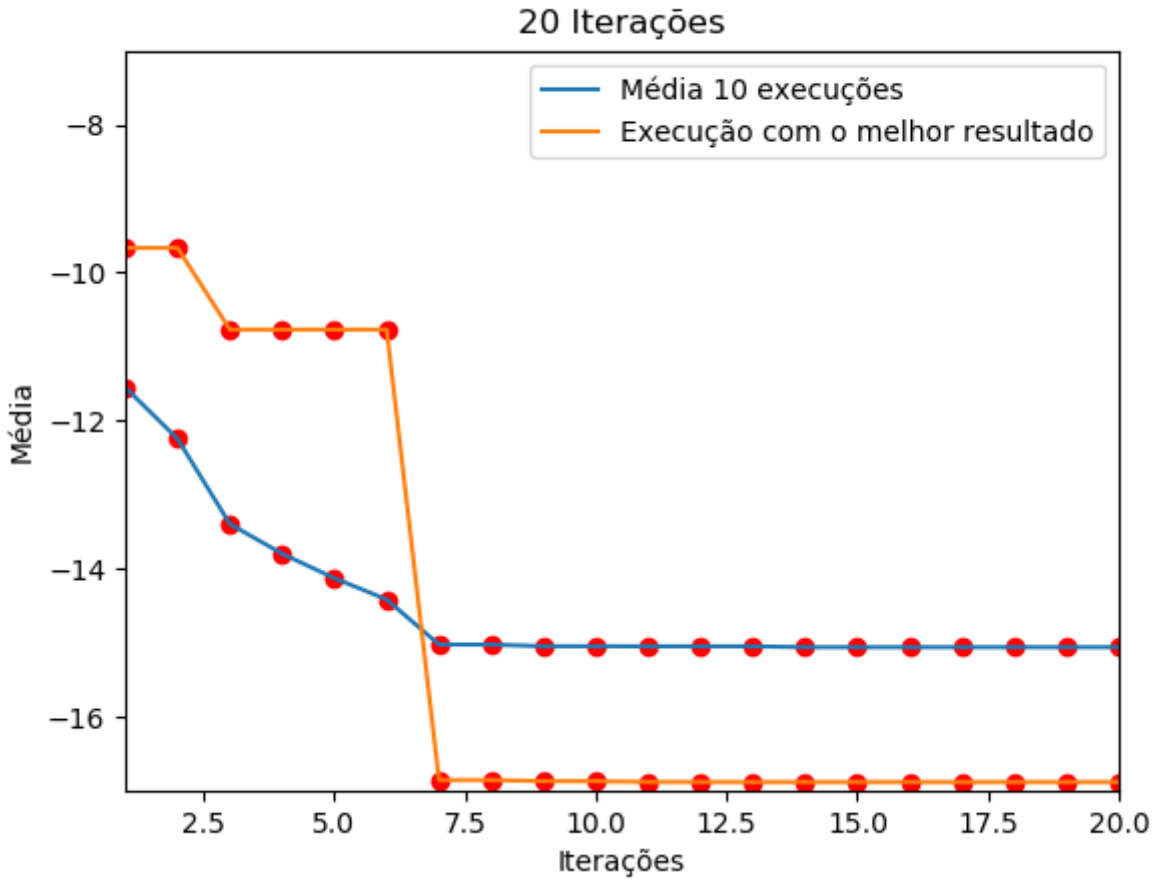


Gráfico de 10 execuções do algoritmo genético com 20 iterações e populações de 10 indivíduos, melhor valor: -16.876



Melhor média de 10 execuções com 10 Iterações

Média: -15.106

Taxa de mutação: 20%, taxa de crossover: 60%

```
PS C:\Users\gabri\desktop\Trab3_v2_IA> python .\Algoritmo_GEN_2.py
Digite a tamanho da população: 10
Digite o número de iterações: 10
Digite a taxa de mutação: 20
Digite a taxa de crossover: 60
Digite o número de execuções:10
melhor: -16.876
[[-14.179, -16.163, -16.801, -16.876, -16.876, -16.876, -16.876, -16.876, -16.876, -16.876]]
Médias das iterações em 10 execuções:
[-14.5393, -14.7377, -14.8647, -14.8867, -14.891200000000001, -15.106100000000001, -15.106100000000001, -15.106100000000001, -15.106100000000001, -15.10610
0000000001]
PS C:\Users\gabri\desktop\Trab3_v2_IA>
```

Bibliografia

https://pt.wikipedia.org/wiki/Algoritmo_genético Slides do AVA.