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

INTELIGÊNCIA ARTIFICIAL

KELVIN LEHRBACK
20171BSI0448

OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS

SERRA
2022

1 - Explicação Teórica do Algoritmo

Dada uma função, um setup, uma condição de parada e o tamanho da população, o algoritmo de otimização por enxame de partículas é responsável por encontrar uma ótima solução para o problema (normalmente o mínimo ou o máximo de uma função). O algoritmo é baseado no comportamento social de bando de pássaros e cardumes, que, através da troca de informações, determina a trajetória que cada um deles deverá tomar no espaço de busca. Ou seja, cada partícula está numa determinada posição em busca de um objetivo em comum. Dada essa posição, ela informa às demais partículas e assim, estas buscam se deslocar mais próximo do que seria a melhor posição (objetivo) informado pela respectiva partícula. A condição de parada é a quantidade de iterações.

1.2 - Os parâmetros W, C1 e C2

O PSO(Particle Swarm Optimization) contém 3 parâmetros para atualizar a velocidade de suas partículas. Esses parâmetros são necessários para “espalhar” ou “randomizar” os indivíduos, evitando situações do tipo “planicie”, por exemplo, que é quando uma determinada partícula não encontra uma boa solução mas não consegue sair de sua posição atual.

O Setup (que é quando definimos os valores de W, C1 e C2) depende de como está implementado o algoritmo, e varia de função para função.

2 - Problema Proposto

Implementar o algoritmo PSO para encontrar o menor valor da função eggholder

$$f(x,y) = -(y+47)\sin\sqrt{\left|\frac{x}{2}+(y+47)\right|} - x\sin\sqrt{|x-(y+47)|}$$

E utilizando a seguinte fórmula para atualizar a velocidade das partículas:

$$v_i(t+1) = W * v_i(t) + \phi_1 * rand_1(.) * (p_B - x_i(t)) + \phi_2 * rand_2(g_B - x_i(t))$$

Bem como a seguinte fórmula para atualizar a posição das partículas:

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

E tendo como domínio da função a ser minimizada: $x \in [-512,+512]$ e $y \in [-512,+512]$.

Sabemos que o mínimo da função é -959.6407, tendo como os pontos 512 e 404.2319. Portanto, eis o nosso objetivo (ou valores próximos disso).

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 **pandas** contém funções utilizadas para criação das planilhas.

A biblioteca random contém funções para criação de números aleatórios.

Enquanto da biblioteca **math** foram utilizadas as funções de seno e cosseno, além de raiz quadrada.

3.2 - A função Eggholder

Esta é a função que temos que minimizar data a fórmula que vimos na seção 2, cada iteração da partícula chama essa função que por sua vez, retorna o valor dado a posição da partícula

```
#Funcao fitness
def eggholder_function(position):
    #Modularizando para melhor manutencao da funcao
    seno_raiz = math.sin(math.sqrt(abs((position[0] / 2) +
(position[1] + 47))))
    x_seno_raiz = position[0] *
(math.sin(math.sqrt(abs(position[0] - (position[1] + 47)))))
    return (-1*(position[1]+47) * seno_raiz) - x_seno_raiz
```

3.3 - Randomização da posição e velocidade.

Foram criadas duas funções para randomizar a posição e velocidade inicial das partículas. Vale destacar o domínio de cada uma, sendo -512 ~ +512 para a posição e -77 ~ +77 para a velocidade.

```
def random_position():
    #Intervalos designados na especificacao do trabalho
    return [random.uniform(-512, 512),random.uniform(-512, 512)]

def random_velocidade():
    #Intervalos designados na especificacao do trabalho
    return [random.uniform(-77, 77),random.uniform(-77, 77)]
```

3.4 - Atualização da velocidade

Vimos também na seção 2 a fórmula para a atualização da velocidade, cuja a mesma se encontra na função a seguir:

```
def update_velocidade(velocidade_atual,Watual, pbest, gbest,
posicao, c1, c2):
    x = (Watual * velocidade_atual[0]) + c1 *
random.uniform(0,1)*(pbest[0] - posicao[0]) + c2 *
random.uniform(0,1)*(gbest[0] - posicao[0])
    y = (Watual * velocidade_atual[1]) + c1 *
random.uniform(0,1)*(pbest[1] - posicao[1]) + c2 *
random.uniform(0,1)*(gbest[1] - posicao[1])
    return [x,y]
```

3.5 - Atualização da posição

Ainda na seção 2, vimos também como atualizar a posição das partículas.

```
def update_posicao(posicao, velocidade):
    x = posicao[0] + velocidade[0]
    y = posicao[1] + velocidade[1]

    return [x,y]
```

3.6 Atualizando o setup (W)

A única modificação feita no setup é de W, cujo mesmo inicial com um valor máximo e vai diminuindo conforme a execução (número de iterações) até um valor mínimo também estabelecido.

```
def update_w(iteracao, Wmax, Wmin, n_iterations):
    return Wmax - (iteracao*(Wmax-Wmin) / n_iterations)
```

3.7 - Enxame de Partículas (PSO)

Tendo as funções anteriores prontas, hora de finalmente executar o algoritmo.

A primeira coisa a se fazer é definir o setup, escolhi 15 para o valor máximo de W e 1 para o mínimo. Minhas constantes C1 e C2 estão com o valor 2.5.

Setup criado e informado o número de iterações e a quantidade de partículas, o algoritmo inicia seus respectivos loops, sendo:

- O loop da condição de parada (quantidade de iteração).
- O loop para o cálculo do fitness de cada partícula: Neste loop é onde cada partícula informa qual a sua posição e compara com a melhor posição de todo o bando.
- O loop de atualização da velocidade e posição de todas as partículas.

Ao final da execução desta função é retornado um array contendo:

- O melhor valor encontrado
- A posição deste melhor valor
- Um array contendo o gbest de cada iteração

```
#Dada o numero de iteracoes e a quantidade de particulas, execute!
def run_pso(n_iterations, qtd_particulas):
      #Setup
      Wmax = 15
      Wmin = 1
      Watual = Wmax
      c1 = 2.5
      c2 = 2.5

      #2 - Inicializar aleatoriamente a posicao inicial(x) de cada
particula
      posicao_particulas = []
      valor_particulas = []
      vetor_velocidade = []

      #3 - Velocidade inicial(v) para todas as particulas
      velocidade_inicial = random_velocidade()
      for i in range(qtd_particulas):
            posicao_particulas.append(random_position())
            valor_particulas.append(float('inf'))
            vetor_velocidade.append(velocidade_inicial)
```

```
     #Atribuindo valores iniciais para pbest e gbest
     pbest_posicao = posicao_particulas
     gbest_valor = float('inf')
     gbest_posicao = [0,0]
     all_gbest_iteration = []

     #Loop principal, ele que determina quando o algoritmo para
     iteration = 0
     while iteration < n_iterations:
          #Para cada particula, calculo sua fitness
          for i in range(qtd_particulas):
               #4 a) - Calculando aptidao de 'p'
               #Funcao fitness
               result_iteracao_particula =
eggholder_function(posicao_particulas[i])

               #4 b) - Verificando a melhor posicao de 'p'
               if(result_iteracao_particula <
valor_particulas[i]):
                    valor_particulas[i] =
result_iteracao_particula
                    pbest_posicao[i] = posicao_particulas[i]

               #5 - Verificando a melhor aptdao da populacao
               if(result_iteracao_particula < gbest_valor):
                    gbest_valor = result_iteracao_particula
                    gbest_posicao = posicao_particulas[i]

          #atualizo W para randomizar a posicao das particulas
          Watual = update_w(iteration, Wmax, Wmin, n_iterations)
          #Para cada particula, atualizo a sua velocidade
(tomando cuidado com dominio)
          for i in range(qtd_particulas):
               #6 a) - Atualizando velocidade
               nova_velocidade =
update_velocidade(vetor_velocidade[i], Watual, pbest_posicao[i],
gbest_posicao, posicao_particulas[i], c1, c2)
               #Limitando a velocidade x
               if(nova_velocidade[0] < -77):
                    nova_velocidade[0] = -77
               elif(nova_velocidade[0] > 77):
```

```
                    nova_velocidade[0] = 77

                #Limitando a velocidade y
                if(nova_velocidade[1] < -77):
                    nova_velocidade[1] = -77
                elif(nova_velocidade[1] > 77):
                    nova_velocidade[1] = 77

                vetor_velocidade[i] =nova_velocidade
                #6 b) - Atualizando a posicao
                nova_posicao =
update_posicao(posicao_particulas[i], vetor_velocidade[i])
                if(nova_posicao[0] < -512):
                    nova_posicao[0] = -512
                elif(nova_posicao[0] > 512):
                    nova_posicao[0] = 512

                if(nova_posicao[1] < -512):
                    nova_posicao[1] = -512
                elif(nova_posicao[1] > 512):
                    nova_posicao[1] = 512

                posicao_particulas[i] =nova_posicao

            #Salvar o gbest no array de cata iteracao
            all_gbest_iteration.append(gbest_valor)

            #7 - Condicao de terminao nao foi alcancada
            iteration = iteration + 1

    return [gbest_valor, gbest_posicao, all_gbest_iteration]
```

3.8 - Salvando os resultados em gráficos e planilhas

Neste trabalho optei por gerar os gráficos e planilhas automaticamente após a execução do algoritmo, evitando o trabalho de ter que passar tudo manualmente para outro aplicativo. Sendo assim, funções para tais foram criadas

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

def save_planilha(dicionario_dados, nome_arquivo):
    #Converto o dicionario de dados para um dataframe
    df = pd.DataFrame.from_dict(dicionario_dados, orient='index')
    df = (df.T)
    #Salvo no formato de planilhas, dado o nome do arquivo
(extensao .xlsx)
    df.to_excel(nome_arquivo)
```

3.9 - Desvio Padrão

Nas planilhas também é salvo o desvio padrão das execuções, sendo assim, segue a função para o cálculo do mesmo:

```
#Calculo do desvido padrao
def desvio_padrao(lista, media):
    result = 0
    for i in range(len(lista)):
        result += (lista[i] - media) ** 2
    #Dividindo os termos
    result = result / len(lista)

    #Tirando a raiz
    result = result ** 0.5
    return result
```

3.10 - Execução para cada quantidade de partículas e salvando resultados

Conforme foi solicitado, o algoritmo executa 10x para cada quantidade de partículas (50 e 100), além de, para cada quantidade de partículas e cada número de iterações, é necessário salvar os dados da execução. Assim, a função 'run' tem esse objetivo, recebendo a quantidade de partículas e chamando o algoritmo PSO para cada quantidade de iterações. Calculando a média, salvando o melhor resultado, salvando os gráficos, planilhas e calculando desvio padrão.

```
#Dada a quantidade de particulas, execute!
def run(qtd_particulas):

      #Valor, posicao e lista de Gbest da respectiva iteracao
      best_result_20 = [float('inf'), [0,0], []]
      media_result_20 = 0
      array_result_20 = []
      media_iteration_20 = [0] * 20

      best_result_50 = [float('inf'), [0,0], []]
      media_result_50 = 0
      array_result_50 = []
      media_iteration_50 = [0] * 50

      best_result_100 = [float('inf'), [0,0], []]
      media_result_100 = 0
      array_result_100 = []
      media_iteration_100 = [0] * 100

      qtd_rodadas = 10

      for i in range(qtd_rodadas):
            result_20 = run_pso(20, qtd_particulas)
            result_50 = run_pso(50, qtd_particulas)
            result_100 = run_pso(100, qtd_particulas)

            array_result_20.append(result_20[0])
            array_result_50.append(result_50[0])
            array_result_100.append(result_100[0])

            #Pegando os melhores resultados
```

```
            if(result_20[0] < best_result_20[0]):
                  best_result_20 = result_20

            if(result_50[0] < best_result_50[0]):
                  best_result_50 = result_50

            if(result_100[0] < best_result_100[0]):
                  best_result_100 = result_100

            #Soma para realizar a media
            media_result_20 += result_20[0]
            media_result_50 += result_50[0]
            media_result_100 += result_100[0]

            #Soma para realizar a media das iteracoes
            for i in range(20):
                  media_iteration_20[i] += result_20[2][i]

            for i in range(50):
                  media_iteration_50[i] += result_50[2][i]

            for i in range(100):
                  media_iteration_100[i] += result_100[2][i]

      #Realizando a media (valor)
      media_result_20 = media_result_20 / qtd_rodadas
      media_result_50 = media_result_50 / qtd_rodadas
      media_result_100 = media_result_100 / qtd_rodadas

      #Desvio padrao
      desvio_20 = desvio_padrao(array_result_20, media_result_20)
      desvio_50 = desvio_padrao(array_result_50, media_result_50)
      desvio_100 = desvio_padrao(array_result_100,
media_result_100)

      #Realizando a media (iteracao)
      for i in range(20):
            media_iteration_20[i] = media_iteration_20[i] /
qtd_rodadas

      for i in range(50):
            media_iteration_50[i] = media_iteration_50[i] /
```

```
qtd_rodadas

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

      #Salvando os graficos
      legenda = str(qtd_particulas)  + ' particulas . 20 iteracoes'
      save_graph([x for x in range(20)], media_iteration_20,
best_result_20[2], legenda)

      legenda = str(qtd_particulas)  + ' particulas . 50 iteracoes'
      save_graph([x for x in range(50)], media_iteration_50,
best_result_50[2], legenda)

      legenda = str(qtd_particulas)  + '_particulas_100_iteracoes'
      save_graph([x for x in range(100)], media_iteration_100,
best_result_100[2], legenda)

      #Criando e salvando os valores em planilhas
      dic_20 = {}
      dic_50 = {}
      dic_100 = {}

      dic_20["resultados"] = array_result_20
      dic_20["best_result"] = [best_result_20[0]]
      dic_20["media"] = [media_result_20]
      dic_20["desvio_padrao"] = [desvio_20]

      dic_50["resultados"] = array_result_50
      dic_50["best_result"] = [best_result_50[0]]
      dic_50["media"] = [media_result_50]
      dic_50["desvio_padrao"] = [desvio_50]

      dic_100["resultados"] = array_result_100
      dic_100["best_result"] = [best_result_100[0]]
      dic_100["media"] = [media_result_100]
      dic_100["desvio_padrao"] = [desvio_100]

      nome_arquivo =
("result_planilhas/%s_particulas_20_iteracoes.xlsx"
%str(qtd_particulas))
```

```
      save_planilha(dic_20, nome_arquivo)

      nome_arquivo =
("result_planilhas/%s_particulas_50_iteracoes.xlsx"
%str(qtd_particulas))
      save_planilha(dic_50, nome_arquivo)

      nome_arquivo =
("result_planilhas/%s_particulas_100_iteracoes.xlsx"
%str(qtd_particulas))
      save_planilha(dic_100, nome_arquivo)
```

3.11 - A Main

E por fim, temos a função main, que simplesmente cria os diretórios para salvar os gráficos e planilhas e chama a função “run” para cada número de partículas.

```
def main():
    #Criando pasta "plot_graphs" se nao existir
    path = Path("plot_graphs")
    path.mkdir(exist_ok=True)

    #Criando pasta "result_planilhas" se nao existir
    path = Path("result_planilhas")
    path.mkdir(exist_ok=True)

    #Para 50 particulas
    run(50)

    #Para 100 particulas]
    run(100)
    print(">> Checar pasta: plot_graphs")
    print(">> Checar pasta: result_planilhas")
main()
```

4 - Resultados

Após a execução do algoritmo, os gráficos e planilhas são salvos em suas respectivas pastas, e assim, conseguimos ver os resultados da execução para cada quantidade de partículas (50 e 100) e para número de iterações (20, 50 e 100).

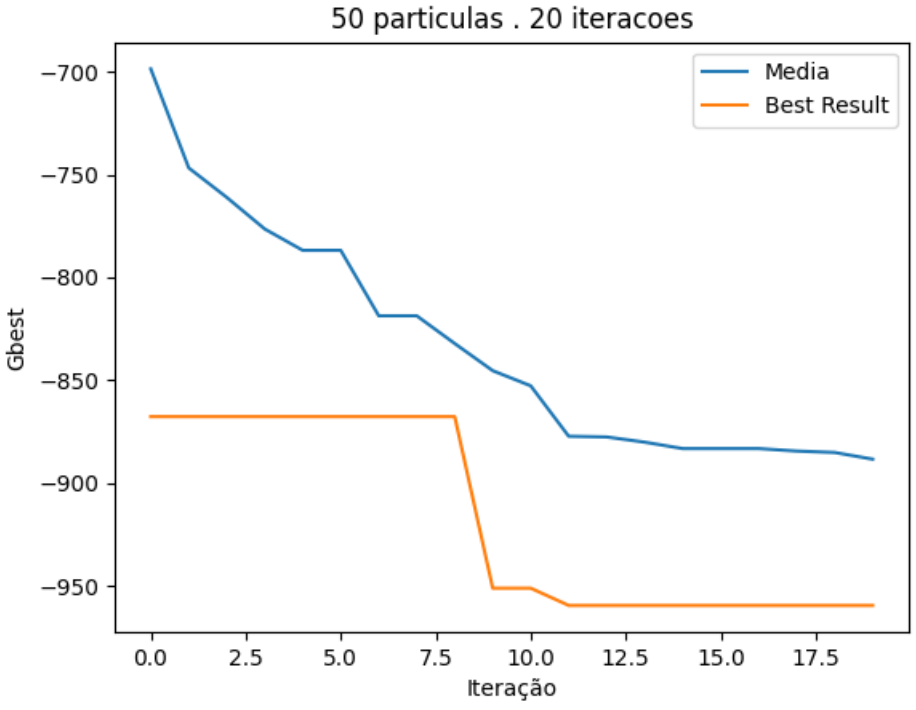
Lembrando que foram executadas 10 rodadas e partir desses números, 6 gráficos e 6 planilhas foram geradas, conforme vamos ver a seguir.

Lembrando que, os gráficos e planilhas mostrados a seguir são resultados de uma determinada execução, obviamente caso eu execute o algoritmo novamente os valores não serão exatamente iguais, mas irão convergir para um mesmo resultado.

Para melhor visualização, vou pôr lado a lado o gráfico e planilha de determinado setup de execução (quantidade de partículas e número de iterações.

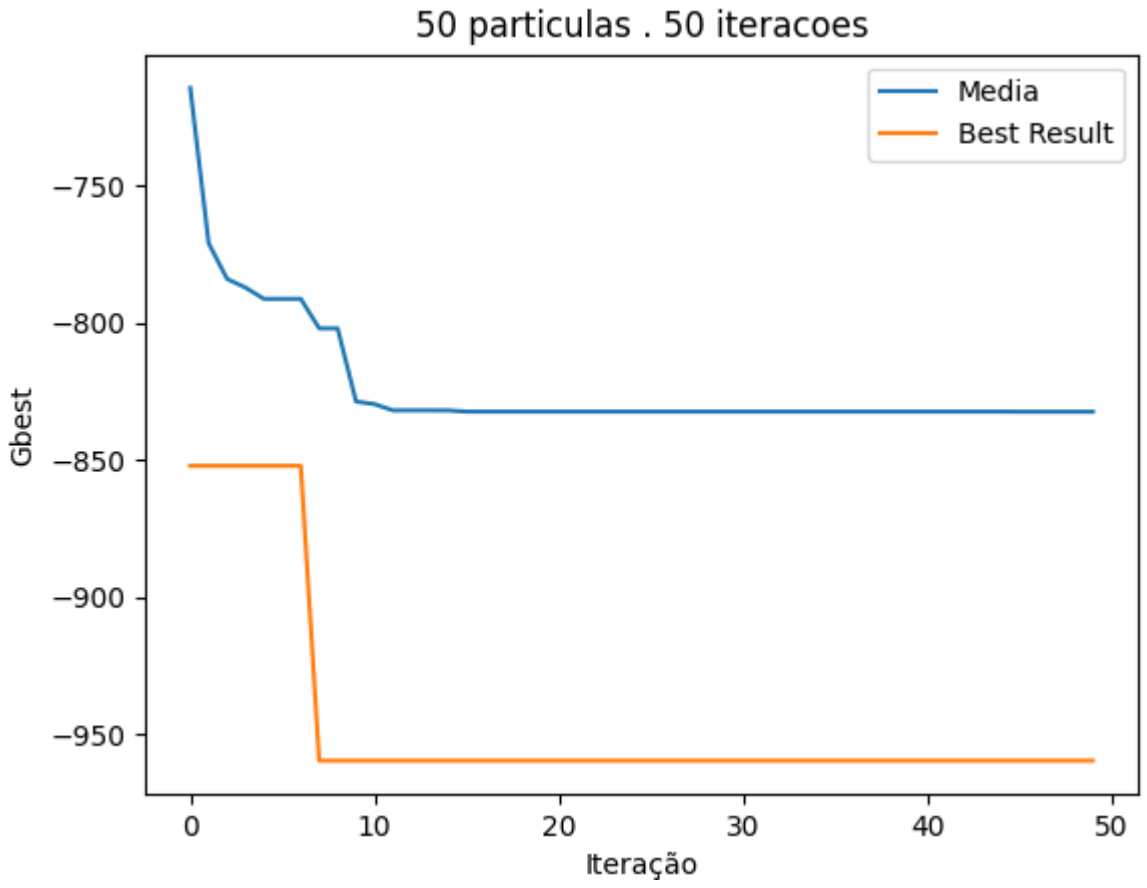
4.1 - 50 Partículas & 20 iterações

	A	B	C	D	E
1		resultados	best_result	media	desvio_padrao
2	0	-959.59	-959.59	-888.44	77.23
3	1	-856.31			
4	2	-950.32			
5	3	-709.34			
6	4	-953.76			
7	5	-959.54			
8	6	-946.59			
9	7	-858.11			
10	8	-857.95			
11	9	-832.90			



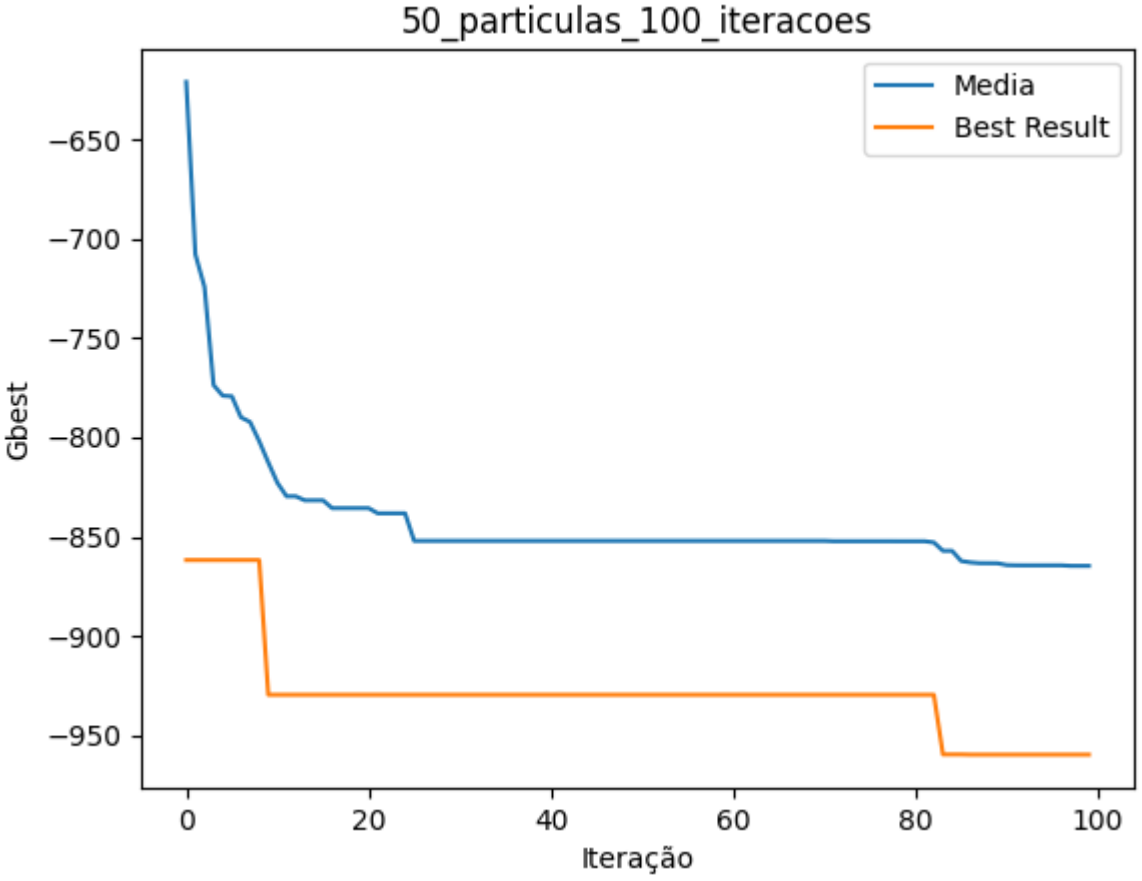
4.2 - 50 partículas & 50 iterações

	A	B	C	D	E
1		resultados	best_result	media	desvio_padra
2	0	-652.25	-959.59	-832.41	121.16
3	1	-959.26			
4	2	-959.58			
5	3	-959.59			
6	4	-850.81			
7	5	-673.46			
8	6	-707.29			
9	7	-744.10			
10	8	-858.17			
11	9	-959.59			



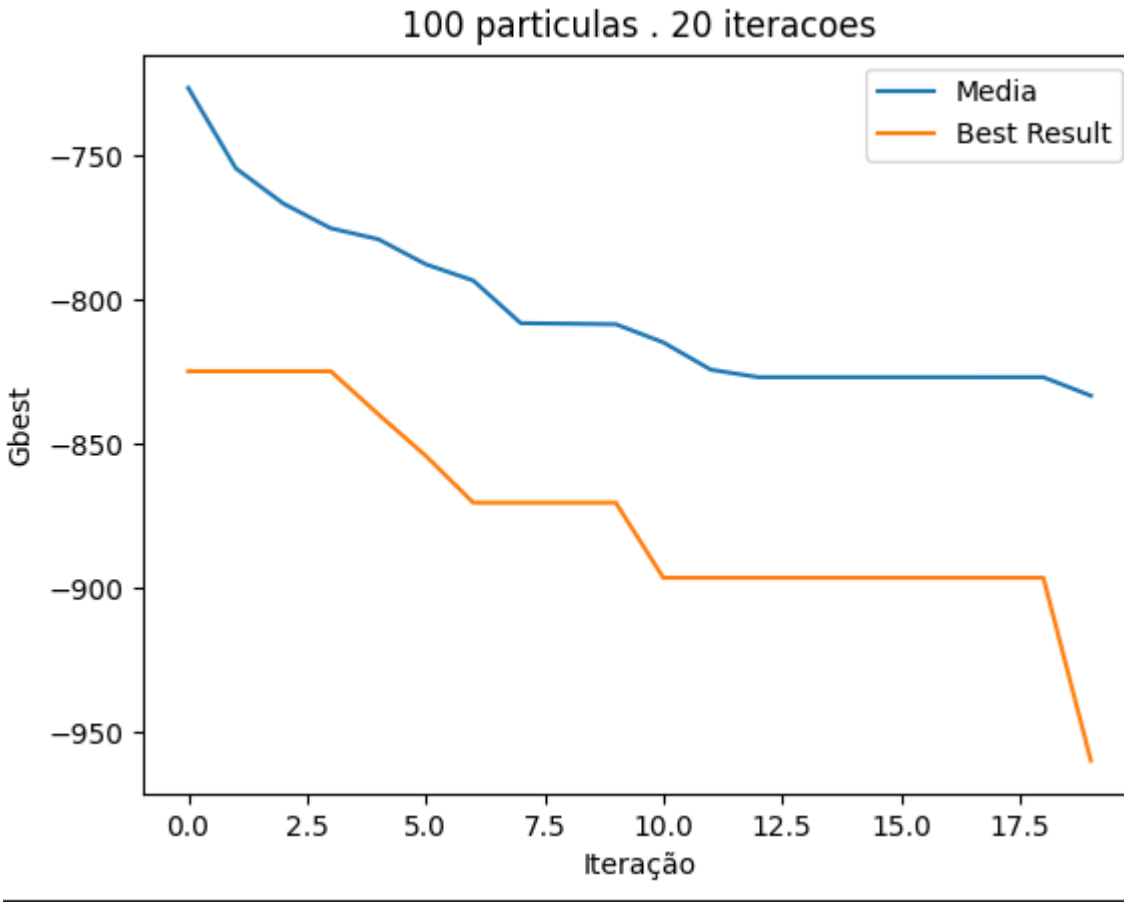
4.3 - 50 partículas & 100 iterações

	A	B	C	D	E
1		resultados	best_result	media	desvio_padrao
2	0	-709.57	-959.64	-864.64	107.96
3	1	-707.33			
4	2	-959.64			
5	3	-714.36			
6	4	-959.60			
7	5	-959.64			
8	6	-959.53			
9	7	-858.60			
10	8	-959.61			
11	9	-858.54			



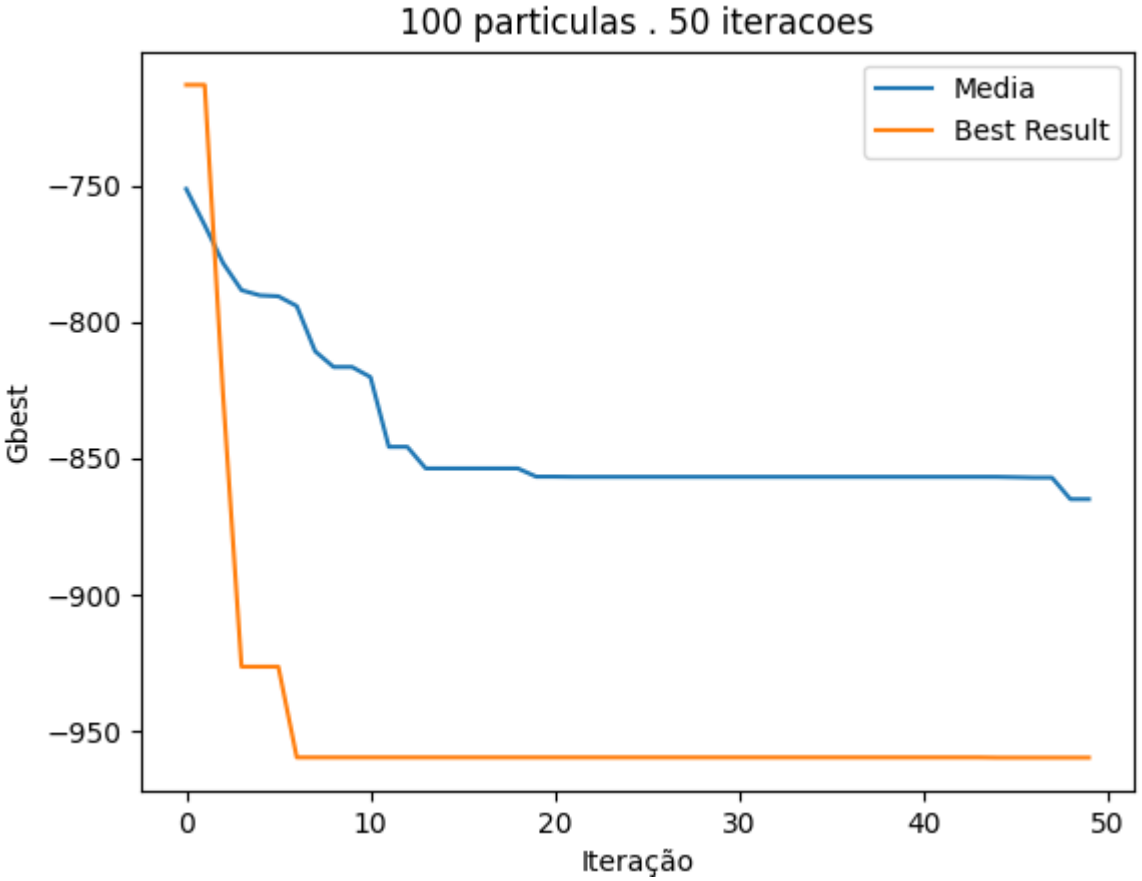
4.4 - 100 partículas & 20 iterações

	A	B	C	D	E
1		resultados	best_result	media	desvio_padrao
2	0	-931.25	-959.64	-833.25	98.85
3	1	-717.28			
4	2	-959.64			
5	3	-858.01			
6	4	-743.54			
7	5	-715.21			
8	6	-859.32			
9	7	-703.72			
10	8	-959.62			
11	9	-884.87			



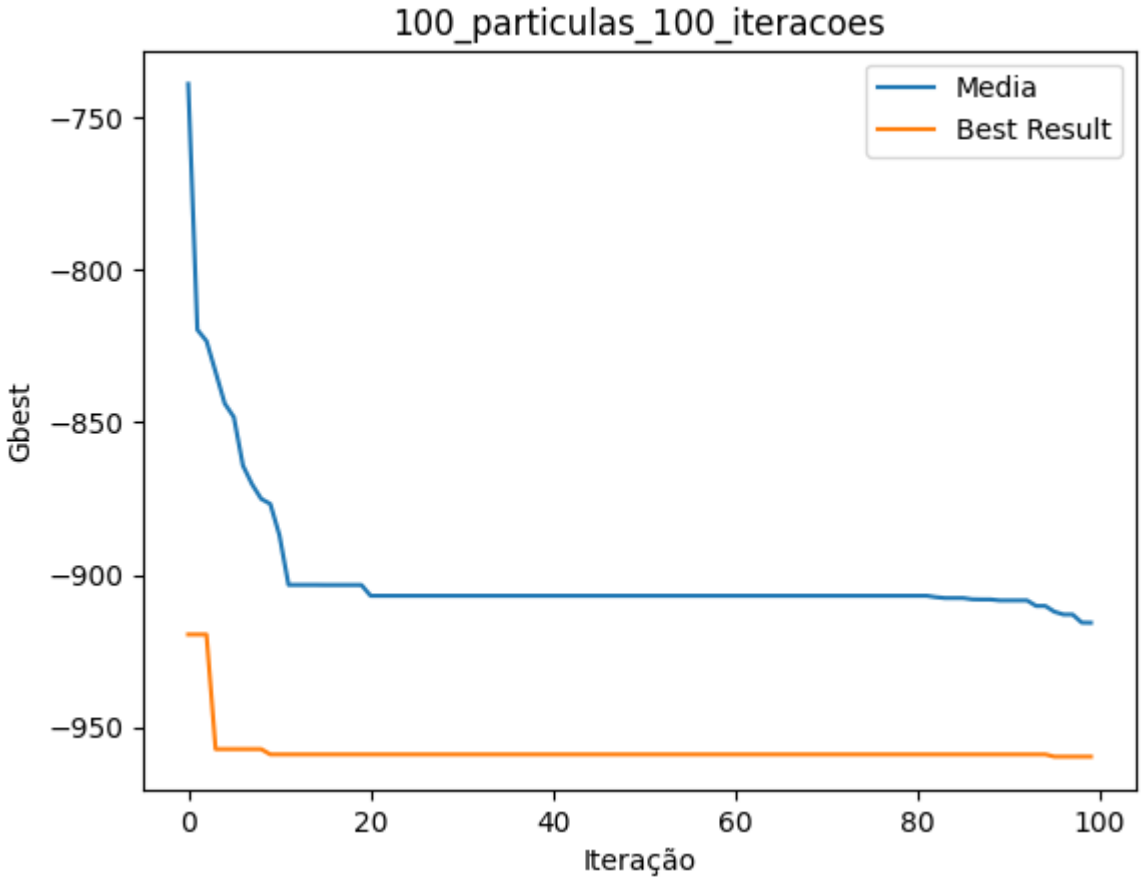
4.5 - 100 partículas & 50 iterações

	A	B	C	D	E
1		resultados	best_result	media	desvio_padrao
2	0	-869.91	-959.64	-864.96	91.30
3	1	-959.64			
4	2	-781.88			
5	3	-959.61			
6	4	-872.33			
7	5	-838.03			
8	6	-739.32			
9	7	-959.54			
10	8	-959.46			
11	9	-709.84			



4.6 - 100 partículas & 100 iterações

	A	B	C	D	E
1		resultados	best_result	media	desvio_padrao
2	0	-892.68	-959.64	-915.72	37.28
3	1	-893.19			
4	2	-856.24			
5	3	-959.51			
6	4	-959.34			
7	5	-893.70			
8	6	-959.63			
9	7	-892.27			
10	8	-959.64			
11	9	-890.97			



5 - Conclusão

Conforme mostrado na seção 4, o algoritmo alcança sem muitos problemas o objetivo (ou valores extremamente próximos do objetivo), ou seja, o ponto mínimo da função, cujo valor é -959.6407, mesmo com valores baixos para partículas e iterações (50 e 20, respectivamente).

Vemos também que conforme aumentamos o número de partículas e iterações, os valores tendem a ser mais precisos, encontrando muitas vezes o ponto mínimo exato e também, diminuindo o desvio padrão, mostrando que o algoritmo, na maioria das vezes, tem um ótimo desempenho! Lembrando que, tudo isso num intervalo de 10 rodadas. Caso aumente o número de rodadas ou até mesmo a quantidade de iterações e partículas, resultados ainda mais precisos serão entregues, visto que, uma quantidade maior de informação entre as partículas serão trocadas (a posição de cada uma no domínio).