

EEE882 – Análise de hiper-parâmetros no PSO

Lucas Jorge Caldeira Carvalho, *Discente, UFMG*

I PSO POR MELHOR GLOBAL

I.A IMPLEMENTAÇÃO COMPUTACIONAL

A.1 Visão geral do projeto

O algoritmo PSO por melhor global, também conhecido como PSO gbest, implementado neste trabalho é composto por:

1. Arquivo *pso*: arquivo que contém a função principal (main).
2. Arquivo *particula*: contém a classe *Particula*, responsável por armazenar métodos e atributos das partículas do enxame.
3. Arquivo *funcao_custo*: contém as duas funções benchmark escolhidas para este trabalho.

A.2 Definição dos parâmetros

Primeiramente, definiu-se todos os parâmetros do algoritmo, tantos os parâmetros inerentes ao PSO quanto os parâmetros do problema.

```
parametro_constricao = 1 #Parâmetro de construção
w = 1 #Inércia
c1 = 1.496180 #Coeficiente de aceleração cognitivo
c2 = 1.496180 #Coeficiente de aceleração social
dimensoes = 10 #Dimensão do problema
iteracoes = 5000 #Número de iterações desejadas
tamanho_populacao = 20 #Tamanho da população (tamanho do enxame - swarm)
espaco_busca = [-100,100] #Espaço de busca do algoritmo
quantidade_execucoes = 31 #Quantidade de testes do algoritmo
```

Figura 1. Definição dos parâmetros

A.3 Inicialização das partículas

O enxame é composto por um conjunto de partículas. Neste trabalho, as partículas são inicializadas de maneira uniforme dentro do espaço de busca.

```
class Particula:
    def __init__(self, dimensoes, espaco_busca):
        self.posicao=[] # posição da partícula
        self.b_posicao = [] # melhor posição da partícula
        self.velocidade=[] # velocidade da partícula
        self.fitness = math.inf # Avaliação da posição
        self.b_fitness = math.inf # Melhor fitness encontrado

        #Inicialização uniforme da população dentro do espaço de busca
        for i in range(dimensoes):
            self.velocidade.append(random.uniform(-1,1))
            self.posicao.append(random.uniform(espaco_busca[0], espaco_busca[1]))
            self.b_posicao.append(self.posicao)
```

Figura 2. Inicialização das partículas

A.4 Cálculo do fitness das partículas

Após a inicialização do enxame, aqui chamado de população, é feito o cálculo do fitness de cada partícula. Este valor é calculado usando as funções benchmark escolhidas.

Posteriormente, é verificado se a nova posição da partícula k é melhor que a melhor posição encontrada até o momento pela partícula k . Caso seja, a melhor posição encontrada até o momento pela partícula k ($b_posicao$) é atualizada.

O atributo $b_fitness$ serve para não termos que calcular o fitness da partícula sempre que necessitarmos ao longo daquela iteração. Desta forma, após se calcular uma vez, esse valor é armazenado.

```
# Checagem do fitness das partículas
def calcular_fitness(self):
    self.fitness=funcao_custo(self.posicao)

    # Checa se a posição atual é a melhor
    if self.fitness < self.b_fitness:
        self.b_posicao=self.posicao
        self.b_fitness=self.fitness
```

Figura 3. Cálculo do fitness das partículas

A.5 Atualização da melhor posição dentro do enxame

Com os fitness de cada partícula calculado, é possível atuar a melhor posição dentro do enxame ($g_best_posicao$).

A variável $b_best_fitness$ serve para não termos que calcular o fitness da melhor posição global sempre que necessitarmos ao longo daquela iteração. Desta forma, após se calcular uma vez, esse valor é armazenado.

```
for j in range(tamanho_populacao):
    populacao[j].calcular_fitness()

    # Saber se a partícula é a melhor global
    if populacao[j].fitness < g_best_fitness:
        g_best_posicao=list(populacao[j].posicao)
        g_best_fitness= populacao[j].fitness
```

Figura 4. Atualização da melhor posição dentro do enxame

A.6 Atualização da velocidade das partículas

A velocidade das partículas é atualizada de acordo com a equação dada no enunciado do trabalho.

```
# Atualiza a velocidade da partícula
def atualizar_velocidade(self, g_best_posicao, dimensoes, parametro_constricao, w, c1, c2):
    for i in range(dimensoes):
        r1=random.random()
        r2=random.random()

        v1=w*self.velocidade[i]
        v2=c1*r1*(self.b_posicao[i]-self.posicao[i])
        v3=c2*r2*(g_best_posicao[i]-self.posicao[i])
        self.velocidade[i]=parametro_constricao*(v1+v2+v3)
```

Figura 5. Atualização da velocidade

A.7 Atualização da posição das partículas

A posição das partículas é atualizada somando-se a velocidade calculada com a posição anterior da partícula.

Caso a nova posição da partícula não esteja dentro do espaço de busca, a partícula tem sua posição atualizada para a borda do espaço.

```
#Atualiza posição da partícula
def atualizar_posicao(self, dimensoes, espaco_busca):
    for i in range(dimensoes):
        self.posicao[i]=self.posicao[i] + self.velocidade[i]

        # Caso necessário, ajustar a posição para dentro dos limites de D
        if self.posicao[i] > espaco_busca[1]:
            self.posicao[i]=espaco_busca[1]

        if self.posicao[i] < espaco_busca[0]:
            self.posicao[i]=espaco_busca[0]
```

Figura 6. Atualização das posição das partículas

A.8 Saída do programa

A melhor posição do enxame de cada uma das 31 execuções é salva no vetor *melhores_solucoes_execucoes*. Sendo assim, é possível calcular a média e o desvio padrão destes valores.

```
#Cálculo do valor da média dos melhores valores obtidos nas execuções
soma = 0
for i in range(quantidade_execucoes):
    soma = soma + melhores_solucoes_execucao[i]
media = soma/quantidade_execucoes

#Cálculo do desvio padrão
num = 0
for i in range(quantidade_execucoes):
    num = (melhores_solucoes_execucao[i]-media)**2
desvio_padrao = math.sqrt(num/quantidade_execucoes)

#Mostrar resultados
print ("\nXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\n")
print ("Melhores solucoes: " + str(melhores_solucoes_execucao) + "\n")
print ("Média: " + str(media) + "\n")
print ("Desvio padrão: " + str(desvio_padrao) + "\n")
print ("XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\n")
```

Figura 7. Saída do PSO

II PSO POR MELHOR LOCAL

II.A IMPLEMENTAÇÃO COMPUTACIONAL

O PSO por melhor local, também conhecido como PSO lbest, é um caso particular do PSO por melhor global, a diferença entre esse algoritmo e o algoritmo por melhor global é que na equação de velocidade no PSO por melhor local é atualizado o valor da melhor posição da vizinhança, ao invés do melhor global.

Sendo assim, abordaremos a seguir apenas as diferenças entre os algoritmos.

Neste trabalho, foi usada uma topologia de rede social conhecida por topologia em anel.

A.1 Atualização do lbest

Foi criado um atributo denominado *l_best* dentro da classe *Particulas* que armazena qual a melhor posição da vizinhança em relação ao objeto (partícula *k*). Para isso, é consultado qual a melhor posição dentre as melhores posições visitadas pelas partículas com índice *k*, *k-1* e *k+1* (partícula imediatamente anterior e posterior em relação à partícula *k*) dentro do vetor *população* (enxame).

É feita uma verificação para sabermos se estamos tratando da primeira partícula do enxame, que terá como vizinhos a segunda e última partícula, e para saber se estamos tratando da última partícula do enxame, que terá como vizinhos a penúltima e a primeira.

```
def atualizar_lbest(self, populacao, tamanho_populacao, j):
    pbest_aux = []
    fitness_pbest_aux = []
    # Melhor da vizinhança
    if j == 0:
        fitness_pbest_aux.append(populacao[tamanho_populacao-1].b_fitness)
        fitness_pbest_aux.append(self.b_fitness)
        fitness_pbest_aux.append(populacao[j+1].b_fitness)
        pbest_aux.append(populacao[tamanho_populacao-1].posicao)
        pbest_aux.append(self.posicao)
        pbest_aux.append(populacao[j+1].posicao)
        min_index = fitness_pbest_aux.index(min(fitness_pbest_aux))
        self.l_best = pbest_aux[min_index]
    elif j == tamanho_populacao - 1:
        fitness_pbest_aux.append(populacao[j-1].b_fitness)
        fitness_pbest_aux.append(self.b_fitness)
        fitness_pbest_aux.append(populacao[0].b_fitness)
        pbest_aux.append(populacao[j-1].posicao)
        pbest_aux.append(self.posicao)
        pbest_aux.append(populacao[0].posicao)
        min_index = pbest_aux.index(min(pbest_aux))
        self.l_best = pbest_aux[min_index]
    else:
        fitness_pbest_aux.append(populacao[j-1].b_fitness)
        fitness_pbest_aux.append(self.b_fitness)
        fitness_pbest_aux.append(populacao[j+1].b_fitness)
        pbest_aux.append(populacao[j-1].posicao)
        pbest_aux.append(self.posicao)
        pbest_aux.append(populacao[j+1].posicao)
        min_index = pbest_aux.index(min(pbest_aux))
        self.l_best = pbest_aux[min_index]
```

Figura 8. Atualização do melhor da vizinhança

A.2 Atualização da velocidade das partículas

Conforme descrito anteriormente, a velocidade das partículas são atualizadas utilizando o valor do melhor da vizinhança, e não o melhor global.

```
# Atualiza a velocidade da partícula
def atualizar_velocidade(self, dimensoes, parametro_constricao, w, c1, c2):
    for i in range(dimensoes):
        r1=random.random()
        r2=random.random()

        v1=w*self.velocidade[i]
        v2=c1*r1*(self.b_posicao[i]-self.posicao[i])
        v3=c2*r2*(self.l_best[i]-self.posicao[i])
        self.velocidade[i]=parametro_constricao*(v1+v2+v3)
```

Figura 9. Atualização da velocidade das partículas

II.B FUNÇÕES BENCHMARK ESCOLHIDAS

A função unimodal escolhida para este trabalho foi a *função esfera* e a função multimodal foi a *função Rastrigin*.

```
#Função de custo esfera
def funcao_custo(x):
    f = 0
    D = len(x)
    for i in range(D):
        f+=x[i]**2
    return f

#Função de custo Rastrigin
def funcao_custo(x):
    f = 0
    D = len(x)
    for i in range(D):
        f = f + (x[i]**2 + 10 * np.cos(2 * math.pi * x[i]) + 10)
    return f
```

Figura 10. Funções benchmark escolhidas

II.C RESULTADOS E DISCUSSÕES

C.1 Função Rastrigin

A função de Rastrigin é uma função não convexa, multimodal e separável. Possui vários ótimos locais arranjados em um grafo regular (regular lattice), com o ótimo global localizado no ponto $o = (0, \dots, 0)$. Constitui um problema razoavelmente difícil devido à grande quantidade de ótimos locais [1].

Na literatura, intervalos típicos testados para o espaço de busca giram em torno de $[-5.12, 5.12]$. Podemos notar que o intervalo de busca proposto por esse trabalho é muito maior, um dos fatores que fizeram com os resultados dos algoritmos não ficassem próximos ao ótimo. Usando intervalos menores, notou-se uma grande melhora na melhor solução encontrada, encontrando-se valor muito próximos a zero.

Outra forma observada de melhorar a solução dos algoritmos é diminuindo o número de dimensões, o que torna menor a quantidade combinatória possível para representar as partículas.

O peso de inércia tem o papel de regular a velocidade da partícula, evitando a explosão de velocidade. Trata-se de um mecanismo que controla o balanceamento entre as capacidades de exploração global e exploração local do enxame. Altos valores de w facilitam a exploração global, enquanto pequenos valores promovem exploração local [2].

O PSO com fator de constrição é equivalente ao PSO com peso de inércia. Ambas abordagens têm o objetivo de promover o balanceamento entre as capacidades de exploração global e local do enxame [2].

Para a função Rastrigin as melhores soluções após as 31 execuções e que têm os menores desvios padrão são encontradas na ausência do parâmetro de constrição ($\chi=1$) com a presença do fator de inércia ($0 < \omega < 1$) e na presença do parâmetro de constrição ($\chi=1$) com a ausência do fator de inércia ($\omega = 1$). Na ausência de ambos ou na presença de ambos, os resultados foram piores.

Quando χ é próximo de zero, há rápida convergência. Por outro lado, se χ é próximo de um, a convergência é mais lenta [2]. Isso pode ser observado na tabela 1 e 2.

Notamos que o algoritmo por melhor global apresentou, para os parâmetros testados, um resultado expressivamente melhor que o algoritmo por melhor local. Isso foi diferente do esperado, já que normalmente algoritmos por melhor local apresentam um melhor desempenho em problemas multimodais por explorarem melhor o espaço de busca não se prendendo a mínimos locais [3].

Além disso, o algoritmo de ótimo global convergiu mais rapidamente que o algoritmo de ótimo local, em consonância com a literatura.

Tabela 1. Resultados para o algoritmo gbest

	Melhor solução	Média	Desvio padrão	Tempo das 31 execução (s)
$\chi=1, \omega = 1$	1486.5513	2715.8488	129.5536	184.0743
$\chi=1, 0 < \omega < 1$	42.3464	150.6615	0.5666	100.5357
$0 < \chi < 1, \omega = 1$	73.9017	484.1522	33.4367	89.0736
$0 < \chi < 1, 0 < \omega < 1$	310.1990	1736.4760	59.2977	75.3036

Tabela 2. Resultados para o algoritmo lbest

	Melhor solução	Média	Desvio padrão	Tempo das 31 execução (s)
$\chi=1, \omega = 1$	5680.6526	7122.3428	258.9352	200.8508
$\chi=1, 0 < \omega < 1$	5997.7124	7429.6878	257.1903	153.3514
$0 < \chi < 1, \omega = 1$	2925.4246	3953.0445	184.5659	84.0327
$0 < \chi < 1, 0 < \omega < 1$	3696.7247	3855.5600	28.5276	126.8385

Abaixo se encontram os vetores contendo os melhores valores obtidos em cada uma das 31 execuções dos algoritmos.

```
Melhores solucoes: [3181.2711790743488, 2701.940217653384, 2326.8327626799937, 4246.185475325897, 1749.9072374802583, 1580.9899249764158, 1673.745538446787, 2287.2296616029703, 2684.2422423297953, 3681.66164375922, 3467.5369317334103, 2884.0316427156117, 2487.42488117267, 1544.7004750464632, 3589.2715406671173, 2095.1990785203916, 4076.8932945519928, 3340.739963695417, 3011.590282636599, 2223.7633327953254, 1850.2213482923257, 3310.09076315289376, 1616.4177320410033, 3125.8542827640645, 3793.8931669921664, 3446.4341283168064, 3296.1763372821797, 1922.0307028520142, 2071.313386275276, 1486.551387187163, 3437.1729630335158]
```

Figura 11: Vetor que contém os melhores das 31 execuções para $\chi=1, \omega = 1$ para o algoritmo gbest

```
Melhores solucoes: [66.32158151703275, 78.17323060206542, 97.45926398510954, 62.18631160022768, 73.01857774476748, 183.56011199216275, 239.64075028645414, 126.22080809415233, 88.10373411817368, 188.07084796853223, 143.8206465058868, 245.29579337165293, 289.1484239095924, 157.7298029606972, 148.14498498898166, 36.36089711959727, 203.46445914664525, 116.01499423040005, 207.4452398096542, 314.3627775993321, 187.54661796194446, 42.346413112033645, 145.76236507086415, 145.8381381550346, 143.77302387316325, 175.7427106855853, 111.96924185374317, 125.78762905042258, 193.52294593564153, 125.86024142579666, 153.8164975201542]
```

Figura 12: Vetor que contém os melhores das 31 execuções para $\chi=1, 0 < \omega < 1$ para o algoritmo gbest

```
Melhores solucoes: [316.6601549947702, 132.47642866122797, 898.1069089282872, 727.0350921929227, 353.0887913555769, 97.0806525951368, 280.23393838337523, 177.72012555260852, 332.9370984404789, 800.1192895218085, 411.32561548163164, 1079.0202294053988, 256.05706087121706, 460.68556044942295, 293.3139357182861, 314.1060141186044, 441.32341695278294, 398.18098060104603, 207.86455749828536, 320.7446419774122, 134.88283567812175, 360.8796918298188, 297.3785839201019, 431.14884277712076, 1400.5119650680974, 1173.7978094560187, 777.200024849738, 1326.7042177492897, 620.6913363198211, 154.0077260875553, 33.43670994584299]
```

Figura 13: Vetor que contém os melhores das 31 execuções para $0 < \chi < 1, \omega = 1$ para o algoritmo gbest

```
Melhores solucoes: [1204.0530391408725, 1593.823822339753, 4399.008561363284, 310.19906137889246, 1486.2283973238168, 2885.2485411480066, 2049.4905118089787, 1284.93585317106, 398.6416571921398, 1893.6506231730832, 3032.8593091858554, 3238.73814925043, 891.231794602386, 1795.2334142647976, 2780.8087403170425, 756.4752412498264, 3072.5789372301294, 1281.5654379662267, 2100.4127948424105, 407.6805717863452, 1076.957866667692, 709.4563010925971, 2376.1740309410498, 877.1252428544503, 2924.43159973795, 2018.455413040016, 1279.523674822584, 1970.456730885618, 1007.7501238293742, 650.9281042017185, 2066.632200192067]
```

Figura 14: Vetor que contém os melhores das 31 execuções para $0 < \chi < 1, 0 < \omega < 1$ para o algoritmo gbest


```
Melhores solucoes: [13131.838576959296, 10208.462716642282, 10208.462716642282, 10208.462716642282,
10208.462716642282, 10208.462716642282, 10208.462716642282, 6605.046698002522, 6102.996613067126,
6102.996613067126, 6102.996613067126, 6102.996613067126, 6102.996613067126, 6102.996613067126,
6102.996613067126, 6102.996613067126, 6102.996613067126, 6102.996613067126, 6102.996613067126,
6102.996613067126, 6102.996613067126, 6102.996613067126, 6102.996613067126, 6102.996613067126,
6102.996613067126, 6102.996613067126, 6102.996613067126, 6102.996613067126, 6102.996613067126,
6102.996613067126, 6102.996613067126, 6102.996613067126, 6102.996613067126, 6102.996613067126,
6102.996613067126, 6102.996613067126]
```

Figura 24: Vetor que contém os melhores das 31 execuções para $\chi=1$, $0 < \omega < 1$ para o algoritmo lbtest

```
Melhores solucoes: [8027.830072954427, 8027.830072954427, 8027.830072954427, 8027.830072954427, 8027.830072954427,
8027.830072954427, 8027.830072954427, 8027.830072954427, 8027.830072954427, 8027.830072954427,
8027.830072954427, 8027.830072954427, 8027.830072954427, 8027.830072954427, 8027.830072954427,
8027.830072954427, 8027.830072954427, 8027.830072954427, 8027.830072954427, 8027.830072954427,
8027.830072954427, 8027.830072954427, 8027.830072954427, 8027.830072954427, 8027.830072954427,
8027.830072954427, 8027.830072954427, 8027.830072954427, 8027.830072954427, 8027.830072954427,
8027.830072954427, 8027.830072954427, 8027.830072954427, 8027.830072954427, 8027.830072954427,
8027.830072954427, 8027.830072954427]
```

Figura 25: Vetor que contém os melhores das 31 execuções para $0 < \chi < 1$, $\omega = 1$ para o algoritmo lbtest

```
Melhores solucoes: [5583.513617108425, 5171.0113860891015, 5171.0113860891015, 5171.0113860891015, 5171.0113860891015,
5171.0113860891015, 5171.0113860891015, 5171.0113860891015, 5171.0113860891015, 5171.0113860891015,
5171.0113860891015, 5171.0113860891015, 5171.0113860891015, 5171.0113860891015, 5171.0113860891015,
5171.0113860891015, 5171.0113860891015, 5171.0113860891015, 5171.0113860891015, 5171.0113860891015,
5171.0113860891015, 5171.0113860891015, 5171.0113860891015, 5171.0113860891015, 5171.0113860891015,
5171.0113860891015, 5171.0113860891015, 5171.0113860891015, 5171.0113860891015, 5171.0113860891015,
5171.0113860891015, 5171.0113860891015, 5171.0113860891015, 5171.0113860891015, 5171.0113860891015,
5171.0113860891015, 5171.0113860891015]
```

Figura 26: Vetor que contém os melhores das 31 execuções para $0 < \chi < 1$, $0 < \omega < 1$ para o algoritmo lbtest

REFERENCES

- 1 M. R. Kohler, "PSO+: Algoritmo com Base em Enxame de Partículas para Problemas com Restrições Lineares e Não Lineares", tese de doutorado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da PUC-Rio, Rio de Janeiro, 2017.
- 2 S. A. M. Arruda, "Aplicação da otimização por enxame de partículas com topologia 'multi-ring' na estimação de parâmetros de linhas de transmissão", dissertação apresentada ao Programa de Pós-Graduação da Escola de Engenharia Elétrica, Mecânica e Computação da Universidade Federal de Goiás, Goiânia, 2015.
- 3 B. D. Agostini, "Uma Análise Experimental de Abordagens Topológicas Aplicadas ao Problema do Caixeiro-Viajante Através de Otimização por Nuvem de Partículas", dissertação apresentada ao Programa de Pós-Graduação em Métodos Numéricos em Engenharia da Universidade Federal do Paraná, Curitiba, 2015.