Algoritmos de Otimização Bioinspirados

Allan Moreira de Carvalho
Centro de Engenharia e Ciências Sociais Aplicadas, Universidade Federal do ABC

Santo André, 15 de março de 2023

Resumo

Nesse trabalho, os algorítimos de otimização beados em meta heurísticas bioinspiradas *Particle Swarm Optimization (PSO)*, *Flower Pollination Algorithm (FPA)*, *Symbiotic Organisms Search (SOS)* e *Grey Wolf Optimizer (GWO)* têm a sua performance estatística aferida em um conjunto de seis funções de teste. Os resultados apontam o algoritmo *SOS* como o mais robusto. Uma breve descrição da meta heurística *GWO* é realizada e seus resultados estatísticos são comparados com os demais algoritmos. As implementações desse trabalho estão disponíveis no repositório https://github.com/properallan/ENE300/.

Conteúdo

1 Introdução

A *Inteligência Computacional (IC)* é um ramo da *Inteligência Artificial (IA)* que propõe *metaheurísticas* inspiradas na natureza na resolução de problemas dificilmente tratáveis pelos métodos tradicionais baseados em soluções analíticas ou ainda, métodos numéricos, baseadas em gradiente. Nesse trabalho, será feita uma análise de convergência dos seguintes algoritmos biospirados:

- *Particle Swarm Optimization (PSO)*
- *Flower Pollination Algorithm (FPA)*
- *Symbiotic Organisms Search (SOS)*

Tabela 1: Conjunto de Funções Objetivo. * O valor de mínimo da função `regularized_ts` foi aquele encontrado pelo processo de otimização, as demais função são tipicamente utilizadas para avaliar algoritmos de otimização Surjanovic and Bingham; Yang (2010).

Nome	Função Objetivo	Domínio	Mínimo
`easom`	$f(\mathbf{x}) = -\cos x_1 \cos x_2 e^{\left(-(x_1-\pi)^2-(x_2-\pi)^2\right)}$	$x_i \in [-100, 100]$	-1.0000
`eggholder`	$f(\mathbf{x}) = -(x_2+47)\sin\left(\sqrt{\left\lvert x_2+\frac{x_1}{2}+47\right\rvert}\right) - x_1\sin\left(\sqrt{\lvert x_1-(x_2+47)\rvert}\right)$	$x_i \in [-512, 512]$	-959.6400
`griewank`	$f(\mathbf{x}) = \sum_{i=1}^{2}\frac{x_i^2}{4000} - \sum_{i=1}^{2}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	$x_i \in [-600, 600]$	0.0000
`shubert`	$f(\mathbf{x}) = \left(\sum_{i=1}^{5} i\cos\left((i+1)x_1+i\right)\right)\left(\sum_{i=1}^{5} i\cos\left((i+1)x_2+i\right)\right)$	$x_i \in [-10, 10]$	-186.7309
`sixhump`	$f(\mathbf{x}) = \left(4-2.1x_1^2+\frac{x_1^4}{3}\right)x_1^2 + x_1x_2 + \left(-4+4x_2^2\right)x_2$	$x_1 \in [-3, 3]$ $x_2 \in [-2, 2]$	-1.0316
`regularized_ts`	$f(\mathbf{x}) = \sum_{i=1}^{3}(a_i+b_ix_i+c_ix_i^2) + \alpha(\sum_{i=1}3x_i - 550)^2$	$x_1 \in [100, 196]$ $x_2 \in [50, 114]$ $x_3 \in [200, 332]$	93.6891^*

- *Grey Wolf Optimizer (GWO)*

A performance dos algoritmos foi avaliada quanto sua capacidade de minimizar um conjunto de seis funções objetivo, cujos domínios computacionais e equacionamento são apresentados na Tabela 1. A função objetivo `regularized_ts` representa um problema de otimização onde deseja-se minimizar os custos de operação de um sistema termoelétrico.

2 Análise Estatística (PSO, FPA, SOS)

Para manter a isonomia do resultados, todos os algoritmos foram testados 20 vezes para cada uma das funções objetivo, utilizando sempre uma população com o mesmo número de 80 indivíduos e uma quantidade máximo de 200 avaliações da função objetivo como critério de parada. Os resultados de convergência são apresentados em função do número de iterações. No caso da meta heurística SOS, cada iteração executa 4 avaliações da função objetivo e por esse motivo as curvas de convergência possuem um número total de iterações menor que os demais.

O algoritmo PSO requer outros hiper parâmetros além do critério de parada e tamanho da população. São eles, uma limitação para os vetores de velocidade, escolhido entre $[-1, 1]$, uma função para os pesos de inércia, nesse caso foi utilizada uma distribuição randômica, e também coeficientes de aceleração individuais, fixados em $c_1 = 1$ e $c_2 = 1.5$.

O algoritmo FPA requer um parâmetro de peso que controla a permuta entre um processo de polinização local e um processo global, foi escolhido o valor $p = 0.75$. O algoritmo SOS não requer outros hiper parâmetros.

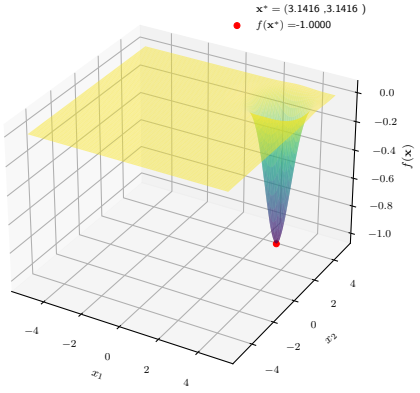
As Figura de 2 à 7 apresentam na coluna da esquerda um gráfico de dispersão dos valores mínimos obtividos nas 20 rodadas para cada uma das funções objetivo testadas, e na colunda da direita, gráficos de convergência ao longo das iterações mostrando o intervalo de confiância de 95% em torno da tendência central.

3 Grey Wolf Optimizer (GWO)

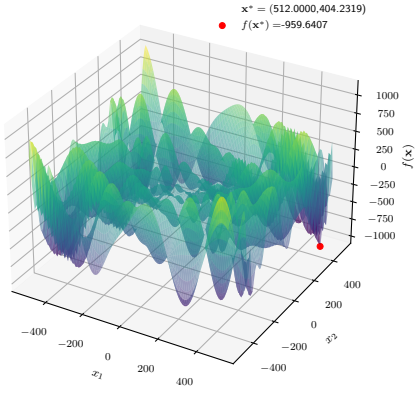
O algoritmo *GWO* Mirjalili et al. (2014) é inspirado na organização social hierárquica e estratégia predatória dos lobos cinzentos (*Canis lupus*). A organização social obedece uma hierarquia rígida de dominação social, no topo estão os líderes, denominados α. Logo abaixo, estão os indivíduos β, ainda que subordinados aos α, possuem dominância sobre os indivíduos situados logo abaixo na cadeia, dessa

Figura 1: Conjunto de Funções Objetivo utilizadas.

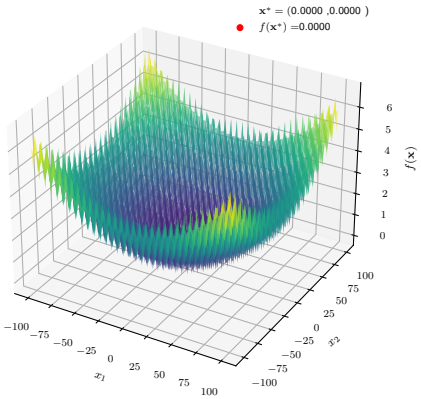
(a) `easom`



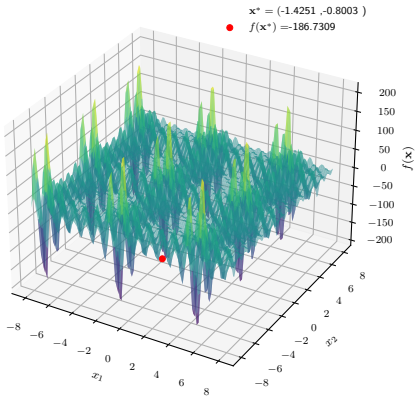
(b) `eggholder`



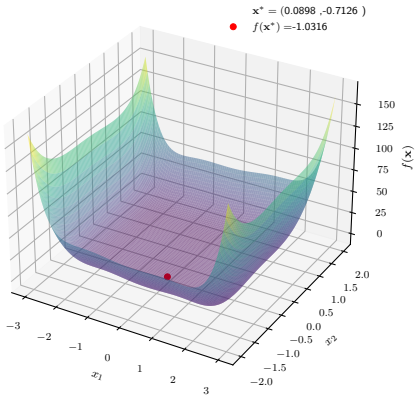
(c) `griewank`



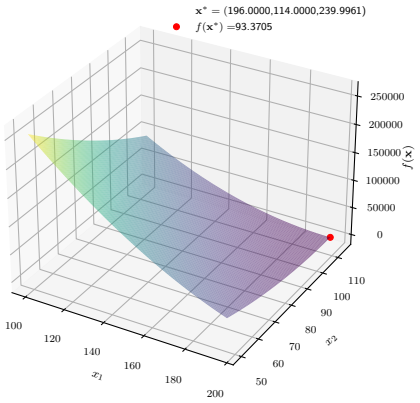
(d) `shubert`



(e) `sixhump`



(f) `regularized_ts`



forma os β podem auxiliar os α nas tomadas de decisões e gerenciamento do grupo. Logo abaixo estão os indivíduos δ e por último, o bode expiatório do grupo, está a casta dos ω.

O algoritmo também baseia-se na estratégia de caça dos lobos, que consiste em quatro etapas: *cercar a presa*; *caçar* ; *atacar*; *procurar*. O *cercamento* da presa é modelado pelas equações **??**

Figura 2: Função objetivo `easom`.

(a) Dispersão. (b) Convergência.

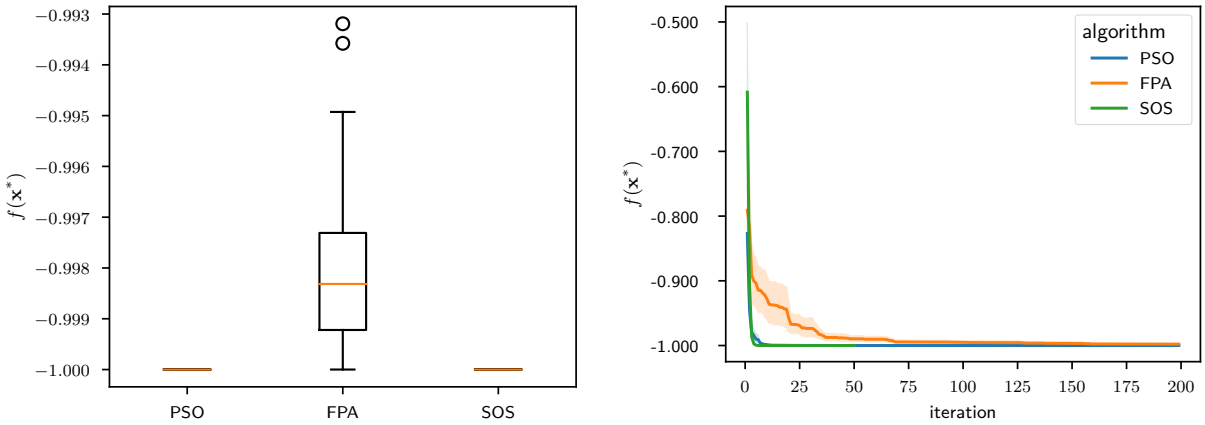
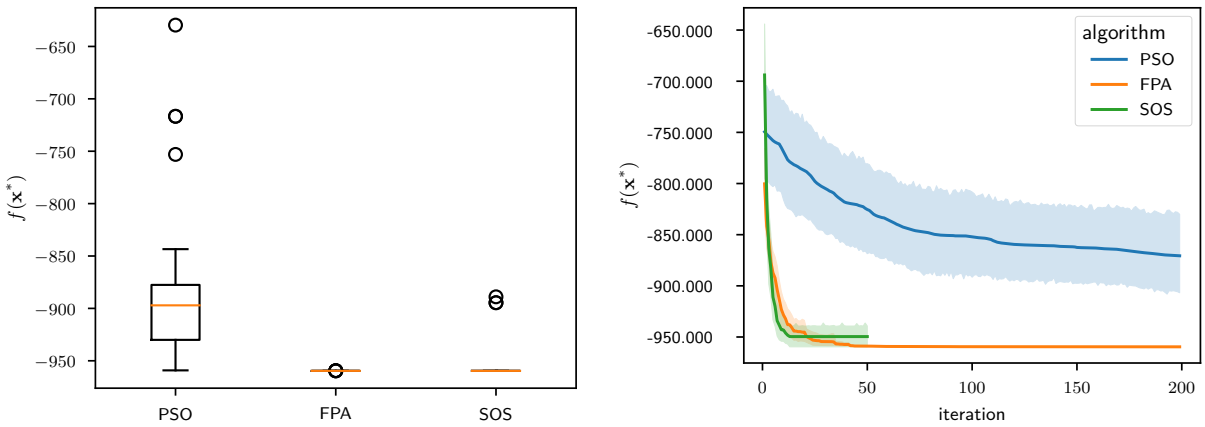


Figura 3: Função objetivo `eggholder`.

(a) Dispersão. (b) Convergência.



$$\mathbf{D} = \left|\mathbf{C} \cdot \mathbf{x}_p(t) - \mathbf{x}(t)\right| \tag{1}$$

$$\mathbf{x}(t+1) = \mathbf{x}_p(t) - \mathbf{A} \cdot \mathbf{D} \tag{2}$$

, a cada nova iteração $t+1$, o vetor de posição $\mathbf{x}$ é atualizado, levando em consideração o candidato à ótimo (posição da presa, $\mathbf{x}_p$) e a distância euclidiana $\mathbf{D}$. Os vectores $\mathbf{C}$ e A, são atualizados conforme

$$\mathbf{A} = 2\mathbf{a} \cdot \mathbf{r}_1 - \mathbf{a} \tag{3}$$

$$\mathbf{C} = 2 \cdot \mathbf{r}_2 \tag{4}$$

, onde $\mathbf{r}$ é um vetor com componentes randômicas entre 0 e 1 e o vetor $\mathbf{a}$ tem valores decrescentes de 2 até 0. Desse modo, cada lobo pode atualizar sua posição randomicamente ao redor de uma presa, porém essa nova posição o leva cada vez mais próximo da presa a cada iteração.

Na etapa de *caça*, fica evidente a organização hierárquica da matilha, os lobos dominantes têm papel de guiar o grupo. Para mimetizar esse comportamento, o vetor posição dos melhores candidatos

Figura 4: Função objetivo `griewank`.

(a) Dispersão. (b) Convergência.

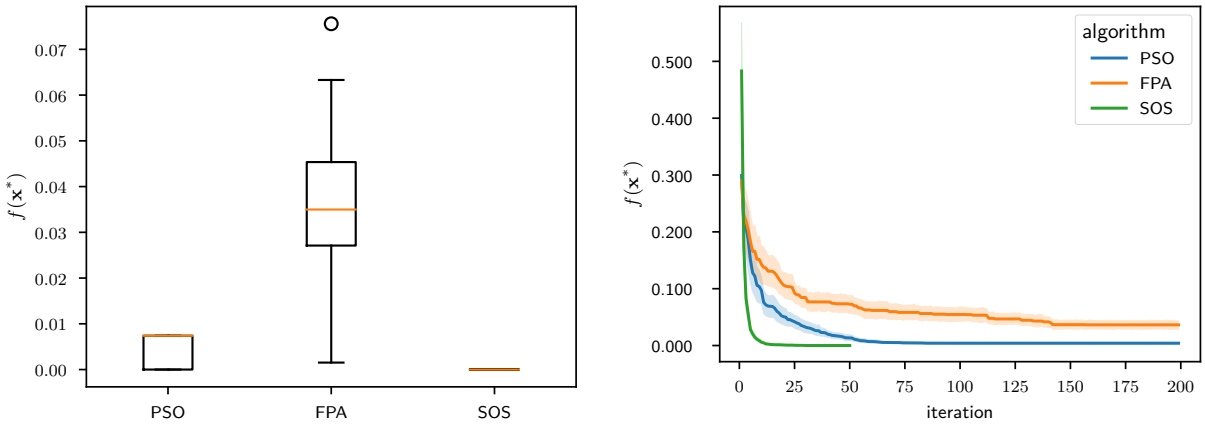
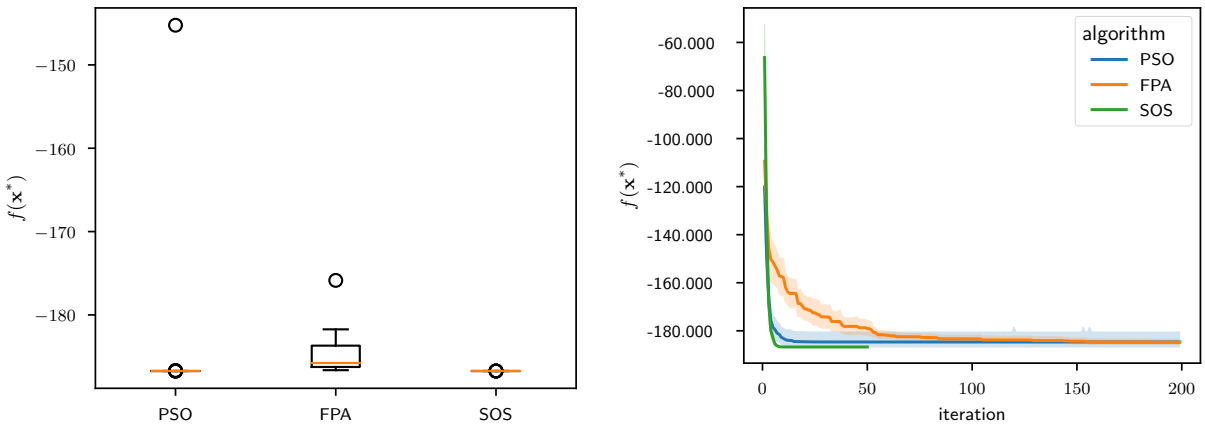


Figura 5: Função objetivo `shubert`.

(a) Dispersão. (b) Convergência.



à ótimo, ou seja, com melhor *fitness*, são ranqueados como α, β e δ, esses vetores são salvos, enquanto que os outros membros da matilha são forçados à atualizar suas posições, conforme as equações 5 levando em consideração a posição dos membros dominantes.

$$\mathbf{D}_\alpha = |\mathbf{C}_1 \cdot \mathbf{x}_\alpha - \mathbf{x}|, \qquad \mathbf{D}_\beta = \left|\mathbf{C}_2 \cdot \mathbf{x}_\beta - \mathbf{x}\right|, \qquad \mathbf{D}_\delta = |\mathbf{C}_3 \cdot \mathbf{x}_\delta - \mathbf{x}| \tag{5}$$

$$\mathbf{x}_1 = \mathbf{x}_\alpha - \mathbf{A}_1 \cdot (\mathbf{D}_\alpha), \qquad \mathbf{x}_2 = \mathbf{x}_\beta - \mathbf{A}_2 \cdot \left(\mathbf{D}_\beta\right), \qquad \mathbf{x}_3 = \mathbf{x}_\delta - \mathbf{A}_3 \cdot (\mathbf{D}_\delta) \tag{6}$$

$$\mathbf{x}(t+1) = \frac{\mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3}{3} \tag{7}$$

A estratégia de *atacar* é emulada pelo valor decrescente de $\mathbf{a}$, o que leva os lobos para posições cada vez mais próximas da presa $\mathbf{x}_p$. A estratégia de busca pela presa, é representada pelos valores randômicos contidos nos vetores $\mathbf{A}$ e $\mathbf{C}$, o que leva os lobos à procurarem por novas posições, evitando que o algoritmo fique preso em mínimos locais e aumentando a capacidade de exploração do espaço amostral.

Figura 6: Função objetivo `sixhump`.

(a) Dispersão. (b) Convergência.

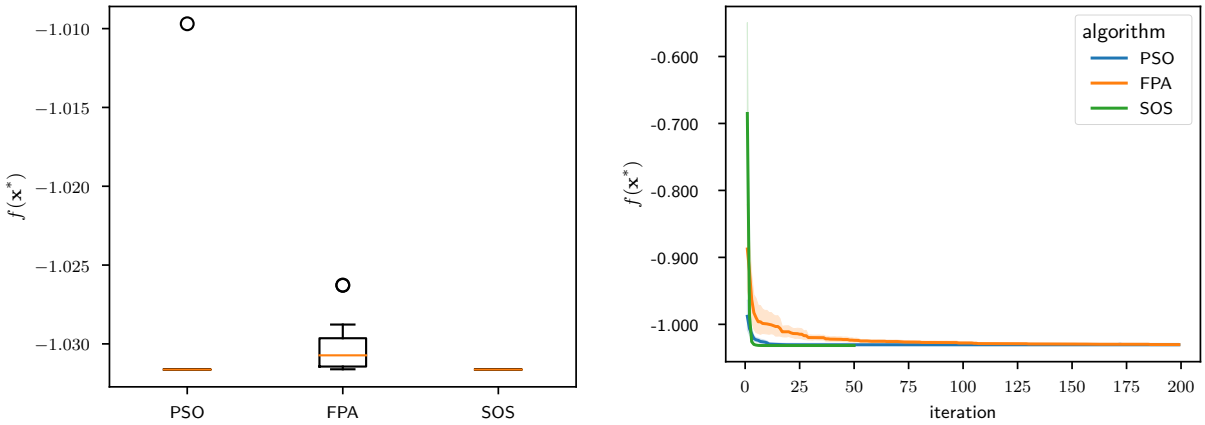
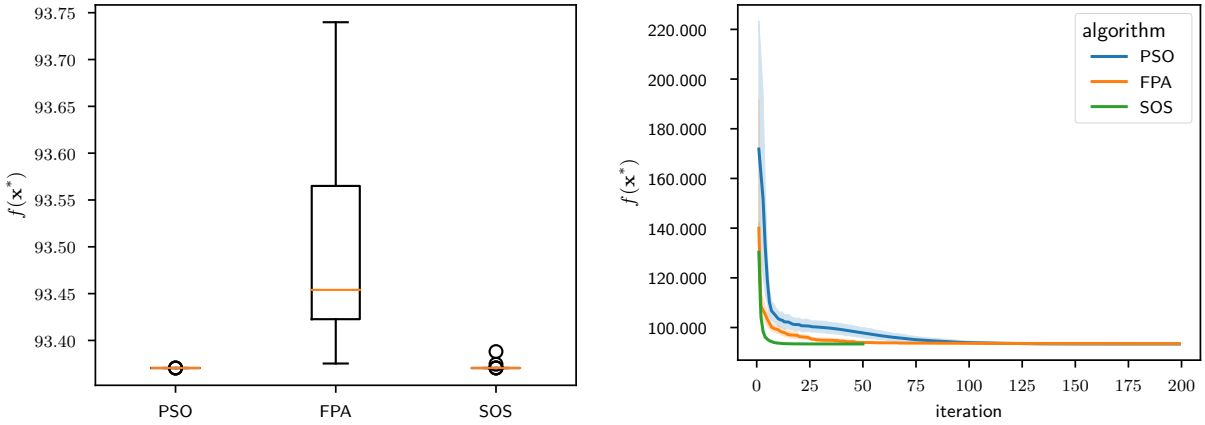


Figura 7: Função objetivo `regularized_ts`.

(a) Dispersão (b) Convergência



3.1 Análise Estatística (GWO)

A análise estatística do algoritmo *GWO* utilizou uma população com o mesmo número de indivíduos (80) e o mesmo critério de parada (200 avaliações) descritos na seção 1. O algoritmo em questão não possuiu outros hiper parâmetros, os valores de **a** e **r** seguiram a referência Mirjalili et al. (2014).

Os resultados em termos de dispersão e convergência são apresentados nas Figuras 8 à 13, os resultados para as demais meta heurísticas foram repetidos para melhor comparação.

A Tabela **??** apresenta os valores médios e os desvios obtidos por cada uma das meta heurísticas no conjunto de funções objetivo proposto, com destaque em verde para os melhores resultados e em vermelho os piores para cada uma das funções.

3.2 Função `eason`

Os algoritmos PSO e *SOS* apresentaram os melhores resultados em termos de acurácia e dispersão, conforme visto no gráfico 2 (a). Todos os algoritmos aproximaram bem o valor ótimo, sendo que FPA apresentou a convergência mais lenta, Figura 8 (b).

3.3 Função eggholder

Os algoritmos *PSO* e *GWO* não foram capazes de obter uma boa aproximação para o ótimo global, mostrando uma tendência à convergirem para um ótimo local, apresentando também uma maior dispersão, como pode ser visto na Figura 9 (a). A Figura 9 (b), mostra a convergência dos algoritmos, o algoritmo SOS apresentou a melhor velocidade de convergência, porém o FPA foi capaz de obter um valor ótimo mais próximo do ótimo global de referência.

3.3.1 Função griewank

Nesse problema, os algoritmos *PSO*, *SOS* e *GWO* obtiveram baixa dispersão, como visto na Figura 10 (a) e rápida taxa de convergência, 10 (b). Apenas o algortimo *FPA* aparenta convergir para um valor de ótimo local, e também uma maior dispersão nos resultados obtidos nas 20 rodadas.

3.4 Funções shubert e sixhump

De maneira análoga ao caso anterior os algoritmos *PSO*, *SOS* e *GWO* foram superiores, tanto em termos de uma menor dispersão, Figuras **??** (a), quanto em termos da rápida taxa de convergência **??** (b). Diferente da função anterior o algoritmo *FPA* também apresentou bom desempenho, convergindo para o valor de ótimo glocal de referência.

3.5 Função regularized_ts

Os algoritmos *PSO*, *SOS* e *GWO* convergiram para valores ótimos muito próximos, com pouca dispersão, como visto na figura 13 (a). A taxa de convergência, apresentada na Figura 13 (b), para os algoritmos mencionados, também foi bastante semelhante. O desempenho do algoritmo *GWO* foi inferior aos temais, tanto em dispersão, quanto na taxa de convergência. Além disso o algoritmo *GWO* convergiu para um valor ótimo diferente dos demais, indicando que ele não foi capaz de encontrar corretamente o ótimo global, pelo menos dentro do limite de 200 avaliações da função objetivo.

Figura 8: Função objetivo easom.

(a) Dispersão. (b) Convergência.

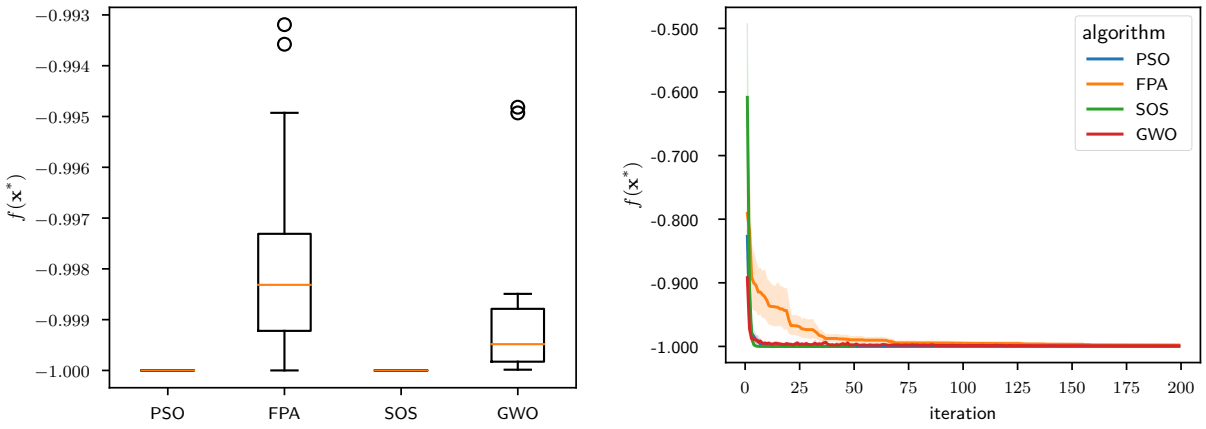


Figura 9: Função objetivo `eggholder`.

(a) Dispersão. (b) Convergência.

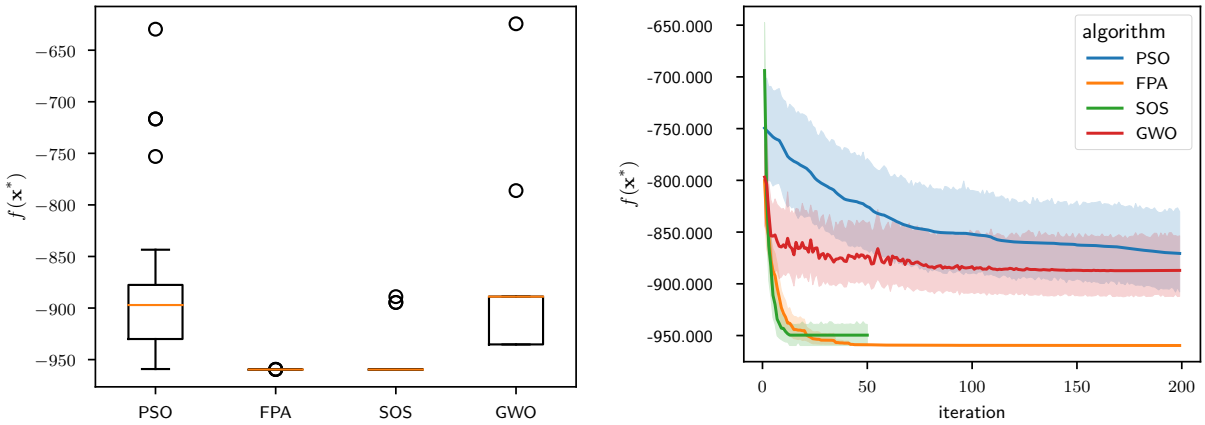
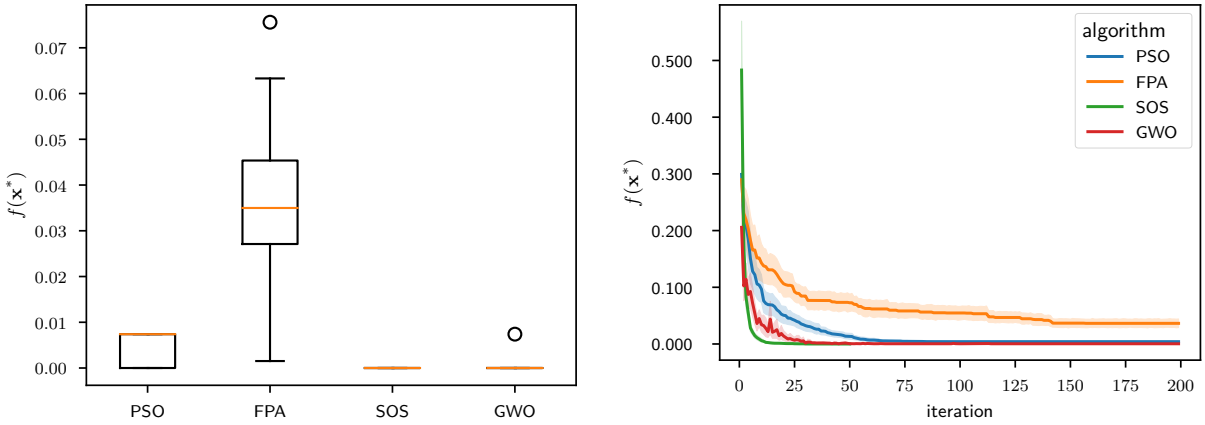


Figura 10: Função objetivo `griewank`.

(a) Dispersão. (b) Convergência.



4 Considerações

As meta heurísticas bioinspiradas apresentaram soluções efetivas para o problema de otimização, representado aqui por um conjunto de seis funções objetivo. Todos os métodos foram capazes de minimizar as funções objetivo propostas, ainda que para alguns casos, o valor ótimo encontrado não corresponda ao ótimo global.

O algoritmo *SOS* teve desempenho superior aos demais, obtendo os menores valores médios para quatro dos seis problemas de minimização propostos. Sua taxa de convergência também foi superior, nesse ponto, vale ressaltar que o custo de cada iteração desse método requer quatro vezes mais avaliações da função objetivo.

Quanto ao valor ótimo obtido, o algoritmo *PSO* foi superior aos demais para apens um entre os seis problemas propostos, e foi o pior também para apenas um desses problemas. O algoritmo *FPA* obteve os piores resultados para três dos problemas propostos, e o melhor resultado para apenas um dos problemas. E por fim, o algoritmo *GWO* foi inferior à todos os demais para um dos problemas propostos e não foi melhor que nenhum deles nos outros testes. Nota-se que a performance de cada

Figura 11: Função objetivo `shubert`.

(a) Dispersão. (b) Convergência.

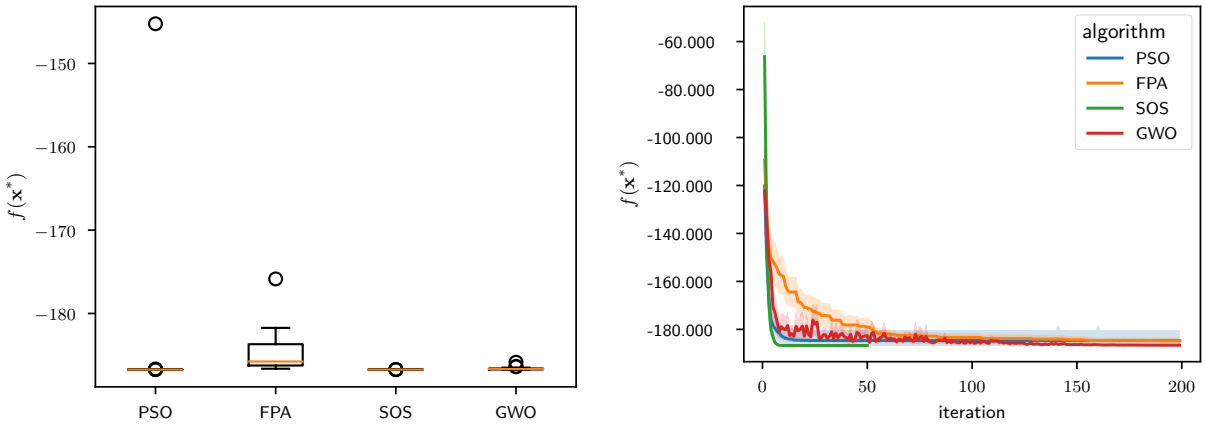
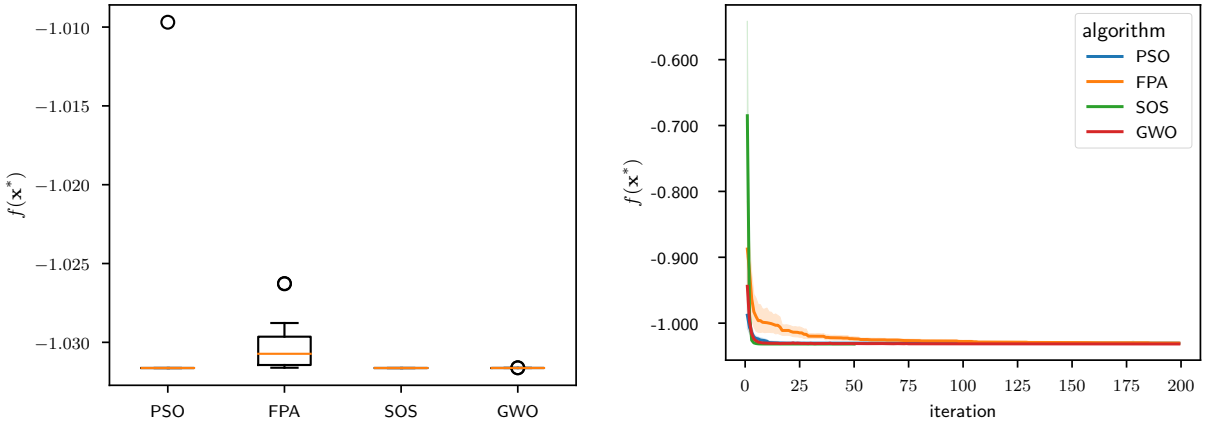


Figura 12: Função objetivo `sixhump`.

(a) Dispersão. (b) Convergência.

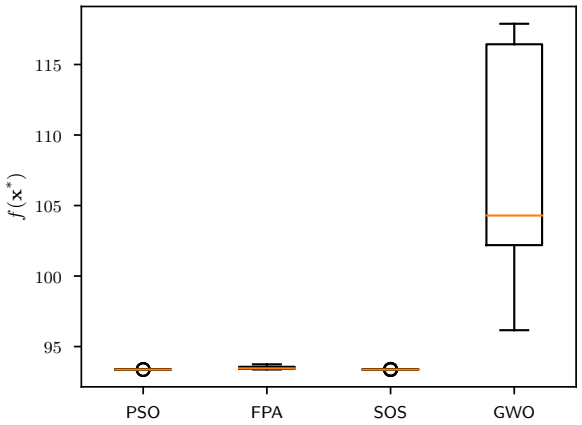


meta heurística depende fortemente da função objetivo a ser otimizada, sendo portanto improvável obter uma meta heurística superior em todos os cenários.

Conclui-se que dentre os algoritmos testados o *GWO* obteve os resultados mais robustos, além disso essa meta heurística não possui hiper parâmetros para serem ajustado, e que poderiam eventualmente afetar a sua performance. Por esse mesmo motivo, ressalta-se que essa conclusão não garante a superioridade desse algoritmo em quaisquer outros cenários, uma vez que os demais algoritmos *PSO* e *FPA* podem ter sua performance alterada por meio de otimização dos seus hiper parâmetros.

Figura 13: Função objetivo `regularized_ts`.

(a) Dispersão



(b) Convergência

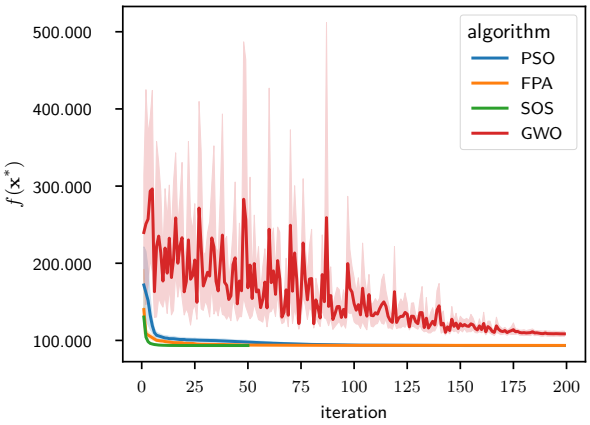


Tabela 2: Resultado em termos de média e desvio padrão.

		shubert	griewank	sixhump	easom	eggholder	regularized_ts
PSO	ave	-184.6530	0.0041	-1.0305	-1.0000	-870.6296	93.3705
	std	9.0423	0.0037	0.0048	0.0000	88.9875	0.0000
FPA	ave	-184.7586	0.0364	-1.0302	-0.9979	-959.6081	93.5028
	std	2.4345	0.0174	0.0015	0.0019	0.0673	0.1169
SOS	ave	-186.7309	0.0000	-1.0316	-1.0000	-949.5999	93.3716
	std	0.0000	0.0000	0.0000	0.0000	23.9239	0.0039
GWO	ave	-186.6014	0.0004	-1.0316	-0.9990	-887.0351	108.4933
	std	0.2039	0.0016	0.0000	0.0014	69.1138	7.4198
	ref	-186.7309	0.0000	-1.0316	-1.0000	-959.6400	93.3705

A Implementação

Os algoritmos foram implementados em `python` e estão disponíveis no repositório publico do `github` https://github.com/properallan/ENE300/. Nesse link encontram-se instruções para instalação do pacote A implementação do algoritmo *GWO* está no arquivo `ene300/optimization/_gwo.py`. O arquivo principal utilizado para as análises desse trabalho está no diretório `resources/tarefas/trabalho_2/statistics.ipynb`.

A.0.1 Implementação do algoritmo *GWO*

```
import numpy as np
import time
from ene300.functions import function_counter

# Grey Wolf Optimizater (GWO)
class GWO:
    def __init__(self):
        pass

```

```
10      def __call__(self, objective_function, position_boundary, population,
        ↪  a_function, itmax, max_fa, direction='minimize'):
11          ini_time = time.process_time()
12
13          objective_function_ = objective_function
14          @function_counter
15          def objective_function(x):
16              if direction == 'minimize':
17                  return objective_function_(x)
18              elif direction == 'maximize':
19                  return - objective_function_(x)
20
21          position_boundary = np.array(position_boundary)
22          dimensions = len(position_boundary)
23
24          position = np.zeros((dimensions, population))
25          fit = np.zeros(population)
26
27          min_position = position_boundary[:,0]
28          max_position = position_boundary[:,1]
29
30          # random initialization of population
31          for i in range(population):
32              position[:,i] = min_position +
                ↪  np.random.rand(dimensions)*(max_position-min_position)
33
34          fit = objective_function(position)
35
36          history = {}
37
38          history['iteration'] = []
39          history['position'] = []
40          history['global_best'] = []
41          history['best_fit'] = []
42          history['cpu_time'] = []
43          history['position_boundary'] = position_boundary
44          history['objective_function'] = objective_function_
45          history['population'] = population
46          history['itmax'] = itmax
47          history['max_fa'] = max_fa
48          history['directon'] = direction
49
50
51          sorted_fit = np.sort(fit)
52          sorted_i = np.argsort(fit)
53
54          best_fit = np.copy(sorted_fit[0])
55
56          # calculate fitness of each agent
```

```
57          alpha = np.copy(position[:, sorted_i[0]])
58          beta  = np.copy(position[:, sorted_i[1]])
59          delta = np.copy(position[:, sorted_i[2]])
60
61          self.itmax = itmax
62          it = 0
63          break_flag = False
64          while it < itmax and break_flag is False:
65              it += 1
66              self.it = it
67              a = self._get_function(a_function)
68
69              for i in range(3,population):
70                  #  update position of each search agent
71                  #  update a, A and C
72                  position[:,i] = self._update_agent(position[:,i], alpha, beta,
                    ↪  delta, a)
73
74              # position constraint
75              for j in range(dimensions):
76                  position[j,:] = np.clip(position[j,:], *position_boundary[j])
77
78              # calculate fitness
79              fit = objective_function(position)
80
81              sorted_fit = np.sort(fit)
82              sorted_i = np.argsort(fit)
83
84              best_fit = np.copy(sorted_fit[0])
85
86              # update alpha, beta and delta
87              alpha = np.copy(position[:, sorted_i[0]])
88              beta  = np.copy(position[:, sorted_i[1]])
89              delta = np.copy(position[:, sorted_i[2]])
90
91              # store best and iterate
92              global_best = np.copy(alpha)
93
94              history['iteration'].append(it)
95              history['position'].append(np.copy(position))
96              history['global_best'].append(np.copy(global_best))
97              history['best_fit'].append(float(best_fit))
98
99              if objective_function.calls >= max_fa:
100                 break_flag == True
101                 break
102
103         cpu_time = time.process_time() - ini_time
104         history['cpu_time'] = cpu_time
```

```
105         history['function_evaluations'] = objective_function.calls
106 
107         return global_best, best_fit, history
108 
109     def _get_function(self, function_dict):
110         if function_dict['function'] == 'constant':
111             weight = function_dict['constant']
112 
113         elif function_dict['function']== 'random':
114             weight = 0.5 + np.random.rand()/2
115 
116         elif function_dict['function'] == 'linear_decrease':
117             t = self.it-1
118             tmax = self.itmax
119             weight = function_dict['max'] - (function_dict['max']
                ↪  -function_dict['min']) * t / tmax
120 
121         elif function_dict['function'] == 'sigmoidal_increase':
122             gen = self.itmax
123             t = self.it-1
124             tmax = self.itmax
125 
126             n = function_dict['n']
127             u_sign = function_dict['u_sign']
128 
129             u = 10**(np.log(gen)-2)
130             weight = (function_dict['start'] -
                ↪  function_dict['end'])/(1+np.exp(u_sign*(t-n*tmax))) +
                ↪  function_dict['end']
131         else:
132             raise NotImplementedError(f"Function {function_dict['function']} not
                ↪  implemented in pso.get_function(self, function_dict)")
133 
134         return weight
135 
136     def _update_agent(self, position, alpha, beta, delta, a):
137         C1 = 2*np.random.rand()
138         C2 = 2*np.random.rand()
139         C3 = 2*np.random.rand()
140 
141         A1 = (2*a*np.random.rand() - a)
142         A2 = (2*a*np.random.rand() - a)
143         A3 = (2*a*np.random.rand() - a)
144 
145         D_alpha = np.abs(C1*alpha -position)
146         D_beta  = np.abs(C2*beta  -position)
147         D_delta = np.abs(C3*delta -position)
148 
149         X1 = alpha - A1*(D_alpha)
```

```
150         X2 = beta  - A2*(D_beta)
151         X3 = delta - A3*(D_delta)
152 
153         X = (X1 + X2 + X3)/3.0
154 
155         return X
```

Referências

S. Mirjalili, S. M. Mirjalili, and A. Lewis. Grey wolf optimizer. *Advances in Engineering Software*, 69:46–61, Mar. 2014. doi: 10.1016/j.advengsoft.2013.12.007. URL `https://doi.org/10.1016/j.advengsoft.2013.12.007`.

S. Surjanovic and D. Bingham. Virtual library of simulation experiments: Test functions and datasets. Retrieved March 14, 2023, from `http://www.sfu.ca/~ssurjano`.

X.-S. Yang. Test problems in optimization. 2010. doi: 10.48550/ARXIV.1008.0549. URL `https://arxiv.org/abs/1008.0549`.