Universidade Federal de Lavras

Max Deivid do Nascimento

Implementação de Algoritmo Particle Swarm Optimization para os problemas de Six-Hump Camel Function e Shubert Function

Trabalho da diciplina de Computação evolucionária que visa resolver os problemas de Six-Hump Camel e Shubert usando o algoritmos PSO.

Professor Bruno Henrique Groener Barbosa

Lavras - MG
Maio de 2025

1 Motivação

Visando resolver os problemas selecionados de Shubert [1] e Six-Hump Camel [2], na diciplina de Computação evolucionária, do programa de mestrado em Engenharia de Sistemas e Automação, foi proposta a utilização da abordagem de algoritmos *Particle Swarm Optimization* (PSO). O objetivo era desenvolver um algoritmo PSO de minimização para ambas as funções e obter um valor o mais próximo possível dos valores mencionados nas propostas.

2 Descrição dos Problemas de Otimização

Nesta seção, descrevemos brevemente dois problemas clássicos de otimização utilizados com frequência na validação de algoritmos: a função de Shubert e a função Six-Hump Camel.

2.1 Função de Shubert

A função de Shubert é um benchmark conhecido por apresentar múltiplos mínimos locais e globais, o que a torna particularmente desafiadora para algoritmos de otimização. A versão bidimensional da função é definida como o produto de duas somas, cada uma contendo termos cossenoidais dependentes da variável correspondente. A função é expressa por:

$$f(x_1, x_2) = \left(\sum_{i=1}^{5} i \cos\left[(i+1)x_1 + i\right] \right) \left(\sum_{i=1}^{5} i \cos\left[(i+1)x_2 + i\right] \right) \quad (1)$$

com $x_1, x_2 \in [-10, 10]$. A função possui múltiplos mínimos globais com o valor mínimo aproximado de -186.7309 [1].

Ainda é póssível restringir $x_1, x_2 \in [-5.12, 5.12]$, mas para o trabalho em questão foi utilizado $x_1, x_2 \in [-10, 10]$ conforme evidenciado pela superfície da figura 1.

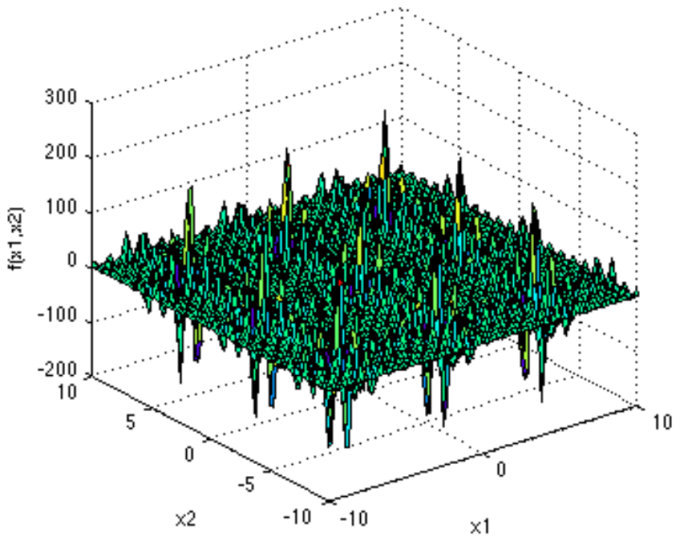

Figura 1: Visualização da função de Shubert

2.2 Função Six-Hump Camel

A função Six-Hump Camel é outra função de teste bastante utilizada em problemas de otimização global, especialmente em duas dimensões. Sua superfície apresenta vários mínimos locais e dois mínimos globais equivalentes. A função é dada por:

$$f(x_1, x_2) = \left(4 - 2.1x_1^2 + \frac{x_1^4}{3}\right) x_1^2 + x_1 x_2 + \left(-4 + 4x_2^2\right) x_2^2 \qquad (2)$$

com $x_1 \in [-3, 3]$ e $x_2 \in [-2, 2]$. Os mínimos globais conhecidos ocorrem próximos a $(x_1, x_2) \approx (\pm 0.0898, \mp 0.7126)$, com um valor mínimo de aproximadamente -1.0316 [2].

Na figura 2 é possível visualizar a superfície denotada pela função (2).

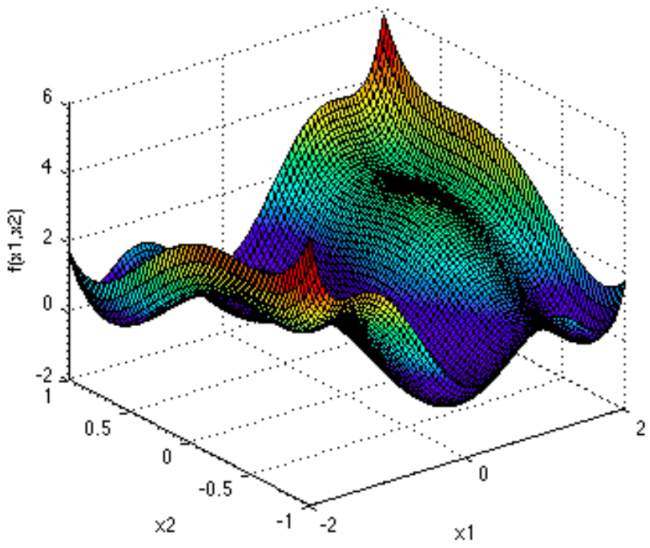

Figura 2: Visualização da função de Six-Hump Camel Function

3 Especificações do Trabalho

Conforme as diretrizes do trabalho, foi proposto que o algoritmo genético desenvolvido apresentasse as seguintes características principais:

- Utilização de representação real;
- Emprego de fator de inércia;
- Emprego de fator de constrição.

Além desses requisitos, também foram estabelecidos critérios para a análise experimental:

- Realizar 30 execuções independentes do algoritmo para cada configuração testada;
- Apresentar os resultados estatísticos (média, mediana, valor máximo e valor mínimo) obtidos nas 30 execuções;
- Ilustrar os resultados por meio de gráficos do tipo boxplot, considerando as seguintes combinações:
  - $w = 0{,}9$, $c_1 = 2$, $c_2 = 2$;
  - $w = [0{,}9, 0{,}4]$, $c_1 = 2$, $c_2 = 2$;
  - $X = 0{,}73$, $c_1 = 2{,}05$, $c_2 = 2{,}05$;

- Apresentar gráficos que demonstrem a evolução dos resultados ao longo das gerações para cada combinação, destacando também o melhor desempenho obtido.

4 Códigos

Para a resolução dos problemas propostos a técnica de algoritmos PSO foi utilizada. Com isso, o primeiro passo foi analisar a documentação da diciplina para implementar o pseudo-código e obter os demais parâmetros envolvidos com o PSO. A figura 3 demostra qual foi o pseudo-código seguido para construir o PSO.

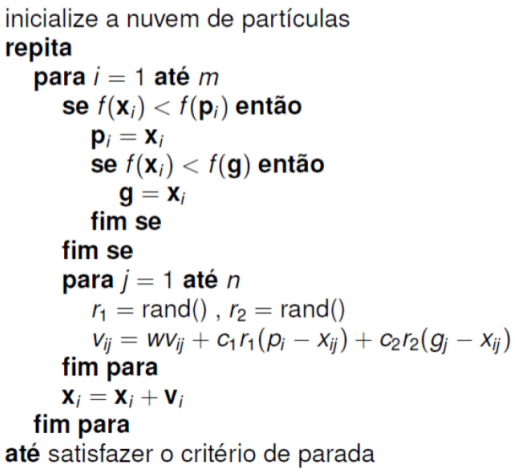

Figura 3: Pseudo-código do ciclo principal do PSO

O código representado pelo pseudo-código foi implementado no arquivo *pso_ algorithm.py*, juntamente com as funções de Shubert e Camel, enquanto os códigos que tratavam das combinações, plots e chamadas do método foram implementados no arquivo *main.py*.

4.1 pso_algorithm

O arquivo *pso_ algorithm.py* foi onde foram implementadas as funções de gerenciamento das partículas do PSO, com os cálculos de velocidade, posição e aquisição das métricas. O código funciona com uma função inicial *run_ pso*, mostrada a seguir.

```
1 def run_pso(generations, m, data_config, w, c1, c2, X):
2     x, v = initialize(m, data_config)
3     p = x
4     g = x[0]
5     best_results = []
```

```
    global_result = []

    # Primeira avaliacao
    scores_x = run_function(data_config, x)
    scores_p = scores_x
    score_g = scores_x[0]

    for gen in range(generations):
        # Ajustando o valor de w, se necessario
        if not X:
            w_now = max(w) - gen*((max(w)-min(w))/generations)

        best_results.append(scores_p)
        # Coleta a fitness media
        global_result.append(score_g)

        for i in range(m):
            scores_x = run_function(data_config, x)
            if scores_x[i] < scores_p[i]:
                p[i] = x[i]
                scores_p[i] = scores_x[i]
                if scores_x[i] < score_g:
                    g = x[i]
                    score_g = scores_x[i]
            for j in range(len(x[i])):
                r1, r2 = get_r(), get_r()
                if not X:
                    v[i][j] = w_now*v[i][j] + c1*r1*(p[i][j] -
                        x[i][j]) + c2*r2*(g[j] - x[i][j])
                else:
                    v[i][j] = X*(v[i][j] + c1*r1*(p[i][j] -
                        x[i][j]) + c2*r2*(g[j] - x[i][j]))
                if v[i][j] <
                    data_config['v_max'][f'x{j+1}']['min']:
                    v[i][j] =
                        data_config['v_max'][f'x{j+1}']['min']
                if v[i][j] >
                    data_config['v_max'][f'x{j+1}']['max']:
                    v[i][j] =
                        data_config['v_max'][f'x{j+1}']['max']
            x[i] = [x[i][k] + v[i][k] for k in
                range(len(x[i]))]
            if x[i][j] <
                data_config['range'][f'x{j+1}']['base']:
                x[i][j] =
                    data_config['range'][f'x{j+1}']['base']
            if x[i][j] >
                data_config['range'][f'x{j+1}']['top']:
                x[i][j] =
                    data_config['range'][f'x{j+1}']['top']
    return g, score_g, best_results, global_result
```

Essa função é a responsável por implementar o pseudo-código da figura 3, iterando sobre o número de vezes k estabelecido, com os parâmetros estipula-

dos na função *main*. Além de executar o código principal, a função também coleta as métricas do PSO através das variaveis *best_results* e *global_result*.

Ao início da função *run_pso* o primeiro passo é fazer a inicialização das partículas, o que é feito pela função *initialize*. Nessa função são passados como parâmetro o número de partículas desejado e a variável data_config, que dita o problema a ser abordado (*Shubert* ou *Camel*), descreve quais os valores mínimo e máximo de x1 e x2 e os valores de velocidade mínima e máxima. O código se vale da função random do *Python* para obter os valores dentro desses intervalos de x e volocidade máxima.

```
def initialize(m, data_config):
    x =
        [[random.uniform(data_config.get('range').get('x1').get('base'),
        data_config.get('range').get('x1').get('top')),
          random.uniform(data_config.get('range').get('x2').get('base'),
              data_config.get('range').get('x2').get('top'))]
              for i in range(m)]

    v =
        [[random.uniform(data_config.get('v_max').get('x1').get('min'),
        data_config.get('v_max').get('x1').get('max')),
          random.uniform(data_config.get('v_max').get('x2').get('min'),
              data_config.get('v_max').get('x2').get('max'))]
              for i in range(m)]
    return x, v
```

Depois de inicializadas as partículas o algoritmo começa sua parte cíclica. De acordo com o número de vezes em que o algoritmo será processado (*generations*), a função *run_pso* irá iterar sobre cada partícula, verificando se a posição de x é melhor que a melhor posição já encontrada para aquela partícula e selecionando os melhores valores como os valores de p. De igual modo, o melhor valor de x dentro da rodada atual é selecionado como g, que é o melhor resultado global.

Para validar e comparar os resultados de cada partícula é necessário avaliar o resultado de suas variáveis dentro da função de cada problema. Isso é feito chamando a função *run_function* no cálculo dos *scores* (pontuações). A função *run_function* recebe *data_config* que indica qual o problema abordado através da chave *"problem"*. A função então direciona o conteúdo das variáveis para a função que calcula o resultado baseado no problema e retorna as pontuações obtidas.

```
def run_function(data_config, variables):
    if data_config.get('problem') == 'shubert':
        return [shubert(x[0], x[1]) for x in variables]
    else:
        return [camel(x[0], x[1]) for x in variables]
```

As funções que implementam os problemas abordados podem ser vistas a seguir.

```
def shubert(x1, x2):
```

```
    sum1 = sum(i * math.cos((i + 1) * x1 + i) for i in
        range(1, 6))
    sum2 = sum(i * math.cos((i + 1) * x2 + i) for i in
        range(1, 6))
    return sum1 * sum2

def camel(x1, x2):
    term1 = (4 - 2.1 * x1**2 + x1**4 / 3) * x1**2
    term2 = x1 * x2
    term3 = (-4 + 4 * x2**2) * x2**2
    return term1 + term2 + term3
```

Depois de calculados os scores e feitas as devidas substituições e seleções dos melhores valores, as velocidades são calculadas tendo base os parâmetros dados (Caso haja X a função se ajusta para usar o fator de constrição, caso contrário é utilizado o fator de inércia. Depois as novas posições baseadas nas novas velocidade são calculadas e o ciclo recomeça até que tenham se passado todas as gerações.

4.2 main

Para poder organizar e categorizar todas as combinações dadas dos parâmetros e selecionar a melhor configuração de parâmetros dentre as possibilidades foi criado o código *main*. Nesse código foram inseridos os valores possíveis de cada parâmetro e as configurações fixas. As combinações solicitadas podem ser visualizadas na tabela 1.

m	g	w	$c1$	$c2$	X
100	30	0.9	2	2	None
100	30	[0.9, 0.4]	2	2	None
100	30	None	2.05	2.05	0.73

Tabela 1: Combinações possíveis

O código usado para processar os dados foi implementado na função *main* abaixo, no arquivo *main.py*. A sua função era chamar a função *run_pso* com a determinada configuração de parâmetros e com isso gerar os gráficos e comparações.

```
import ast
import pandas as pd
import matplotlib.pyplot as plt
import itertools
import statistics

from pso_algorithm import run_pso


def get_result_and_plot(g, m, data_config, w, c1, c2, X,
    target):
```

```
    results, score, scores_p, scores_g = run_pso(g, m,
        data_config, w, c1, c2, X)

    plt.figure(figsize=(6, 4))
    plt.axhline(
        y=target,
        marker="o",
        color="r",
        linestyle="--",
        label=f"Valor esperado ({target})",
    )
    plt.plot(scores_g)
    plt.title(
        f'Boxplot dos Scores por Particula (scores_p) - {data_config.get("problem")}'
    )
    plt.ylabel("Score")
    plt.grid(True)

    return {
        "results": results,
        "score": score,
        "scores_p": scores_p,
        "scores_g": scores_g,
        "median_score_p": [statistics.median(p) for p in
            scores_p],
    }


if __name__ == "__main__":
    g = 30
    m = 100
    data_config = {
        "problem": "shubert",
        "range": {
            "x1": {"base": -10, "top": 10},
            "x2": {"base": -10, "top": 10},
        },
        "v_max": {
            "x1": {"min": -1, "max": 1},
            "x2": {"min": -1, "max": 1},
        },
    }
    all_results = []

    for problem, target in {"shubert": -186.7309, "camel":
        -1.0316}.items():
        data_config["problem"] = problem

        results_01 = get_result_and_plot(g, m, data_config,
            [0.9], 2, 2, None, target)
        results_02 = get_result_and_plot(
            g, m, data_config, [0.9, 0.4], 2, 2, None, target
        )
```

```
60          results_03 = get_result_and_plot(
61              g, m, data_config, None, 2.05, 2.05, 0.73, target
62          )
63
64          for r in [results_01, results_02, results_03]:
65              r["problem"] = problem
66              all_results.append(r)
67
68      # Criar um unico DataFrame com todos os dados
69      df = pd.DataFrame(all_results)
70      df_grouped = df.groupby("problem")
71      for problem, data in df_grouped:
72          plt.figure(figsize=(6, 4))
73          plt.boxplot(data["median_score_p"], patch_artist=True)
74          plt.title("Boxplot␣dos␣Scores␣por␣Particula␣
                (scores_p)")
75          plt.ylabel("Score")
76          plt.xlabel("Configuracoes")
77          plt.grid(True)
78
79      # === Identificar os melhores resultados
80      best_shubert = df[df["problem"] == "shubert"]
81      best_shubert =
            best_shubert.loc[best_shubert["score"].idxmin()]
82
83      best_camel = df[df["problem"] == "camel"]
84      best_camel = best_camel.loc[best_camel["score"].idxmin()]
85
86      # === Leitura dos dados do GA
87      df_ga_shubert =
            pd.read_csv("melhores_fitness_shubert.csv",
            header=None)
88      df_ga_camel = pd.read_csv("melhores_fitness_camel.csv",
            header=None)
89
90      # === Grafico de comparacao SHUBERT
91      plt.figure(figsize=(8, 5))
92      plt.axhline(y=-186.7309, color="r", linestyle="--",
            label="Valor␣esperado␣(target)")
93      plt.plot(best_shubert["scores_g"], 'o-', label="PSO",
            color="blue")
94      plt.plot(ast.literal_eval(df_ga_shubert.iloc[7].values[1]),
            'x-', label="GA", color="green")
95      plt.title("Comparacao␣da␣evolucao␣do␣GA␣e␣PSO␣-␣Shubert")
96      plt.xlabel("Iteracoes")
97      plt.ylabel("Score")
98      plt.legend()
99      plt.grid(True)
100
101     # === Grafico de comparacao CAMEL
102     plt.figure(figsize=(8, 5))
103     plt.axhline(y=-1.0316, color="r", linestyle="--",
            label="Valor␣esperado␣(target)")
104     plt.plot(best_camel["scores_g"], 'o-', label="PSO",
```

```
        color="blue")
105     plt.plot(ast.literal_eval(df_ga_camel.iloc[7].values[1]),
            'x-', label="GA", color="green")
106     plt.title("Comparacao da evolucao do GA e PSO - Camel")
107     plt.xlabel("Iteracoes")ge
108     plt.ylabel("Score")
109     plt.legend()
110     plt.grid(True)
111
112     plt.show()
```

Dentro da função, para cada combinação diferente foi gerado um gráfico de evolução do resultado durante as execuções solicitadas. Além disso, os dados das diferentes combinações foram plotados na forma de gráficos boxplot para cada problema a fim de verificar qual configuração gerou o melhor resultado.

Depois disso os dados das melhores configurações de cada problema foram comparados com as melhores configurações do algorítmo genético. Os dados para cada problema foram plotados no mesmo gráfico para comparar o comportamento dos resultados.

5 Resultados

Nessa seção se discutem os resultados atingidos pelo algoritmo PSO desenvolvido em cada um dos dois problema proposto e ao final é feita a comparação entre os resultados do PSO e os atingidos usando algoritmo genético.

5.1 Shubert

Inicialmente foram feitos testes com 3 diferentes combinações de parâmetros para o problema de *Shubert*, como consta na tabela 1. Com isso cada combinação teve sua evolução plotada em um gráfico como constam as figuras 4, 5 e 6.

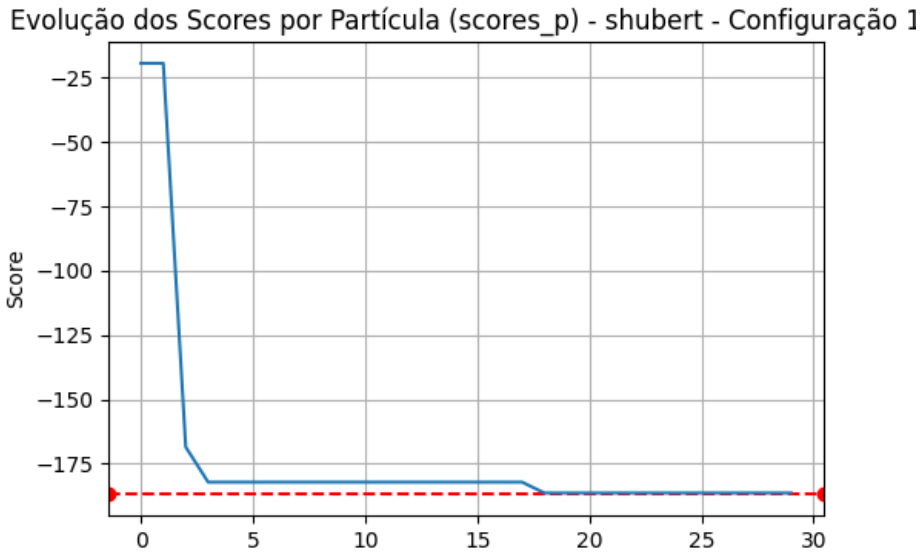

Figura 4: Configuração 01 de Shubert

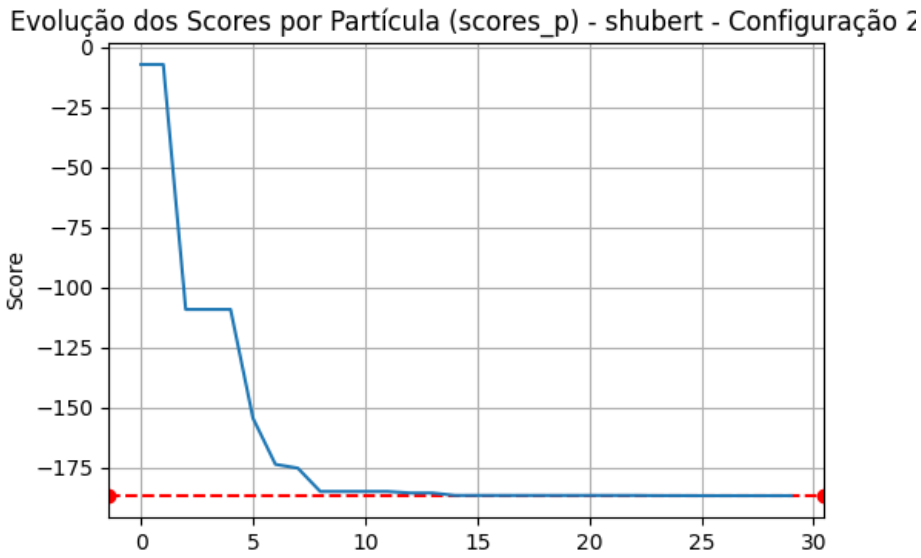

Figura 5: Configuração 02 de Shubert

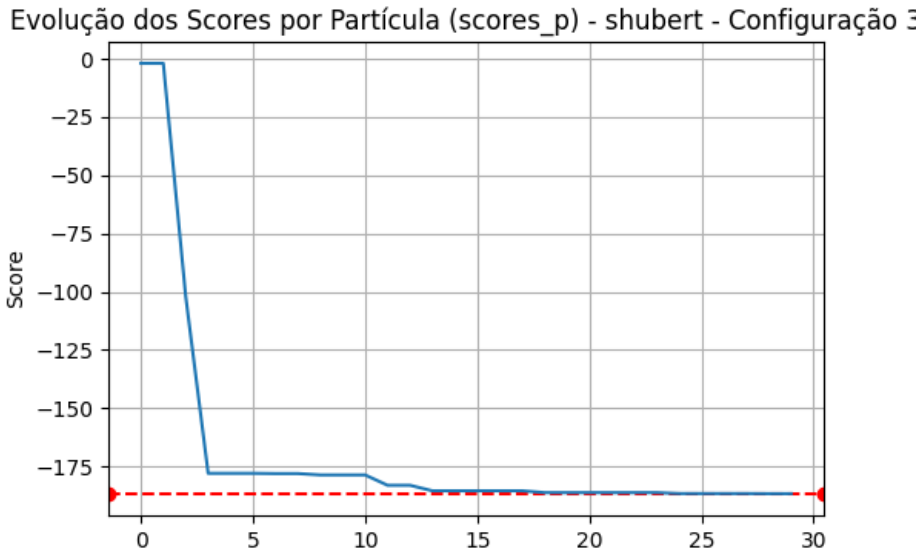

Figura 6: Configuração 03 de Shubert

Em todos os gráficos é possível observar uma linha vermelha tracejada que indica o valor objetivado pela otimização, no caso de *Shubert* seria -186.7309. Dentre as combinações foi possível observar pela comparação entre os gráficos, que dentre as configurações a configuração 03 foi a que se aproximou do valor objetivo mais rapidamente (11° rodada), enquanto a combinação 01 foi a que mais demorou (24° rodada).

Para fins de comparação e avaliação do melhor modelo a métrica utilizada foi a proximidade do valor objetivo. Essa métrica foi dada através da análise do *boxplot*, no qual foram plotados os resultados das três combinações junta conforme mostra a figura 7. Na avaliação, feita em conjunto coma tabela de resultados 3, foi possível perceber que mesmo a configuração 03 tendo sido a que convergiu mais rápido, a configuração que encontrou o menor valor (valor mais próximo do ótimo) foi a combinação 02, por isso, esta será considerada como a melhor.

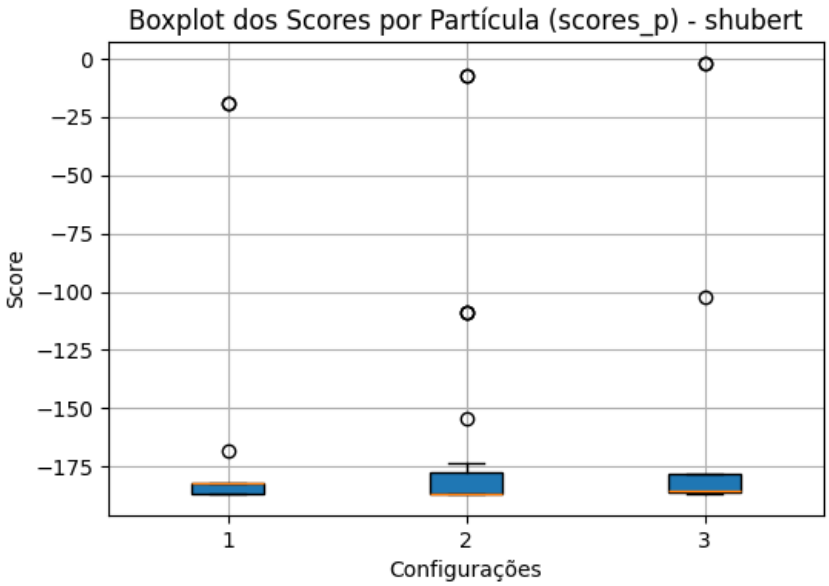

Figura 7: Boxplot de configurações de Shubert

.

Combinação	Mínimo
01	-182,1843
02	-186,6159
03	-185,4357

Tabela 2: Tabela de mínimos de Shubert

Depois de determinada qual a melhor configuração, a evolução do resultado desta foi comparada com a evolução do melhor resultado do algoritmo genético para o mesmo problema. A comparação, vista na figura 8.

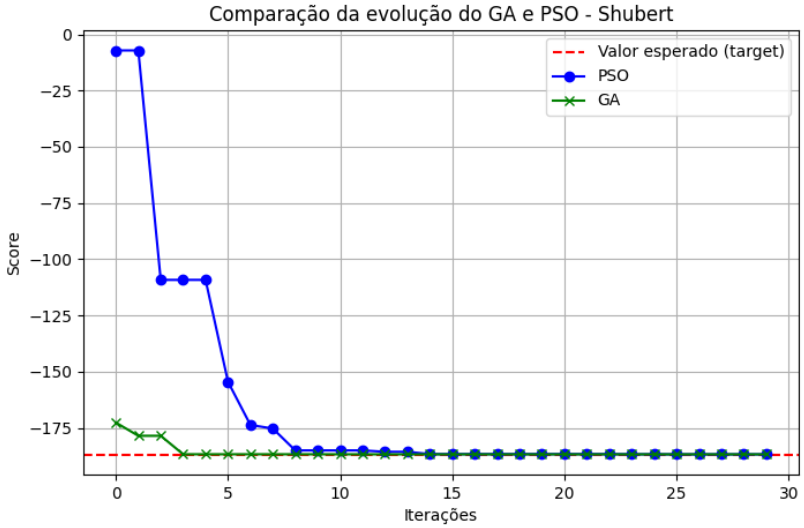

Figura 8: Comparação PSO vs GA Shubert

.

O gráfico indica que o PSO iniciou com um valor muito mais distânte do valor objetivo que o algoritmo genetico, que se aproximou desse valor de mais rapidamente (8° geração), já o PSO se convergiu por volta da 14° rodada. Quanto aos resultados de mínimo, podem ser comparados na tabela **??**, que demonstra que o algoritmo genético se manteve como sendo o melhor em termos de minimização e velocidade de convergência.

Métrica	GA	PSO
Valor de mínimo	-186,7309	-186,6159

Tabela 3: Tabela de mínimos de Shubert

5.2 Six-Hump Camel

Inicialmente foram feitos testes com 3 diferentes combinações de parâmetros para o problema de *Six-Hump Camel*, como consta na tabela 1. Com isso cada combinação teve sua evolução plotada em um gráfico como constam as figuras 9, 10 e 11.

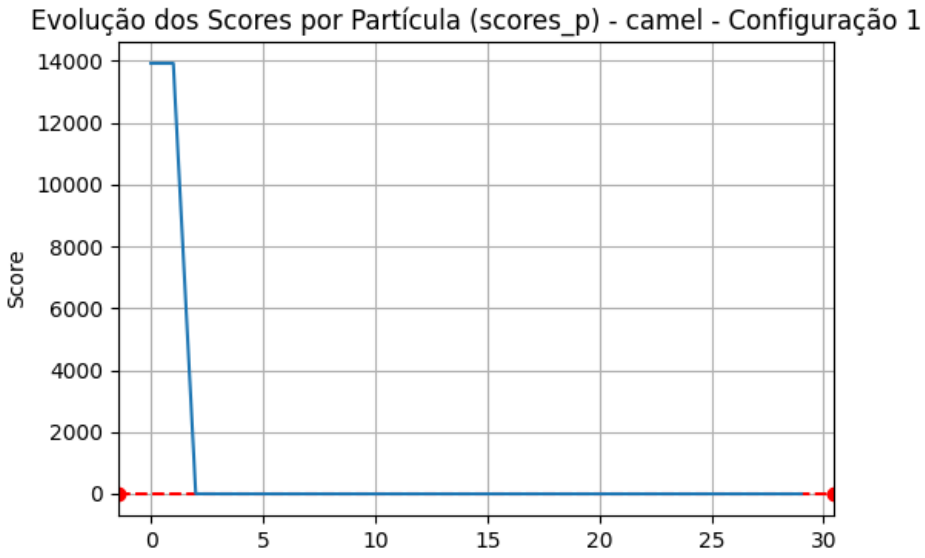

Figura 9: Configuração 01 de Six-Hump Camel

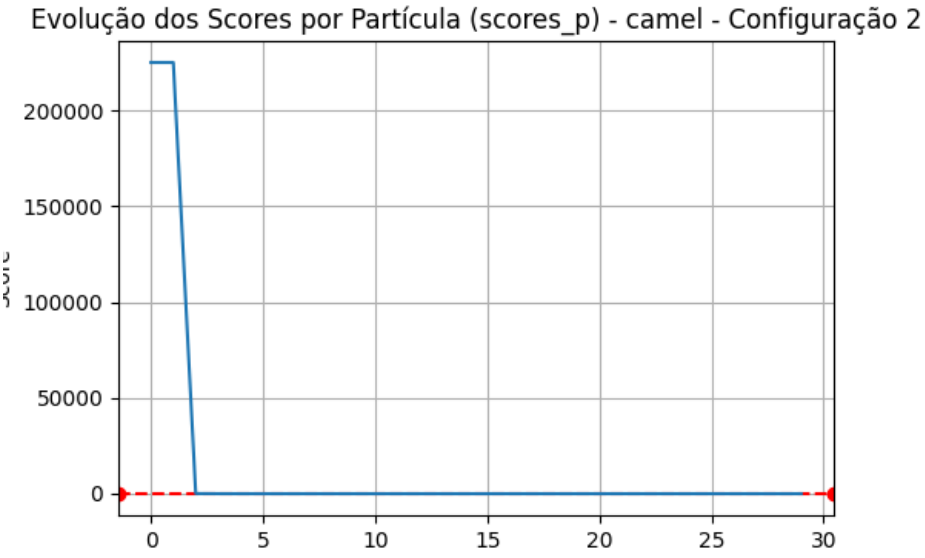

Figura 10: Configuração 02 de Six-Hump Camel

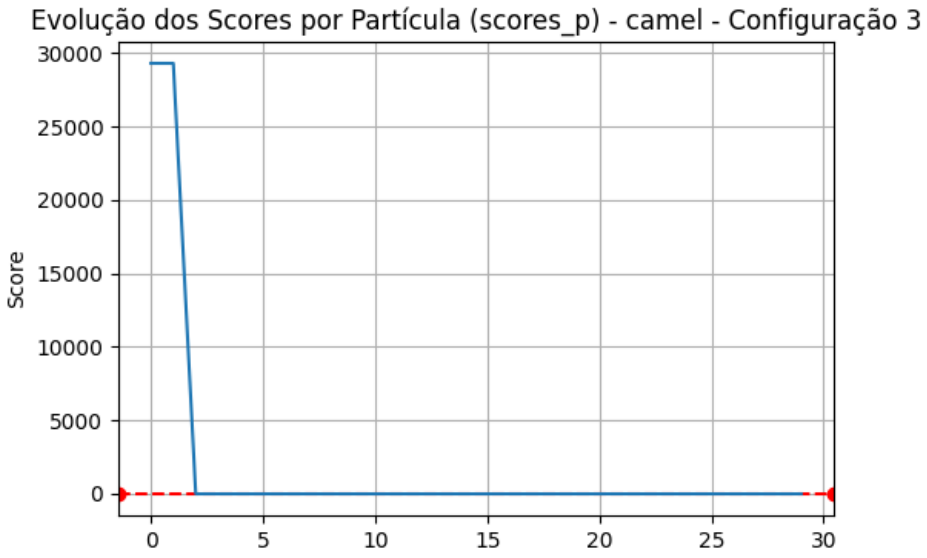

Figura 11: Configuração 03 de Six-Hump Camel

Em todos os gráficos é possível observar uma linha vermelha tracejada que indica o valor objetivado pela otimização, no caso de *Six-Hump Camel* seria -1.0316. Dentre as combinações foi possível observar pela comparação entre os gráficos, que todas as configurações atingiram o valor próximo ao valor esperado na 2° rodada.

Para fins de comparação e avaliação do melhor modelo a métrica utilizada foi a proximidade do valor objetivo. Essa métrica foi dada através da análise do *boxplot*, no qual foram plotados os resultados das três combinações junta conforme mostra a figura 12. Na avaliação, feita em conjunto com a tabela de resultados 5, foi possível perceber que a configuração que encontrou o menor valor (valor mais próximo do ótimo) foi a combinação 01, por isso, esta será considerada como a melhor.

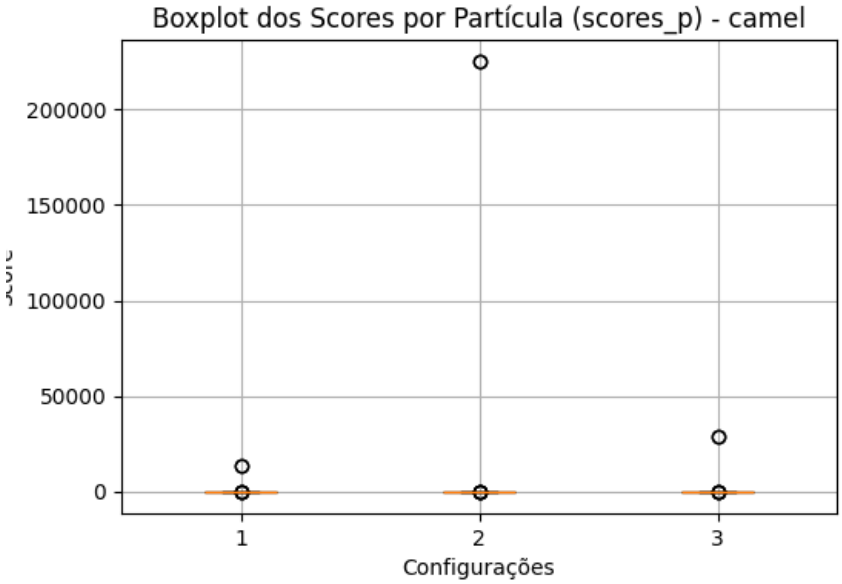

Figura 12: Boxplot de configurações de Six-Hump Camel

.

Combinação	Mínimo
01	-1,0312
02	-1,0288
03	-1,0206

Tabela 4: Tabela de mínimos de Six-Hump Camel

Depois de determinada qual a melhor configuração, a evolução do resultado desta foi comparada com a evolução do melhor resultado do algoritmo genético para o mesmo problema. A comparação, vista na figura 13.

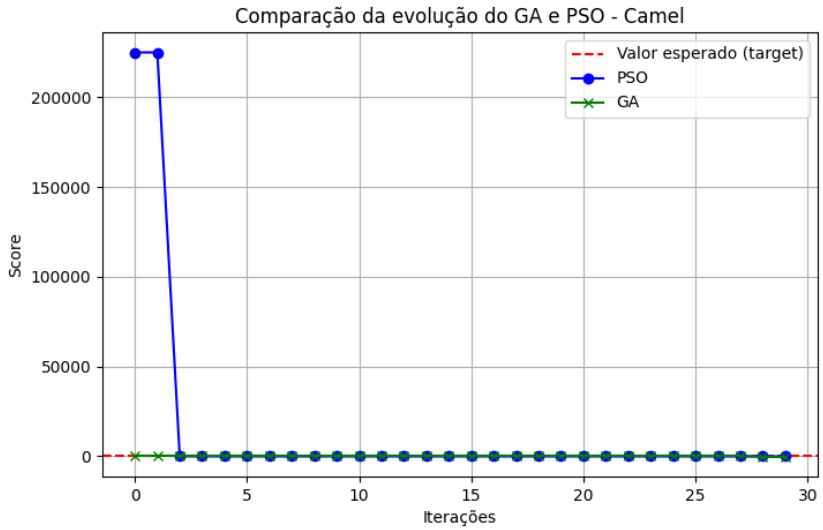

Figura 13: Comparação PSO vs GA Six-Hump Camel

.

O gráfico indica que o PSO iniciou com um valor muito mais distânte do valor objetivo que o algoritmo genético, que se aproximou desse valor de mais rapidamente. Quanto aos resultados de mínimo, podem ser comparados na tabela **??**, que demonstra que o algoritmo genético se manteve como sendo o melhor em termos de minimização e velocidade de convergência.

Métrica	GA	PSO
Valor de mínimo	-1,0316	-1,0312

Tabela 5: Tabela de mínimos de Six-Hump Camel

6 Conclusão

Baseado no código desenvolvido e os resultados alcançados foi possível observar que o algorítmo genético desenvolvido ainda foi melhor na minimização que o algoritmo PSO, porém o algoritmo PSO foi capaz de resolver os dois problemas propostos com precisões de 99,9384511% para Shubert e 99,9655979% para Six-Hump Camel, o que demonstra a eficiência do método. Outro ponto importante é que o algorítmo construído é capaz de abordar outros problemas desde que a função fitness seja implementada e que os devidos parâmetros sejam fornecidos, tal qual a abordagem de algoritmo genético construída.

A Código-fonte

O código-fonte desenvolvido para este trabalho encontra-se disponível no seguinte repositório público do GitHub:

- **Repositório:** `https://github.com/Maxnasc/projeto_pso`

O repositório contém a implementação completa do algoritmo PSO, juntamente com scripts para execução dos experimentos e geração dos gráficos de análise, além de conter os gráficos mostrados no relatório.

Referências

[1] S. Surjanovic and D. Bingham. *Virtual Library of Simulation Experiments: Test Functions and Datasets*. Disponível em: `https://www.sfu.ca/~ssurjano/shubert.html`. Acesso em: abril de 2025.

[2] S. Surjanovic and D. Bingham. *Virtual Library of Simulation Experiments: Test Functions and Datasets*. Disponível em: `https://www.sfu.ca/~ssurjano/camel6.html`. Acesso em: abril de 2025.

[3] NASCIMENTO, Max. *projeto_pso. GitHub*. Disponível em: `https://github.com/Maxnasc/projeto_pso`. Acesso em: 03 mai. 2025.