智能控制作业

WA2214014 杨跃浙 WA2214030 蔡文杰

November 2024

1 作业要求

进化算法（Evolutionary Algorithm, EA）是一类以自然进化和生物遗传学为启发的优化算法，其核心思想来源于物种进化的基本规律，包括自然选择、交叉（重组）、变异和遗传等过程。算法通过模拟这些机制，逐步优化解空间中的候选解，以达到接近或找到全局最优解的目的。

算法首先随机初始化一个种群，每个个体代表一个可能的解，通过目标函数对个体的适应度进行评估。种群中适应度较高的个体更有机会被选中进行遗传操作，模仿自然界“优胜劣汰”的过程。随后，交叉操作通过交换两个个体的部分基因生成新个体，模仿生物繁殖中的基因重组；变异操作则以一定概率随机改变个体基因，模仿突变现象，从而保持种群多样性，避免算法陷入局部最优。最终，根据适应度对种群进行替换，淘汰适应度较低的个体，不断更新种群，直至达到停止条件，如迭代次数上限或解的收敛。

为评估进化算法的优化性能，可选取经典优化函数（如 Sphere 函数、Rastrigin 函数或 Rosenbrock 函数）进行实验。不同函数具有各自特性：单峰函数易于测试算法的收敛精度，而多峰函数可用于验证算法的全局搜索能力。在实验中，通过调整关键参数如种群规模、迭代次数、交叉概率和变异概率，探索它们对算法性能的影响。种群规模直接影响算法的搜索范围和效率，较大的种群规模有助于全局搜索，但可能降低收敛速度；交叉概率和变异概率分别决定了种群基因的多样性和新解的生成速率，需在探索与利用之间取得平衡。

实验中需要记录优化过程中种群最优个体的适应度变化情况，并绘制收敛曲线，以分析算法在不同参数配置下的表现。收敛曲线反映了算法的收敛速度和最终精度，可以直观展示参数对优化结果的影响。

2 实验原理

2.1 函数选择原则

在遗传算法的性能评估中，选择合适的测试基准函数是实验设计的关键。为了能够全面的考察算法在不同优化场景中的表现，函数的选择需要遵循几点原则。首先，为了涵盖优化问题中的多样性与复杂性，选取了具有不同特性的函数。例如，多模态函数（如 Ackley、Rastrigin 和 Schwefel）能够模拟复杂搜索空间，而单模态函数（如 Sphere 和 Rosenbrock）用于检验算法在简单搜索空间中的收敛能力。此外，还包括非凸函数（如 Rosenbrock）和带有相互作用项的函数（如 Matyas），以考察算法在应对复杂函数时的表现。

其次，这些函数包含丰富的全局与局部极值特性。以 Rastrigin 和 Ackley 为例，它们包含大量局部极小值，而全局最优解较难找到，有助于测试算法跳出局部极值的能力。同时，高维扩展性也是函数选择的重要标准，像 Sphere 和 Zakharov 函数可以扩展到任意维度，用于评估算法在高维优化问题上的效率和稳定性。

此外，部分测试函数（如 Griewank 和 Schwefel）包含复杂的计算项，用于模拟真实问题的计算挑战，从而测试算法在计算效率和数值精度上的表现。最后，优化目标的多样化也被纳入考虑。这些函数的目标值从简单的零值（如 Sphere 函数的 0）到复杂的非零值（如 Schwefel 函数）均有所涵盖，从而可以全面考察算法的适应性。

通过在这些具有代表性的测试函数上进行实验，能够全面、系统地评估遗传算法的寻优性能，为其在实际问题中的应用提供科学依据和理论支持。

2.1.1 Sphere 函数

Sphere 函数定义如下：

$$f(x) = \sum_{i=1}^{n} x_i^2$$

该函数是简单的凸函数，具有唯一的全局最优点 $x = 0$，对应最优值 $f(0) = 0$。其主要用于测试优化算法的基本寻优能力。

2.1.2 Ackley 函数

Ackley 函数定义如下：

$$f(x) = -a \exp\left(-b\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(cx_i)\right) + a + e$$

其中常见参数为 $a = 20, b = 0.2, c = 2\pi$。该函数具有大量局部极小值，但全局最优点为 $x = 0$，对应最优值 $f(0) = 0$。

2.1.3 Beale 函数

Beale 函数定义如下：

$$f(x, y) = (1.5 - x + xy)^2 + \left(2.25 - x + xy^2\right)^2 + \left(2.625 - x + xy^3\right)^2$$

此函数为二维函数，唯一全局最优点为 $(x, y) = (3, 0.5)$，对应最优值 $f(3, 0.5) = 0$。

2.1.4 Booth 函数

Booth 函数定义如下：

$$f(x, y) = (x + 2y - 7)^2 + (2x + y - 5)^2$$

此函数为二维函数，最优点为 $(x, y) = (1, 3)$，对应最优值 $f(1, 3) = 0$。

2.1.5 Matyas 函数

Matyas 函数定义如下：

$$f(x, y) = 0.26(x^2 + y^2) - 0.48xy$$

此函数具有椭圆形等高线，全局最优点为 $(x, y) = (0, 0)$，对应最优值 $f(0, 0) = 0$。

2.1.6 Rastrigin 函数

Rastrigin 函数定义如下：

$$f(x) = An + \sum_{i=1}^{n} \left[x_i^2 - A\cos(2\pi x_i) \right]$$

其中 $A = 10$。该函数具有大量局部极小值，全局最优值位于 $x = 0$，对应 $f(0) = 0$。

2.1.7 Rosenbrock 函数

Rosenbrock 函数定义如下：

$$f(x) = \sum_{i=1}^{n-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2 \right]$$

此函数为非凸函数，最优点为 $x = 1$，对应最优值 $f(1, \ldots, 1) = 0$。

2.1.8 Griewank 函数

Griewank 函数定义如下：

$$f(x) = 1 + \frac{1}{4000} \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}} \right)$$

此函数具有复杂的多模态特性，全局最优点为 $x = 0$，对应最优值 $f(0) = 0$。

2.1.9 Schwefel 函数

Schwefel 函数定义如下：

$$f(x) = 418.9829n - \sum_{i=1}^{n} x_i \sin\left(\sqrt{|x_i|} \right)$$

全局最优点约为 $x = 420.9687$，对应最优值 $f(420.9687, \ldots, 420.9687) = 0$。

2.1.10 Zakharov 函数

Zakharov 函数定义如下：

$$f(x) = \sum_{i=1}^{n} x_i^2 + \left(\sum_{i=1}^{n} 0.5ix_i \right)^2 + \left(\sum_{i=1}^{n} 0.5ix_i \right)^4$$

此函数具有平滑的曲面，全局最优点为 $x = 0$，对应最优值 $f(0) = 0$。

2.2 进化算法

进化算法是一类通过模仿生物进化行为的启发式优化算法，主要应用于解决复杂的数学和工程优化问题。这些算法通过模拟自然选择、遗传、变异等生物学过程来搜索最优解，适用于各种不同的应用场景，如工程设计、数据科学、机器学习参数调整等。以下详细介绍了几种基于不同生物行为的进化算法，包括狼群优化算法（Wolf Pack Optimization, WPO）[1]、屎壳郎优化算法（Dung Beetle Optimization, DBO）[2]、粒子群优化算法（Particle Swarm Optimization, PSO）[3]、蜜蜂优化算法（Bee Algorithm, BA）[4]以及鸽子优化算法（Pigeon Inspired Optimization, PIO）[5]，每种算法都具有独特的行为模拟和数学建模方法。

2.2.1 Wolf Pack Optimization (WPO) - 狼群优化算法

狼群优化算法基于狼群的社会结构和捕猎行为，模拟狼群如何通过合作将猎物围困并最终捕捉。在算法中，种群被分为不同的角色：领导者（*alpha*）、次领导者（*beta* 和 *delta*）和跟随者（*omega*）。这种分工使得狼群在探索和开发方面达到平衡。

在狼群优化算法的每一次迭代中，领导者确定移动方向，其余狼根据领导者和次领导者的位置调整自己的位置。领导者的位置由当前最优解决定，次领导者的位置代表局部最优解。位置更新公式考虑了狼与领导者之间的距离和随机探索因素，形式如下：

$$\vec{X}_{t+1} = \vec{X}_t + \vec{r}_1 \cdot (\vec{X}_\alpha - \vec{X}_t) + \vec{r}_2 \cdot (\vec{X}_\beta - \vec{X}_t) + \vec{r}_3 \cdot (\vec{X}_\delta - \vec{X}_t)$$

其中 $\vec{r}_1, \vec{r}_2, \vec{r}_3$ 为随机向量，控制狼群对目标的追踪精度和搜索范围。

随着迭代的进行，搜索范围逐渐缩小，以确保算法能够在全局探索后进行有效的局部搜索，步长的调整公式为：

$$\lambda = \lambda_0 \cdot e^{-kt}$$

其中 λ_0 是初始步长，k 是步长衰减率，t 是迭代次数。这样的设计使得 WPO 在处理多峰和高维优化问题时具有出色的性能。

2.2.2 Dung Beetle Optimization (DBO) - 屎壳郎优化算法

屎壳郎优化算法从屎壳郎的导航行为中汲取灵感，尤其是它们如何使用太阳、月亮和星星来导航并推动球形物体行走的能力。在 DBO 中，每个屎壳郎代表一个解，它们通过感知环境中的梯度信息来更新自己的位置。

屎壳郎根据当前位置感知左右两侧的适应度，确定移动的方向。适应度较高的方向被认为是向优解靠近的方向。这一策略通过以下公式实现：

$$f_{\text{left}} = f(\vec{X}_t + \vec{d}_{\text{left}}), \quad f_{\text{right}} = f(\vec{X}_t + \vec{d}_{\text{right}})$$

$$\vec{X}_{t+1} = \vec{X}_t + \lambda \cdot \vec{d}_{t+1}$$

其中，$\vec{d}_{t+1}$ 是基于适应度较高的方向选择的单位向量，λ 是动态减少的步长，初始较大以探索广泛区域，随后减小以精细搜索。

屎壳郎算法特别适合于动态环境下的优化问题，因为它能快速适应环境变化，并有效找到全局最优或近似最优解。

2.2.3 Particle Swarm Optimization (PSO) - 粒子群优化算法

粒子群优化算法受鸟群觅食行为的启发，其中每个粒子代表潜在的解决方案，通过模仿其他粒子的最佳经验和个体历史最佳经验来更新自己的位置和速度。粒子的速度和位置更新规则如下：

$$\vec{v}_{i,t+1} = \omega \cdot \vec{v}_{i,t} + c_1 \cdot r_1 \cdot (\vec{p}_{i,t} - \vec{x}_{i,t}) + c_2 \cdot r_2 \cdot (\vec{g}_t - \vec{x}_{i,t})$$

$$\vec{x}_{i,t+1} = \vec{x}_{i,t} + \vec{v}_{i,t+1}$$

其中 ω 是惯性权重，有助于平衡全局探索与局部开发；c_1 和 c_2 是学习因子，调节个体和社会学习行为的影响；$\vec{p}_{i,t}$ 是粒子 i 的个体历史最优位置；$\vec{g}_t$ 是当前全局最优位置；r_1 和 r_2 是随机数，增加搜索的随机性。PSO 特别适用于连续函数优化问题，因其简单、有效且容易实现。

2.2.4 Bee Algorithm (BA) - 蜜蜂优化算法

蜜蜂优化算法模拟了蜜蜂的觅食行为，特别是它们如何在花田中搜索并利用资源。算法中的每个“蜜蜂”可以是侦查蜂、采蜜蜂或观察蜂，分别负责探索新的食源、开发已知的食源或从其他蜜蜂学习食源位置。BA 通过以下策略实现优化：

$$\vec{X}_{\text{new}} = \vec{X}_{\text{best}} + \phi \cdot (\vec{X}_{\text{rand1}} - \vec{X}_{\text{rand2}})$$

$$\vec{X}_{\text{local}} = \vec{X}_{\text{current}} + \phi \cdot (\vec{X}_{\text{current}} - \vec{X}_{\text{neighbor}})$$

其中 $\vec{X}_{\text{best}}$ 是当前最佳解，$\vec{X}_{\text{rand1}}$ 和 $\vec{X}_{\text{rand2}}$ 是随机选择的解，ϕ 是控制搜索范围的系数。BA 能够在全局和局部搜索间有效平衡，适用于多模态函数的优化。

2.2.5 Pigeon Inspired Optimization (PIO) - 鸽子优化算法

鸽子优化算法借鉴了鸽子的地磁和视觉导航机制，用于解决复杂的优化问题。PIO 算法分为两个阶段：地磁导航和视觉导航。地磁导航阶段，鸽子使用地磁场确定方向和位置，适用于全局搜索：

$$\vec{X}_{t+1} = \vec{X}_t + \lambda \cdot (\vec{g}_t - \vec{X}_t)$$

视觉导航阶段，当接近目标时，鸽子通过视觉精确定位：

$$\vec{X}_{t+1} = \vec{X}_t + \phi \cdot (\vec{X}_{\text{local}} - \vec{X}_t)$$

其中 $\vec{g}_t$ 是当前全局最优解，$\vec{X}_{\text{local}}$ 是局部最优解，λ 和 ϕ 是导航步长。这种结合长距离导航和精细搜索的策略使 PIO 在高维问题和多峰问题中表现优异。

3 代码实现

我们选择了十个优化问题，使用五种不同的优化算法进行求解和比较。每种算法的终止条件设定为：当优化结果达到最优解的容忍度范围（10^{-3}）或达到最大迭代次数（1000 次）时停止迭代。每个优化问题针对每种算法重复实验 10 次，通过比较平均迭代次数来评估算法的性能，从而分析其优劣。

3.1 函数定义

```python
import numpy as np
# 定义优化问题函数
# Sphere 函数
def sphere(X):
    return sum(x**2 for x in X)

# Ackley 函数
def ackley(X):
    a, b, c = 20, 0.2, 2 * np.pi
    d = len(X)
    sum1 = sum(x**2 for x in X)
    sum2 = sum(np.cos(c * x) for x in X)
    return -a * np.exp(-b * np.sqrt(sum1 / d)) - np.exp(sum2 / d) + a + np.exp(1)

# Beale 函数 (2D)
def beale(X):
    x, y = X
```

```
    return (1.5 - x + x * y)**2 + (2.25 - x + x * y**2)**2 + (2.625 - x + x * y**3)**2

# Booth 函数 (2D)
def booth(X):
    x, y = X
    return (x + 2 * y - 7)**2 + (2 * x + y - 5)**2

# Matyas 函数 (2D)
def matyas(X):
    x, y = X
    return 0.26 * (x**2 + y**2) - 0.48 * x * y

def rastrigin(X):
    n = len(X)
    return 10 * n + sum(x**2 - 10 * np.cos(2 * np.pi * x) for x in X)

def rosenbrock(X):
    return sum(100 * (X[i+1] - X[i]**2)**2 + (1 - X[i])**2 for i in range(len(X) - 1))

def griewank(X):
    sum1 = sum(x**2 for x in X)
    prod1 = np.prod([np.cos(x / np.sqrt(i + 1)) for i, x in enumerate(X)])
    return 1 + sum1 / 4000 - prod1

def schwefel(X):
    return 418.9829 * len(X) - sum(x * np.sin(np.sqrt(abs(x))) for x in X)

def zakharov(X):
    sum1 = sum(x**2 for x in X)
    sum2 = sum(0.5 * (i + 1) * x for i, x in enumerate(X))
    return sum1 + sum2**2 + sum2**4
```

3.2 狼群优化算法

```
import numpy as np
# 狼群优化算法
def wolf_pack_optimization(func, dim, bounds, num_wolves, a_decay, tolerance, optimal_value,
    max_iter):
    # 初始化狼群位置
    wolves = np.random.uniform(bounds[0], bounds[1], (num_wolves, dim))
    scores = np.array([func(w) for w in wolves])

    # 排序狼群 (alpha, beta, delta 为狼群的前三名)
    sorted_indices = np.argsort(scores)
    alpha, beta, delta = wolves[sorted_indices[:3]]
    alpha_score, beta_score, delta_score = scores[sorted_indices[:3]]

    history = [alpha_score]

    for iteration in range(max_iter):
        a = 2 - iteration * (2 / max_iter)  # 动态调整范围因子 a

        for i in range(num_wolves):
            # 计算围攻和随机移动
            r1, r2 = np.random.rand(), np.random.rand()
            A1, C1 = 2 * a * r1 - a, 2 * r2
            D_alpha = abs(C1 * alpha - wolves[i])
            X1 = alpha - A1 * D_alpha

```

```
            r1, r2 = np.random.rand(), np.random.rand()
            A2, C2 = 2 * a * r1 - a, 2 * r2
            D_beta = abs(C2 * beta - wolves[i])
            X2 = beta - A2 * D_beta

            r1, r2 = np.random.rand(), np.random.rand()
            A3, C3 = 2 * a * r1 - a, 2 * r2
            D_delta = abs(C3 * delta - wolves[i])
            X3 = delta - A3 * D_delta

            # 更新狼的位置
            wolves[i] = (X1 + X2 + X3) / 3
            wolves[i] = np.clip(wolves[i], bounds[0], bounds[1])  # 确保位置在边界内

            # 计算新的得分
            score = func(wolves[i])
            if score < scores[i]:
                scores[i] = score

        # 更新 alpha, beta, delta
        sorted_indices = np.argsort(scores)
        alpha, beta, delta = wolves[sorted_indices[:3]]
        alpha_score, beta_score, delta_score = scores[sorted_indices[:3]]

        history.append(alpha_score)

        # 检查收敛条件
        if abs(alpha_score - optimal_value) < tolerance:
            print(f"WOA Converged at iteration {iteration + 1} with alpha_score = {alpha_score}")
            break

    else:
        print(f"WOA Reached maximum iterations ({max_iter}). Best score = {alpha_score}")

    return alpha, alpha_score, history
```

3.3 屎壳郎优化算法

```
import numpy as np

# 屎壳郎优化算法
def dung_beetle_algorithm(func, dim, bounds, num_agents, max_iter, tolerance, optimal_value):
    # 初始化位置和步长
    X = np.random.uniform(bounds[0], bounds[1], (num_agents, dim))
    scores = np.array([func(x) for x in X])
    best_agent = X[np.argmin(scores)]
    best_score = np.min(scores)
    history = [best_score]

    for iteration in range(max_iter):
        for i in range(num_agents):
            # 更新位置（模拟屎壳郎的导航行为）
            step_size = (bounds[1] - bounds[0]) * 0.1 * (1 - iteration / max_iter)  # 动态步长
            direction = np.random.uniform(-1, 1, dim)
            direction /= np.linalg.norm(direction)  # 归一化方向

            # 左右两点（屎壳郎感知的两个方向点）
            X_left = X[i] + step_size * direction
            X_right = X[i] - step_size * direction
```

```
22
23              # 保证两点在边界内
24              X_left = np.clip(X_left, bounds[0], bounds[1])
25              X_right = np.clip(X_right, bounds[0], bounds[1])
26
27              # 计算左右两点的目标函数值
28              score_left = func(X_left)
29              score_right = func(X_right)
30
31              # 更新屎壳郎位置
32              if score_left < score_right:
33                  X[i] = X_left
34              else:
35                  X[i] = X_right
36
37              # 保证屎壳郎位置在边界内
38              X[i] = np.clip(X[i], bounds[0], bounds[1])
39
40          # 更新全局最优解
41          scores = np.array([func(x) for x in X])
42          current_best_score = np.min(scores)
43          if current_best_score < best_score:
44              best_score = current_best_score
45              best_agent = X[np.argmin(scores)]
46
47          history.append(best_score)
48
49          # 检查收敛条件
50          if abs(best_score - optimal_value) < tolerance:
51              print(f"Converged at iteration {iteration + 1} with best_score = {best_score}")
52              break
53
54      else:
55          print(f"Reached maximum iterations ({max_iter}). Best score = {best_score}")
56
57      return best_agent, best_score, history
```

3.4 粒子群优化算法

```
1  import numpy as np
2  # 粒子群优化算法
3  def particle_swarm_optimization(func, dim, bounds, num_particles, w, c1, c2, tolerance,
       optimal_value, max_iter):
4      # 初始化粒子位置和速度
5      X = np.random.uniform(bounds[0], bounds[1], (num_particles, dim))
6      V = np.random.uniform(-abs(bounds[1] - bounds[0]) * 0.1, abs(bounds[1] - bounds[0]) * 0.1, (
           num_particles, dim))
7      personal_best_X = np.copy(X)
8      personal_best_scores = np.array([func(x) for x in X])
9      global_best_X = personal_best_X[np.argmin(personal_best_scores)]
10     global_best_score = np.min(personal_best_scores)
11     history = [global_best_score]
12
13     for iteration in range(max_iter):
14         prev_best_score = global_best_score  # 记录上一次的最优值
15
16         for i in range(num_particles):
17             r1, r2 = np.random.rand(dim), np.random.rand(dim)
```

```
            V[i] = w * V[i] + c1 * r1 * (personal_best_X[i] - X[i]) + c2 * r2 * (global_best_X - X[i
                ])
            X[i] += V[i]
            X[i] = np.clip(X[i], bounds[0], bounds[1])  # 保证粒子位置在边界内

            score = func(X[i])
            if score < personal_best_scores[i]:
                personal_best_X[i] = X[i]
                personal_best_scores[i] = score

        global_best_X = personal_best_X[np.argmin(personal_best_scores)]
        global_best_score = np.min(personal_best_scores)
        history.append(global_best_score)

        # 检查是否满足收敛条件
        if abs(global_best_score - optimal_value) < tolerance:
            print(f"Converged at iteration {iteration + 1} with global_best_score = {
                global_best_score}")
            break

    else:
        print(f"Reached maximum iterations ({max_iter}). Best score = {global_best_score}")

    return global_best_X, global_best_score, history
```

3.5 蜜蜂优化算法

```
import numpy as np
import time

def bee_algorithm(func, dim, bounds, num_bees, elite_bees, tolerance, optimal_value, max_iter, seed=
    None):
    if seed is not None:
        np.random.seed(seed)  # 固定随机性
    else:
        np.random.seed(int(time.time() * 1000) % (2**32))  # 动态随机性

    bees = np.random.uniform(bounds[0], bounds[1], (num_bees, dim))
    scores = np.array([func(b) for b in bees])
    best_bee = bees[np.argmin(scores)]
    best_score = np.min(scores)
    history = [best_score]
    initial_range = 0.1  # 初始探索范围

    for iteration in range(max_iter):
        exploration_range = initial_range * (1 - iteration / max_iter)  # 动态探索范围
        for i in range(num_bees):
            for _ in range(elite_bees):
                new_bee = bees[i] + np.random.uniform(-exploration_range, exploration_range, dim)
                # 边界反弹策略
                for j in range(dim):
                    if new_bee[j] < bounds[0]:
                        new_bee[j] = bounds[0] + (bounds[0] - new_bee[j])
                    elif new_bee[j] > bounds[1]:
                        new_bee[j] = bounds[1] - (new_bee[j] - bounds[1])
                score = func(new_bee)
                if score < scores[i]:
                    scores[i] = score
                    bees[i] = new_bee
```

```
32
33          # 更新全局最优解
34          current_best_score = np.min(scores)
35          if current_best_score < best_score:
36              best_score = current_best_score
37              best_bee = bees[np.argmin(scores)]
38
39          history.append(best_score)
40
41          # 动态调整容差
42          current_tolerance = tolerance * (1 - iteration / max_iter)
43          if abs(best_score - optimal_value) < current_tolerance:
44              print(f"Bee Converged at iteration {iteration + 1} with best_score = {best_score}")
45              break
46      else:
47          print(f"Bee Reached maximum iterations ({max_iter}). Best score = {best_score}")
48
49      return best_bee, best_score, history
```

3.6 鸽子优化算法

```
1  import numpy as np
2  import time
3
4  def pigeon_optimization(func, dim, bounds, num_pigeons, alpha, gamma, beta, tolerance, optimal_value
       , max_iter, seed=None):
5      # 动态随机性或可重复性设置
6      if seed is not None:
7          np.random.seed(seed)
8      else:
9          np.random.seed(int(time.time() * 1000) % (2**32))
10
11     pigeons = np.random.uniform(bounds[0], bounds[1], (num_pigeons, dim))  # 初始化鸽子群
12     scores = np.array([func(p) for p in pigeons])  # 计算每只鸽子的初始分数
13     best_pigeon = pigeons[np.argmin(scores)]  # 初始化全局最优鸽子
14     best_score = np.min(scores)  # 初始化全局最优分数
15     history = [best_score]  # 记录历史最优分数
16
17     initial_alpha = alpha
18     initial_gamma = gamma
19
20     for iteration in range(max_iter):
21         # 动态调整步长 (alpha) 和缩放因子 (gamma)
22         alpha = initial_alpha * (1 - iteration / max_iter)
23         gamma = initial_gamma * (1 - iteration / max_iter)
24
25         for i in range(num_pigeons):
26             # 更新每只鸽子的位置 (引入局部扰动项)
27             step = np.random.uniform(-alpha, alpha, dim) + beta * (best_pigeon - pigeons[i])
28             pigeons[i] += gamma * step
29
30             # 边界反弹机制
31             for j in range(dim):
32                 if pigeons[i][j] < bounds[0]:
33                     pigeons[i][j] = bounds[0] + np.random.uniform(0, alpha)
34                 elif pigeons[i][j] > bounds[1]:
35                     pigeons[i][j] = bounds[1] - np.random.uniform(0, alpha)
36
37             # 计算新的分数并更新个体最优
```

```
            score = func(pigeons[i])
            if score < scores[i]:
                scores[i] = score

        # 更新全局最优解
        current_best_score = np.min(scores)
        if current_best_score < best_score:
            best_score = current_best_score
            best_pigeon = pigeons[np.argmin(scores)]

        # 记录当前迭代的全局最优分数
        history.append(best_score)

        # 检查收敛条件
        if np.abs(best_score - optimal_value) < tolerance:
            print(f"Pigeon Converged at iteration {iteration + 1} with best_score = {best_score}")
            break
    else:
        print(f"Pigeon Reached maximum iterations ({max_iter}). Best score = {best_score}")

    return best_pigeon, best_score, history
```

3.7 主函数

```
import numpy as np
import matplotlib.pyplot as plt
from WPO import wolf_pack_optimization
from DBO import dung_beetle_algorithm
from PSO import particle_swarm_optimization
from BEE import bee_algorithm
from PIO import pigeon_optimization
from fun import sphere,ackley,beale,booth,matyas,rastrigin,rosenbrock,griewank,schwefel,zakharov
import numpy as np
import matplotlib.pyplot as plt
import os
# 修改的主函数
def optimize_and_save_plots():
    # 定义优化算法
    algorithms = {
        "PSO": particle_swarm_optimization,
        "WOA": wolf_pack_optimization,
        "Bee": bee_algorithm,
        "Pigeon": pigeon_optimization,
        "DBA": dung_beetle_algorithm  # 添加屎壳郎优化算法
    }

    # 定义测试函数
    test_functions = {
        "Sphere": (sphere, [-20, 20], 0),
        "Ackley": (ackley, [-20, 20], 0),
        "Beale": (beale, [-20, 20], 0),
        "Booth": (booth, [-20, 20], 0),
        "Matyas": (matyas, [-20, 20], 0),
        "Rastrigin": (rastrigin, [-5.12, 5.12], 0),
        "Rosenbrock": (rosenbrock, [-5, 10], 0),
        "Griewank": (griewank, [-600, 600], 0),
        "Schwefel": (schwefel, [-500, 500], 0),
        "Zakharov": (zakharov, [-5, 10], 0)
    }
```

```
36
37      # 参数设置
38      num_particles = 50
39      tolerance = 1e-3
40      max_iter = 1000
41
42
43      for func_name, (func, bounds, optimal_value) in test_functions.items():
44          dim = 2 if func_name in ["Beale", "Booth", "Matyas"] else 5
45
46          for alg_name, alg_func in algorithms.items():
47              print(f"Running {alg_name} on {func_name}...")
48              plt.figure(figsize=(10, 8))  # 每种函数与算法组合绘制一张图
49
50              for run in range(10):  # 每种算法运行 10 次
51                  if alg_name == "PSO":
52                      _, best_score, history = alg_func(func, dim, bounds, num_particles, 0.5, 1.5,
                            1.5, tolerance, optimal_value, max_iter)
53                  elif alg_name == "Bee":
54                      _, best_score, history = alg_func(func, dim, bounds, num_particles, 5, tolerance
                            , optimal_value, max_iter)
55                  elif alg_name == "Pigeon":
56                      _, best_score, history = alg_func(func, dim, bounds, num_particles, 0.1, 1.0,
                            0.5, tolerance, optimal_value, max_iter)
57                  elif alg_name == "WOA":
58                      _, best_score, history = alg_func(func, dim, bounds, num_particles, 2, tolerance
                            , optimal_value, max_iter)
59                  elif alg_name == "DBA":
60                      _, best_score, history = alg_func(func, dim, bounds, num_particles, max_iter,
                            tolerance, optimal_value)
61                  else:
62                      raise ValueError(f"Unknown algorithm: {alg_name}")
63
64                  # 绘制每次运行的结果
65                  plt.plot(history, label=f"Run {run+1}")
66
67              plt.xlabel("Iterations")
68              plt.ylabel("Best Score")
69              plt.title(f"{alg_name} on {func_name}")
70              plt.legend()
71              plt.grid()
72              filename = os.path.join('/Users/youngbean/Desktop/opti/image', f"{func_name}_{alg_name}.
                    png")
73              plt.savefig(filename)  # 保存图像
74              plt.close()
75
76  # 执行优化并保存收敛图像
77  optimize_and_save_plots()
```

4 实验结果

4.1 狼群优化算法

表 1: 狼群优化算法在各函数上的迭代次数

Times	**Sphere**	**Ackley**	**Beale**	**Booth**	**Matyas**	**Rastrigin**	**Rosenbrock**	**Griewank**	**Schwefel**	**Zakharov**
Run 1	13	20	5	1000	4	43	1000	33	1000	21
Run 2	13	19	3	1000	4	24	1000	31	1000	27
Run 3	11	21	7	1000	4	32	1000	34	1000	18
Run 4	12	24	12	1000	4	17	1000	45	1000	20
Run 5	11	20	10	1000	3	37	1000	27	1000	17
Run 6	10	17	4	1000	3	59	1000	31	1000	14
Run 7	12	21	9	11	2	32	1000	38	1000	26
Run 8	12	24	5	1000	2	24	1000	21	1000	28
Run 9	11	20	26	1000	3	28	1000	62	1000	20
Run 10	12	23	1000	1000	3	13	1000	38	1000	25
Average	11.8	20.9	108.1	1011.2	3.2	31.1	1000	36.0	1000	21.7

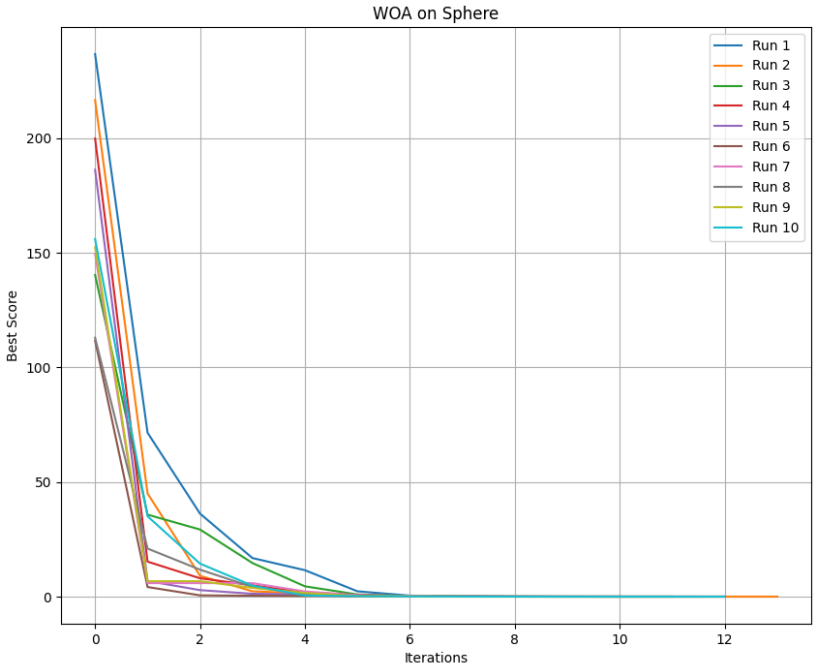

图 1: 狼群优化算法 Sphere 函数迭代结果图

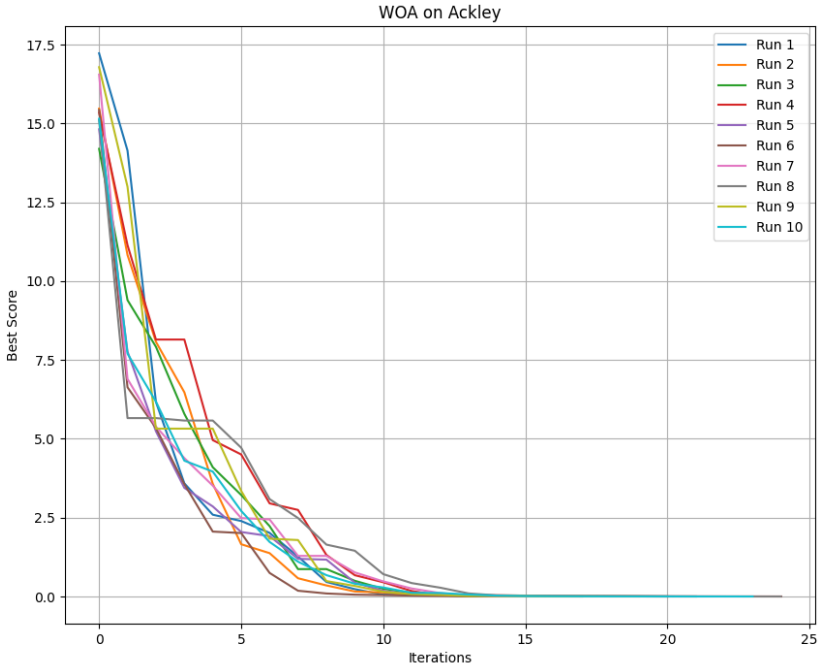

图 2: 狼群优化算法 Ackley 函数迭代结果图

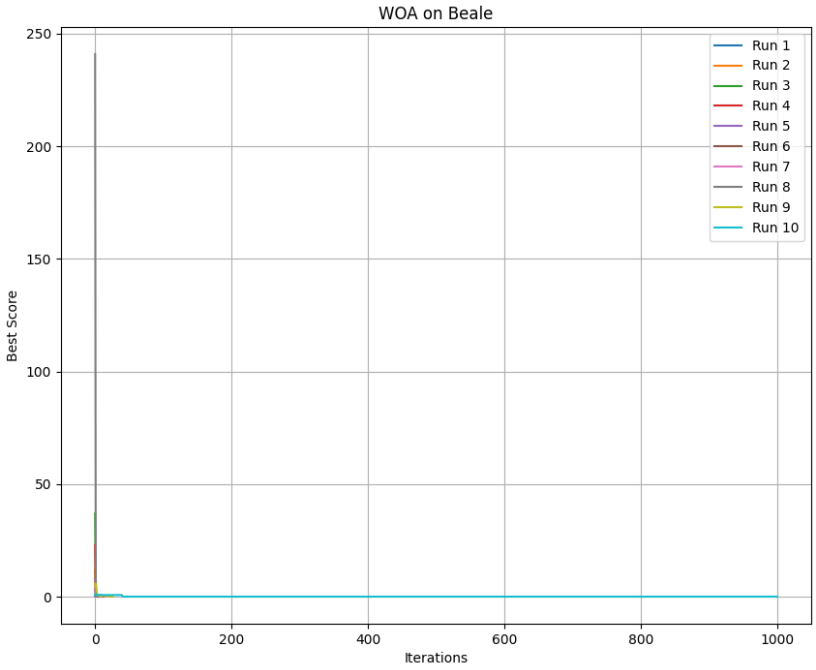

图 3: 狼群优化算法 Beale 函数迭代结果图

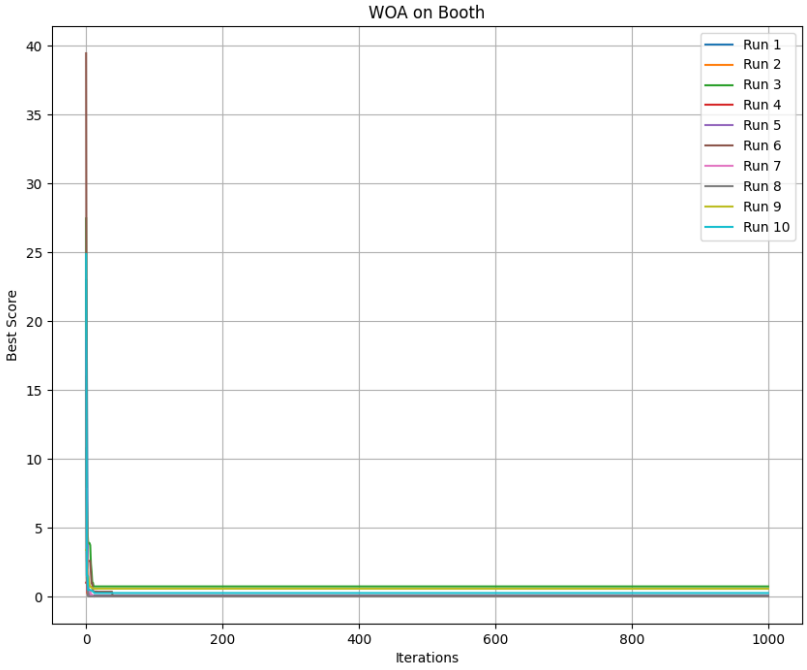

图 4: 狼群优化算法 Booth 函数迭代结果图

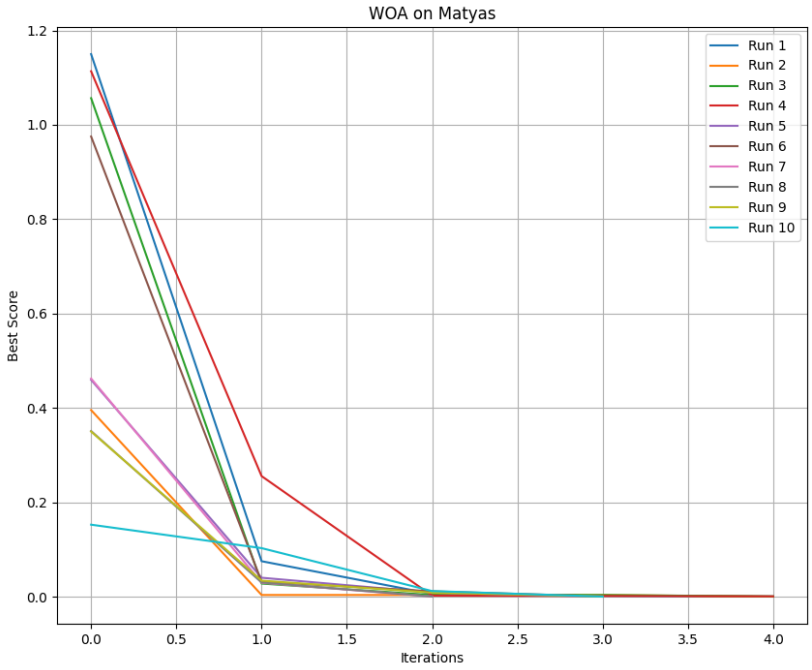

图 5: 狼群优化算法 Matyas 函数迭代结果图

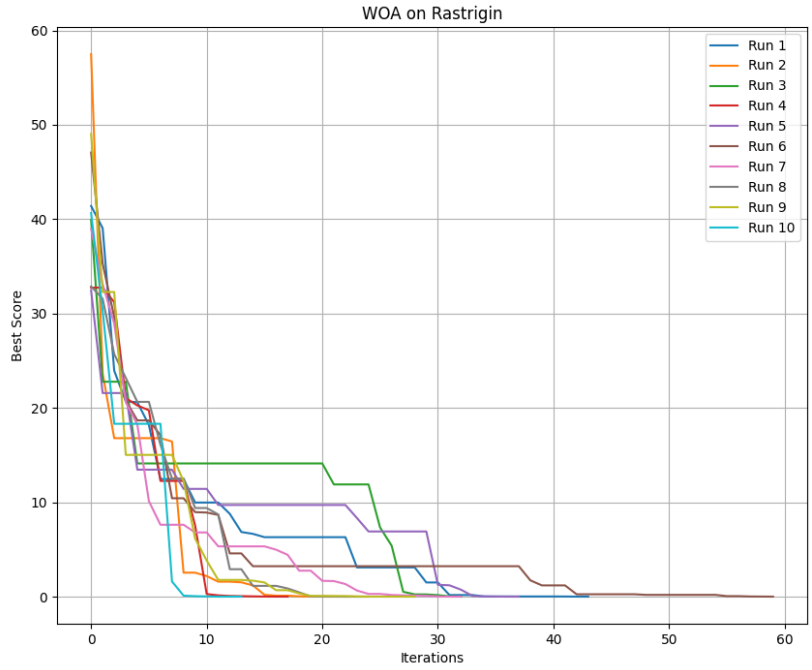

图 6: 狼群优化算法 Rastrigin 函数迭代结果图

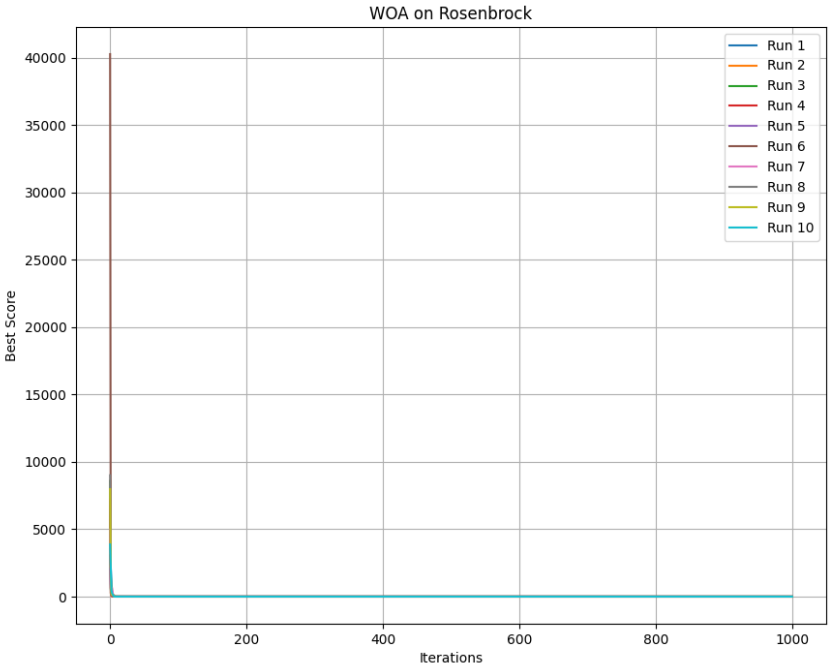

图 7: 狼群优化算法 Rosenbrock 函数迭代结果图

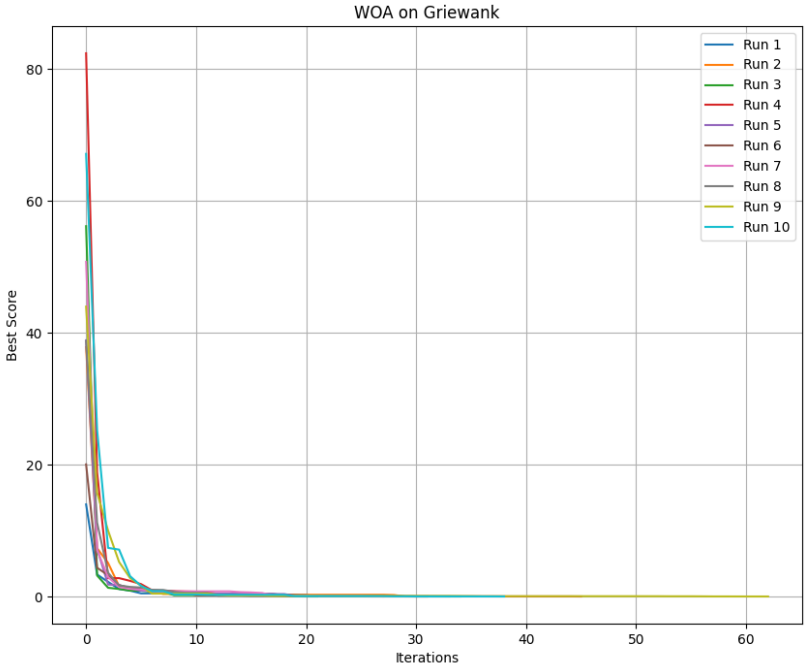

图 8: 狼群优化算法 Griewank 函数迭代结果图

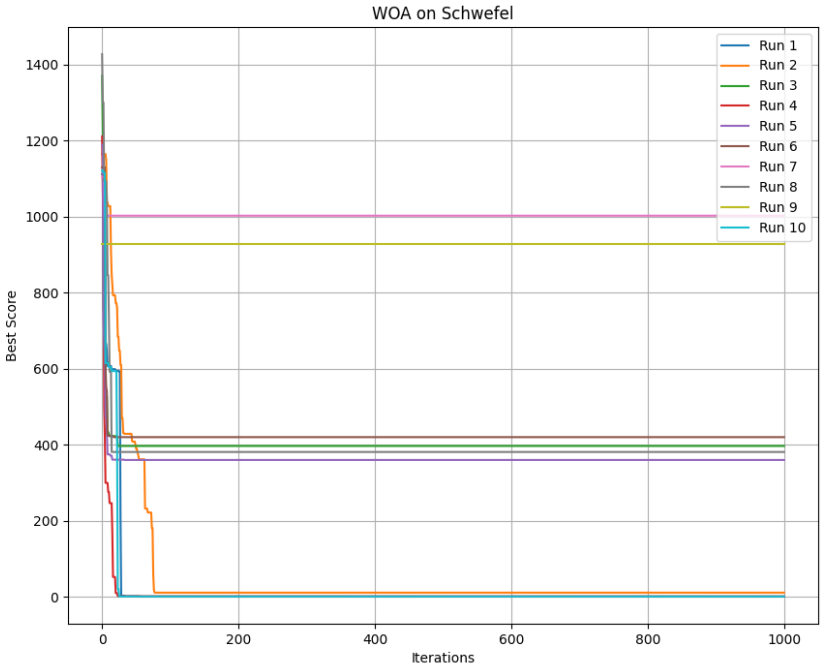

图 9: 狼群优化算法 Schwefel 函数迭代结果图

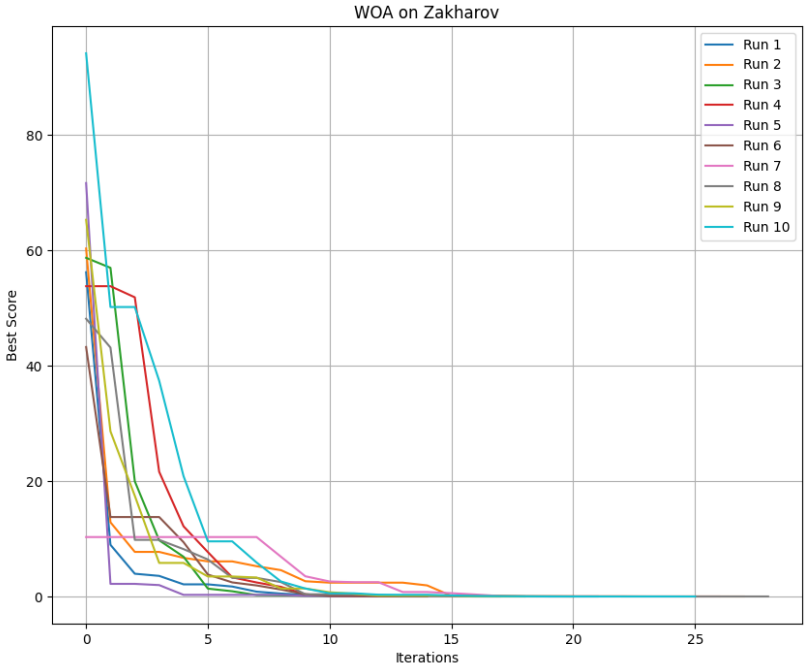

图 10: 狼群优化算法 Zakharov 函数迭代结果图

4.2 屎壳郎优化算法

表 2: 屎壳郎优化算法在各函数上的迭代次数

Times	**Sphere**	**Ackley**	**Beale**	**Booth**	**Matyas**	**Rastrigin**	**Rosenbrock**	**Griewank**	**Schwefel**	**Zakharov**
Run 1	982	1000	577	422	55	1000	1000	1000	1000	976
Run 2	987	1000	879	544	30	1000	1000	1000	1000	973
Run 3	981	1000	704	577	32	1000	1000	1000	1000	963
Run 4	986	1000	682	715	14	1000	1000	1000	1000	971
Run 5	953	1000	778	291	79	1000	1000	1000	1000	934
Run 6	986	1000	188	109	182	1000	1000	1000	1000	972
Run 7	970	1000	519	9	65	1000	1000	1000	1000	950
Run 8	980	1000	400	650	41	1000	1000	1000	1000	982
Run 9	977	1000	681	601	25	1000	1000	1000	1000	959
Run 10	987	1000	491	646	60	1000	1000	1000	1000	971
Average	979.2	1000	589.9	456.4	58.8	1000	1000	1000	1000	965.1

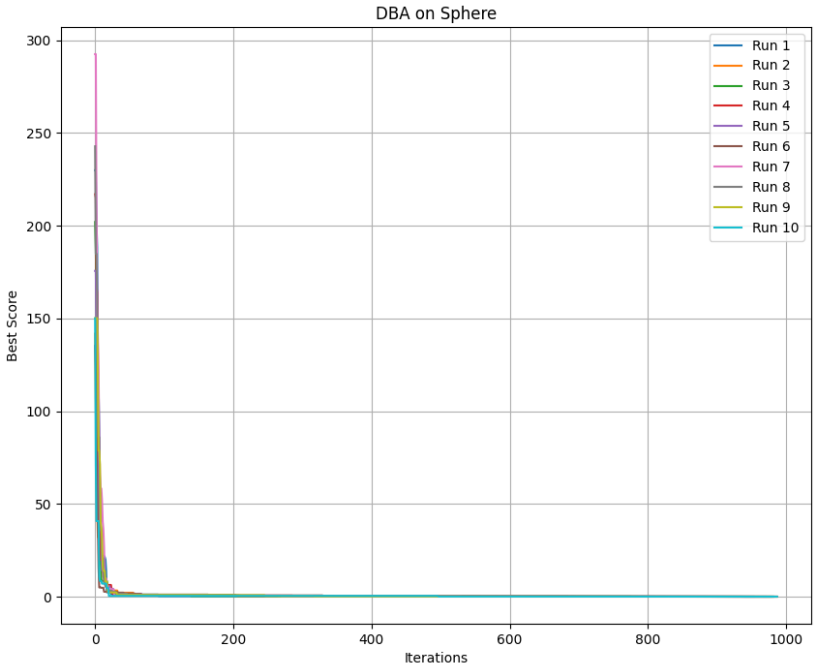

图 11: 屎壳郎优化算法 Sphere 函数迭代结果图

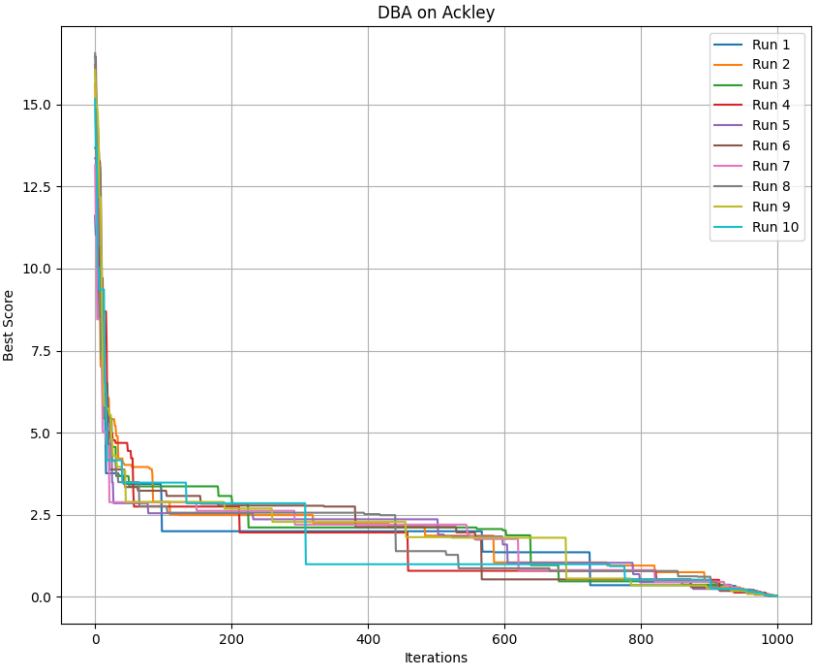

图 12: 屎壳郎优化算法 Ackley 函数迭代结果图

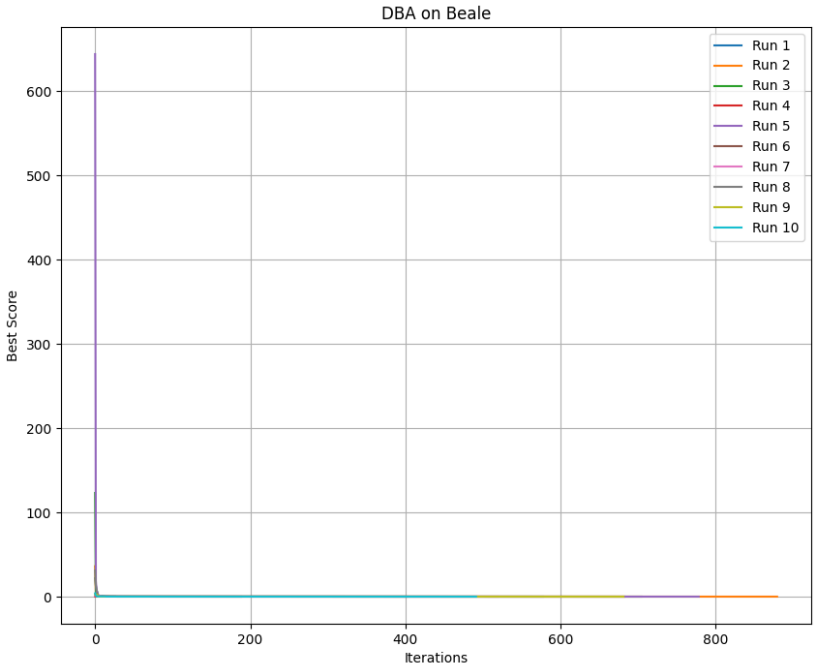

图 13: 屎壳郎优化算法 Beale 函数迭代结果图

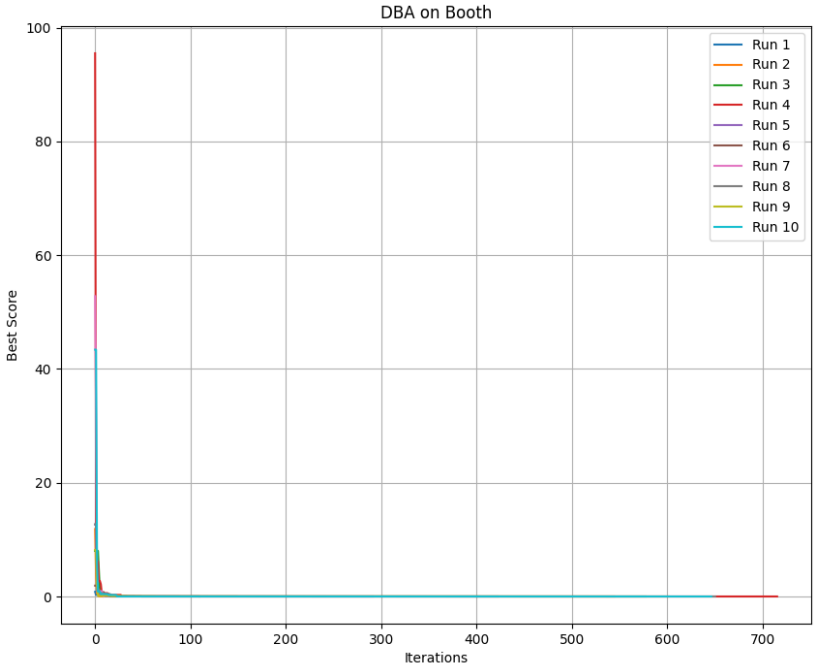

图 14: 屎壳郎优化算法 Booth 函数迭代结果图

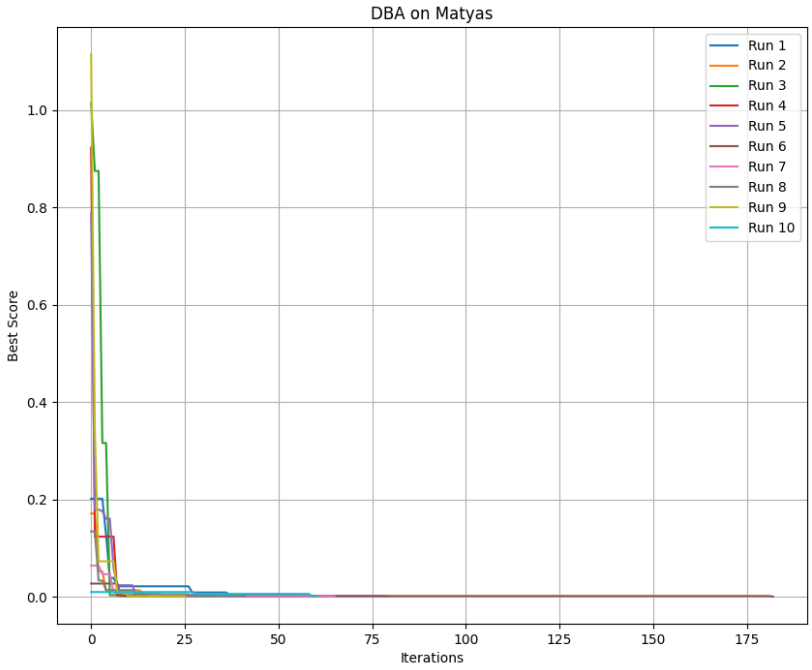

图 15: 屎壳郎优化算法 Matyas 函数迭代结果图

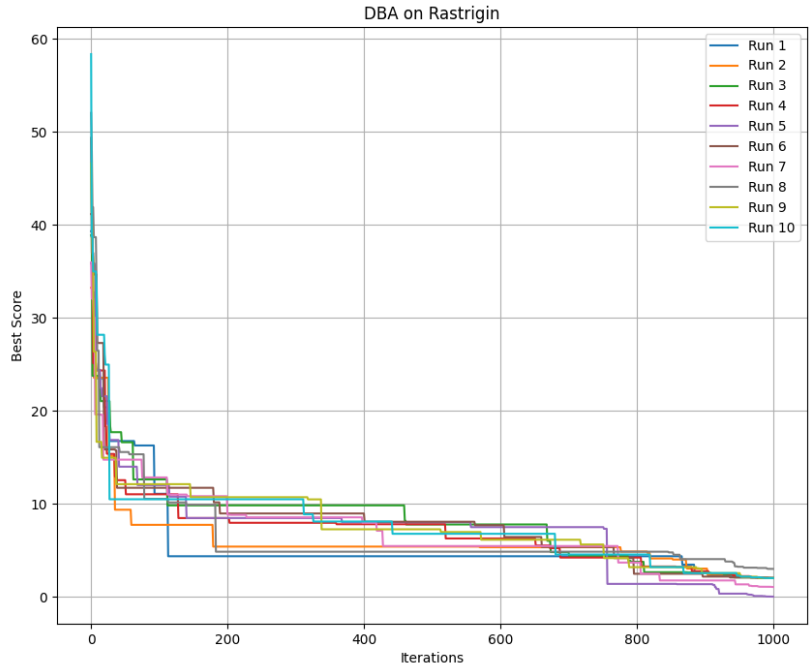

图 16: 屎壳郎优化算法 Rastrigin 函数迭代结果图

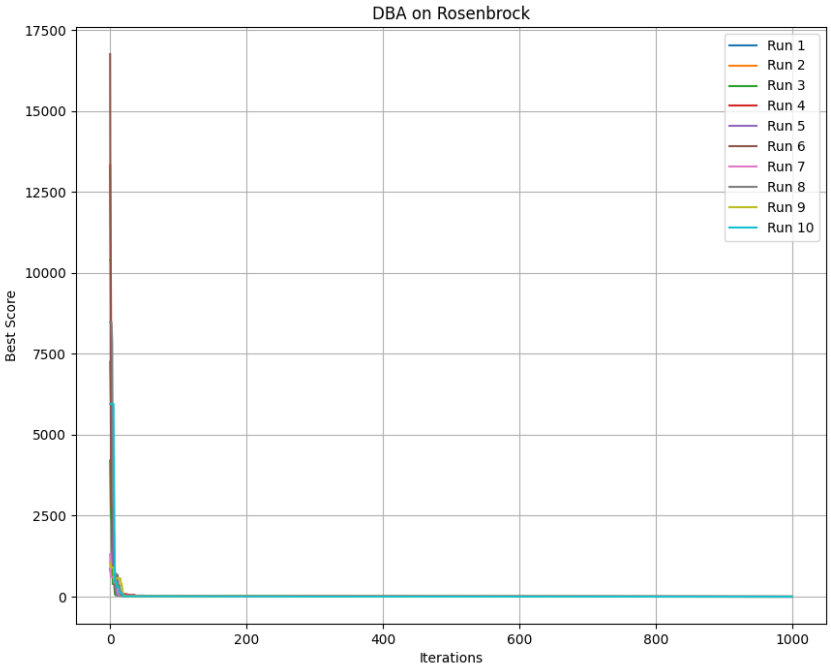

图 17: 屎壳郎优化算法 Rosenbrock 函数迭代结果图

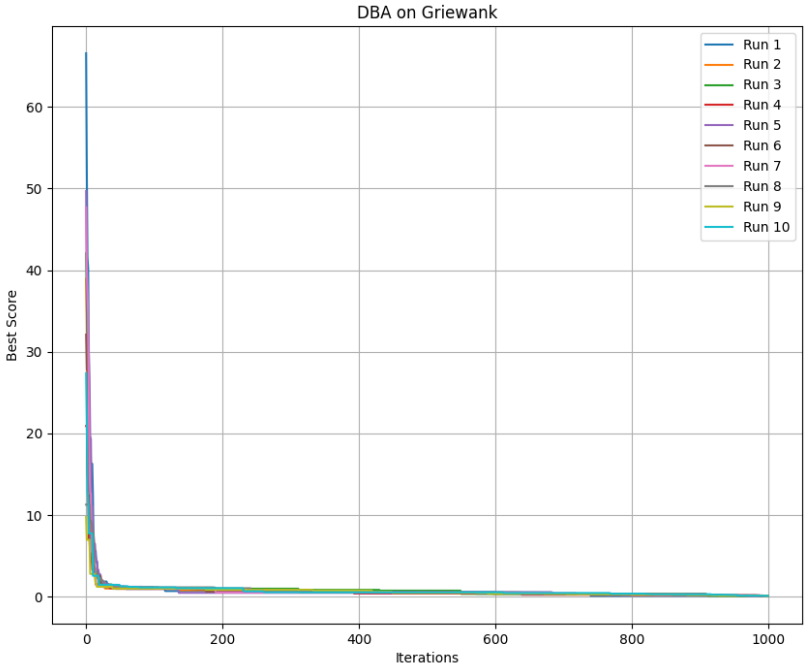

图 18: 屎壳郎优化算法 Griewank 函数迭代结果图

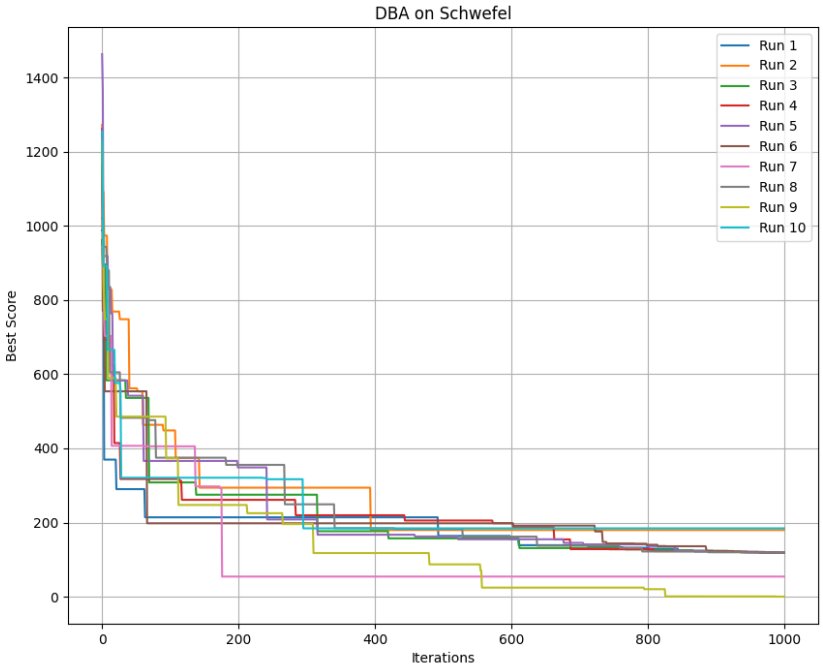

图 19: 屎壳郎优化算法 Schwefel 函数迭代结果图

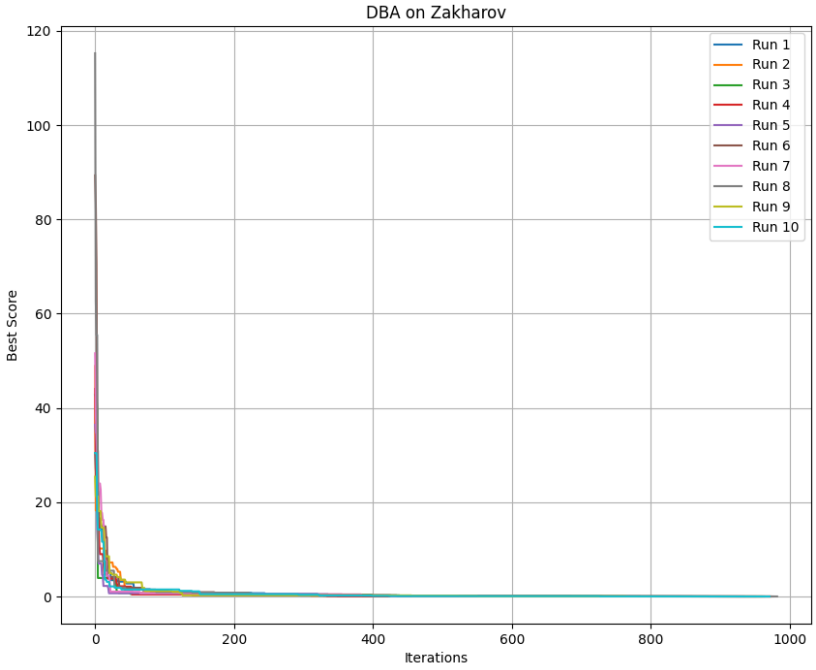

图 20: 屎壳郎优化算法 Zakharov 函数迭代结果图

4.3 粒子群优化算法

表 3: 粒子群优化算法在各函数上的迭代次数

Times	**Sphere**	**Ackley**	**Beale**	**Booth**	**Matyas**	**Rastrigin**	**Rosenbrock**	**Griewank**	**Schwefel**	**Zakharov**
Run 1	23	42	15	15	8	1000	668	1000	40	27
Run 2	26	46	12	15	8	1000	840	1000	1000	27
Run 3	27	43	16	8	8	1000	807	1000	1000	28
Run 4	27	44	14	17	10	1000	1000	1000	1000	22
Run 5	27	44	12	17	10	1000	832	1000	1000	25
Run 6	27	46	18	16	11	1000	823	1000	1000	23
Run 7	27	43	14	11	9	1000	1000	1000	1000	29
Run 8	25	41	14	10	8	1000	823	1000	1000	24
Run 9	24	47	16	16	10	1000	1000	1000	1000	27
Run 10	26	44	15	14	11	1000	729	1000	1000	27
Average	25.7	44.0	14.9	13.9	9.5	1000	852.2	1000	940.0	25.9

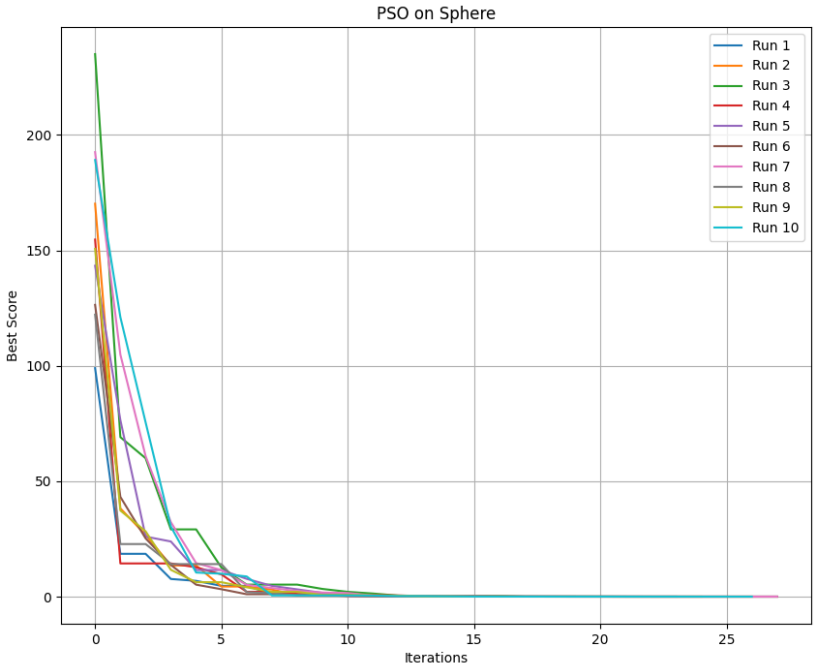

图 21: 粒子群优化算法 Sphere 函数迭代结果图

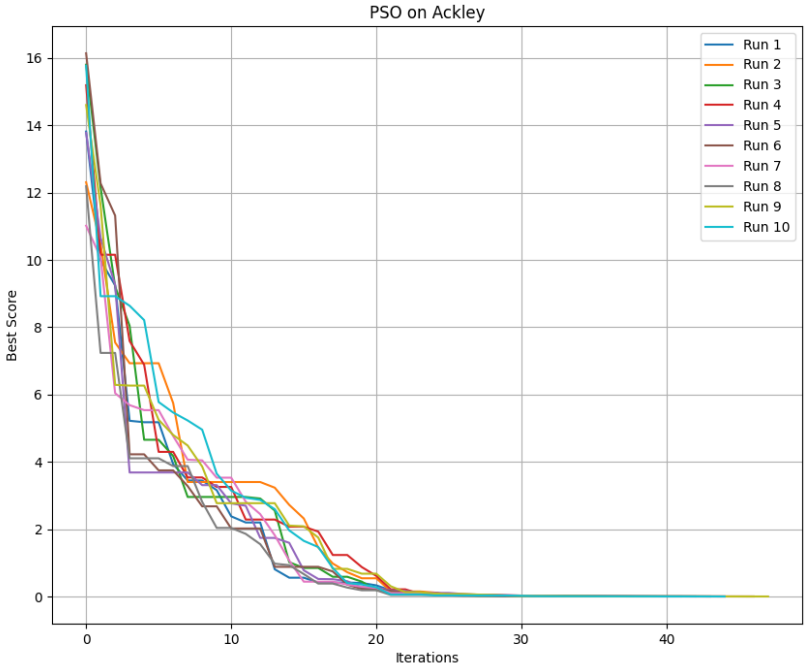

图 22: 粒子群优化算法 Ackley 函数迭代结果图

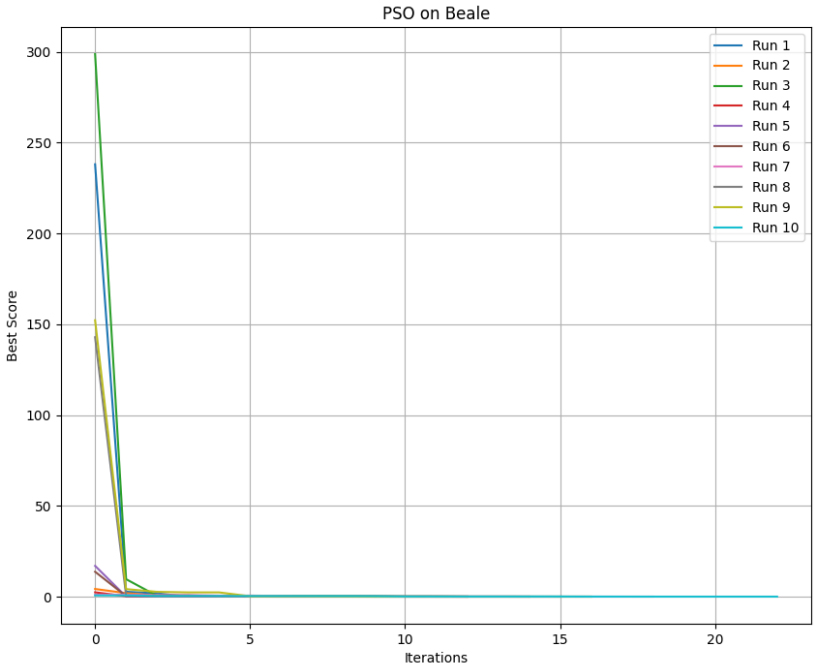

图 23: 粒子群优化算法 Beale 函数迭代结果图

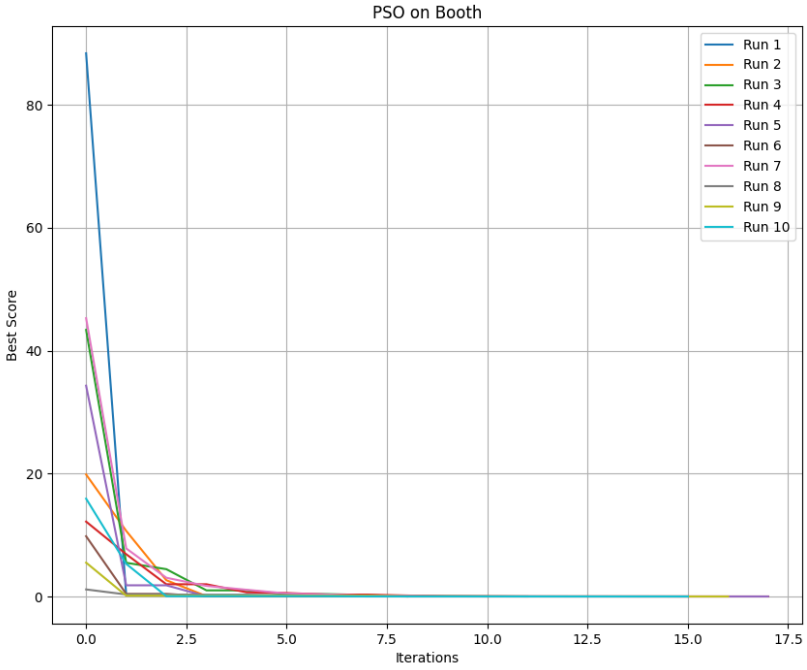

图 24: 粒子群优化算法 Booth 函数迭代结果图

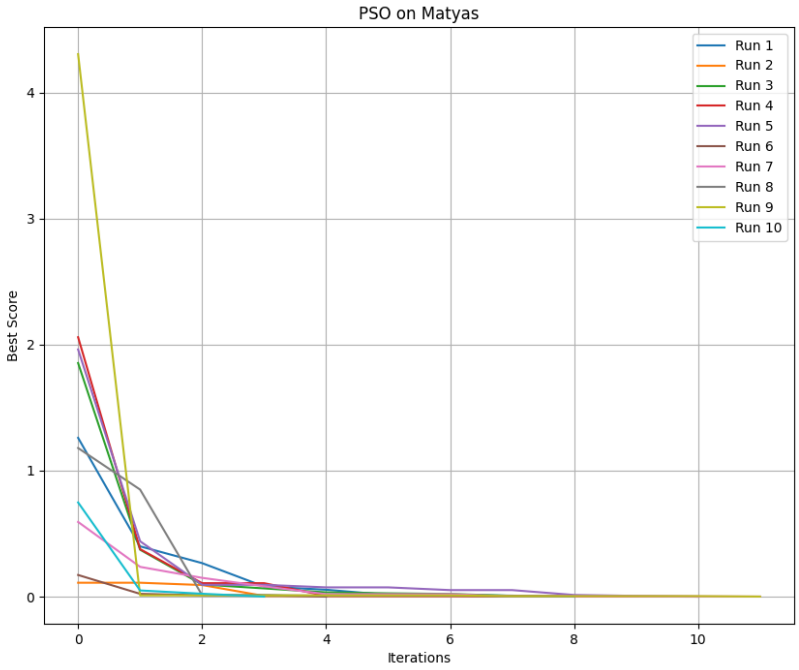

图 25: 粒子群优化算法 Matyas 函数迭代结果图

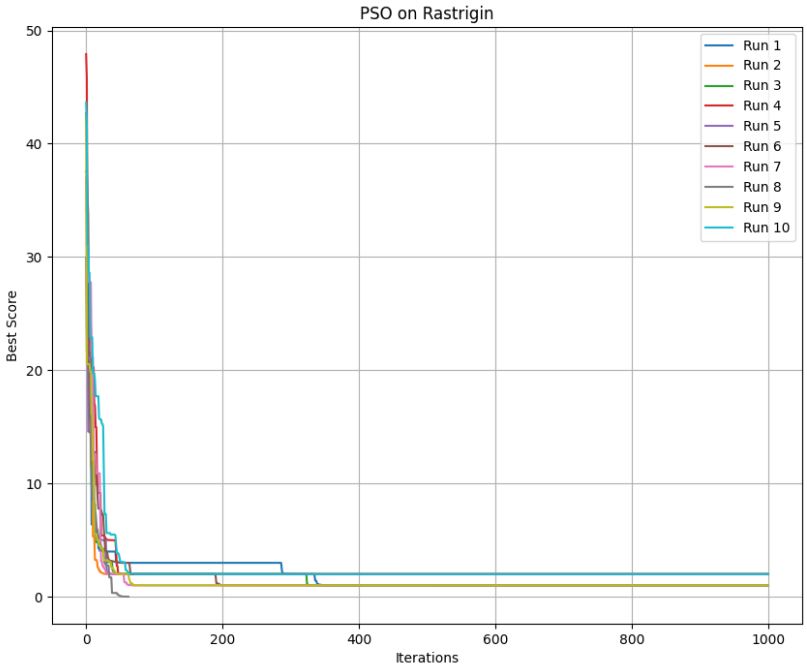

图 26: 粒子群优化算法 Rastrigin 函数迭代结果图

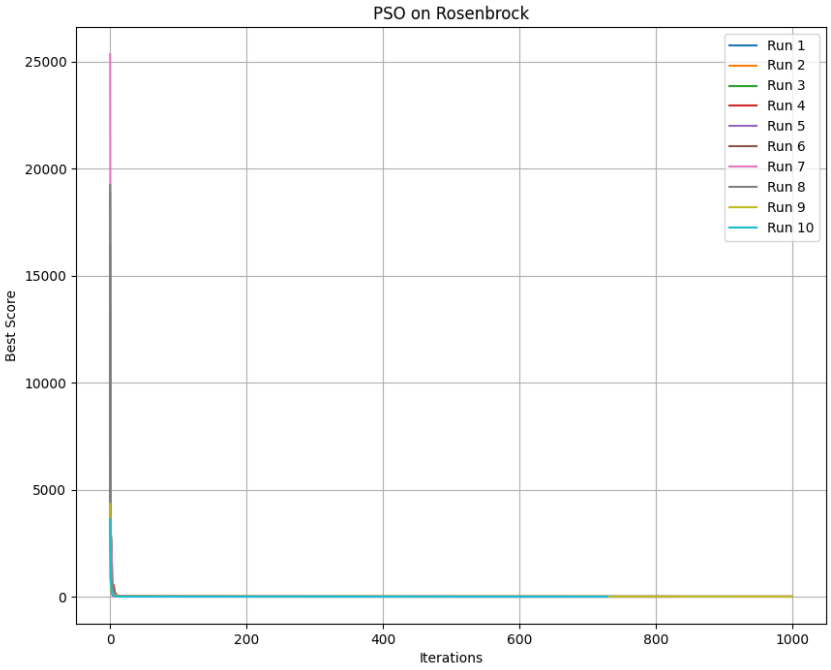

图 27: 粒子群优化算法 Rosenbrock 函数迭代结果图

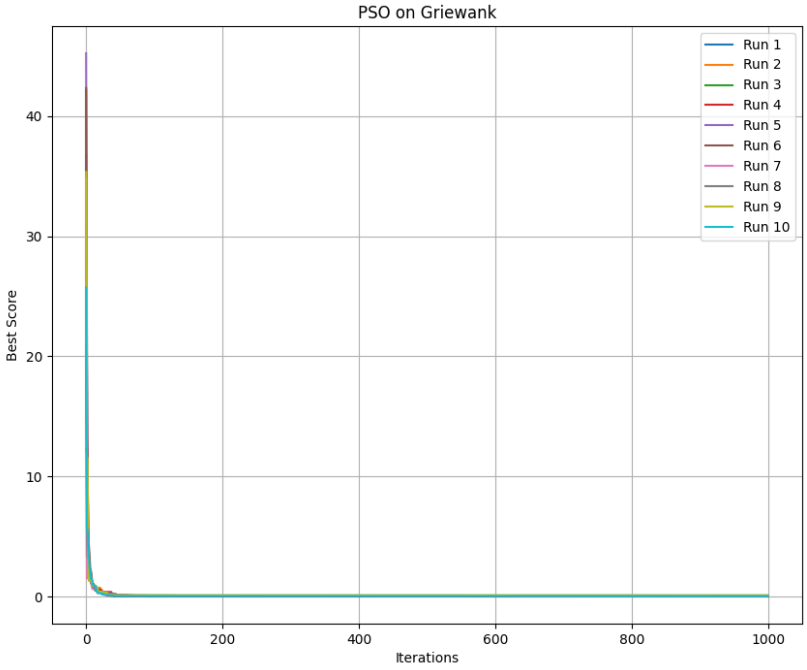

图 28: 粒子群优化算法 Griewank 函数迭代结果图

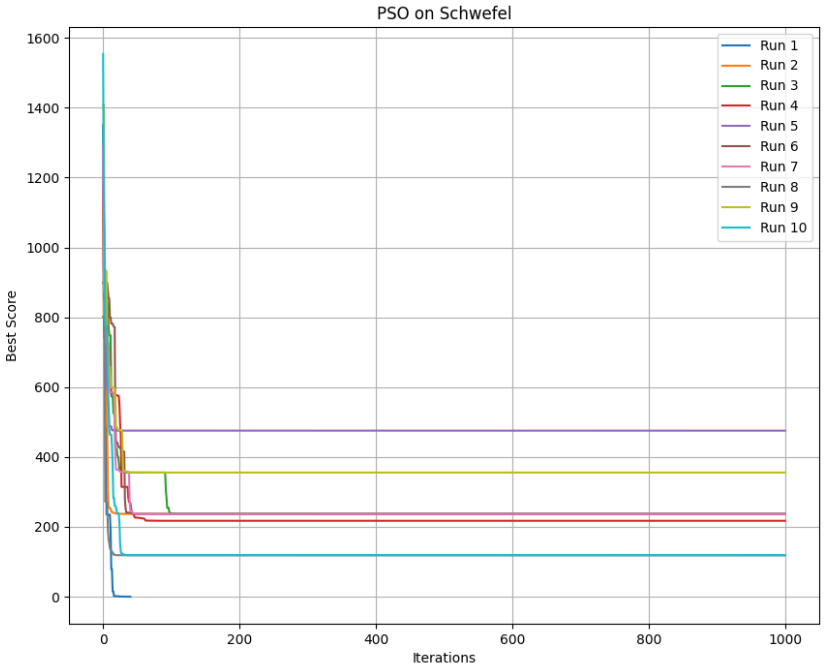

图 29: 粒子群优化算法 Schwefel 函数迭代结果图

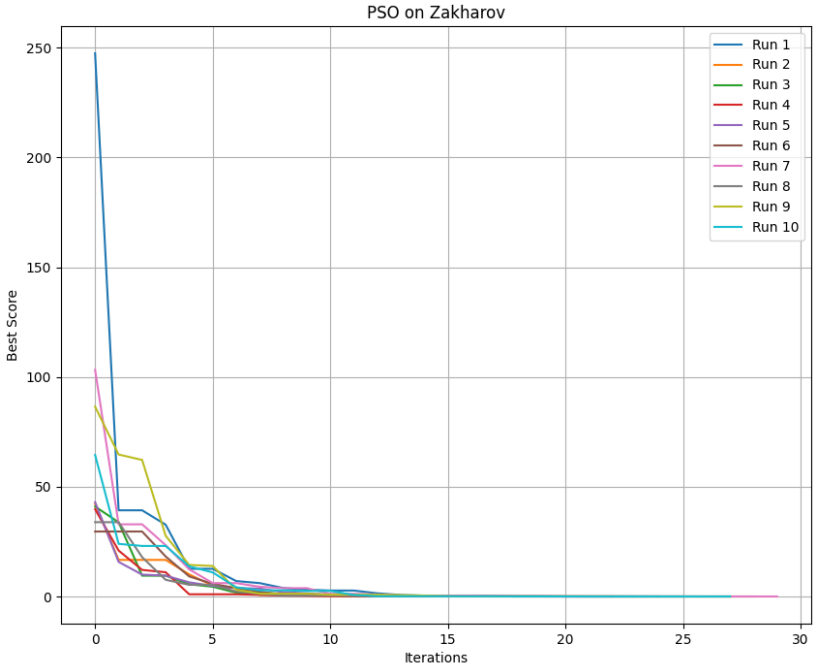

图 30: 粒子群优化算法 Zakharov 函数迭代结果图

4.4 蜜蜂优化算法

表 4: 蜜蜂优化算法在各函数上的迭代次数

Times	**Sphere**	**Ackley**	**Beale**	**Booth**	**Matyas**	**Rastrigin**	**Rosenbrock**	**Griewank**	**Schwefel**	**Zakharov**
Run 1	73	1000	47	14	15	1000	1000	1000	1000	168
Run 2	226	1000	64	36	16	1000	1000	1000	1000	55
Run 3	153	1000	50	50	20	1000	1000	1000	1000	143
Run 4	200	1000	58	28	26	1000	1000	1000	1000	116
Run 5	167	1000	51	26	19	1000	1000	1000	1000	116
Run 6	105	1000	49	31	58	1000	1000	1000	1000	171
Run 7	216	1000	36	28	39	1000	1000	1000	1000	174
Run 8	131	1000	42	86	23	1000	1000	1000	1000	121
Run 9	144	1000	14	3	34	1000	1000	1000	1000	132
Run 10	232	1000	52	23	4	1000	1000	1000	1000	188
Average	164.7	1000	46.7	32.5	25.4	1000	1000	1000	1000	138.4

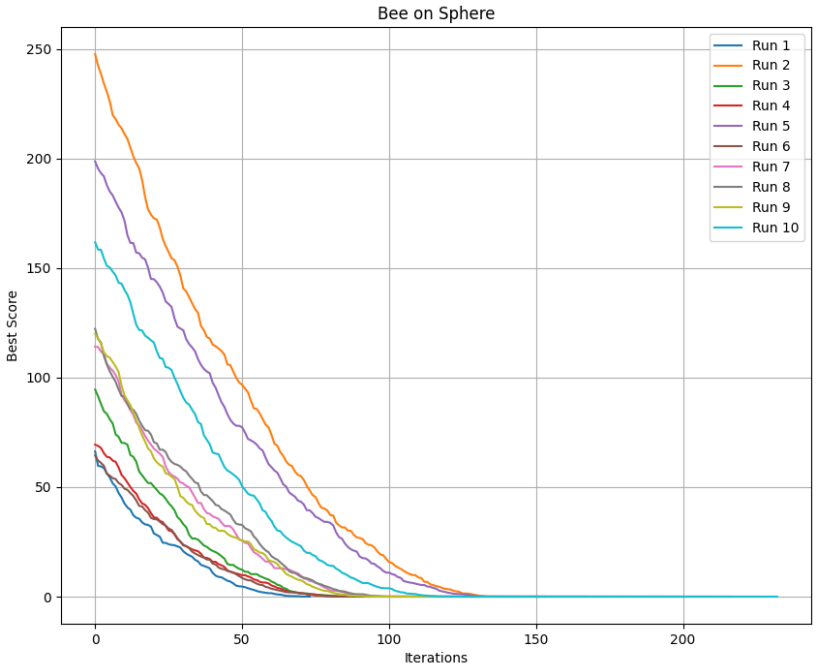

图 31: 蜜蜂优化算法 Sphere 函数迭代结果图

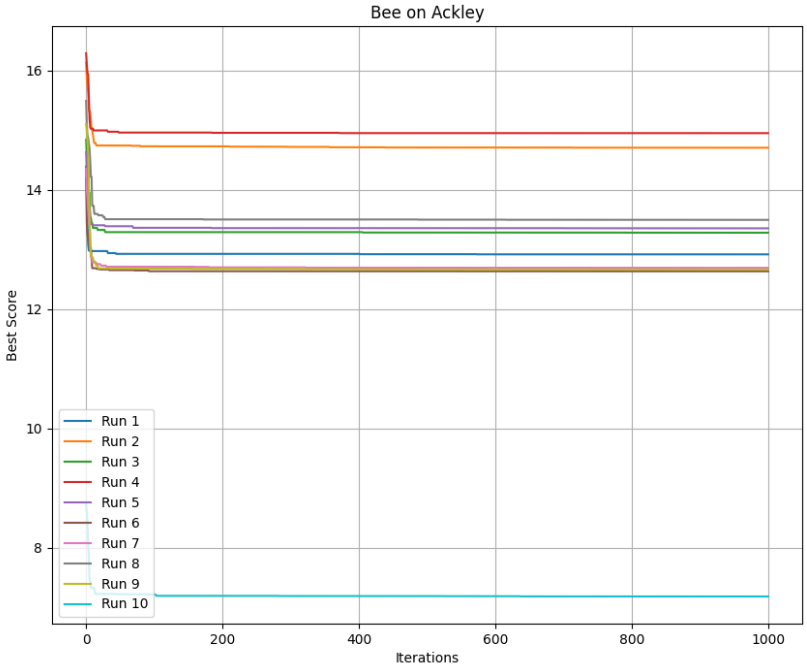

图 32: 蜜蜂优化算法 Ackley 函数迭代结果图

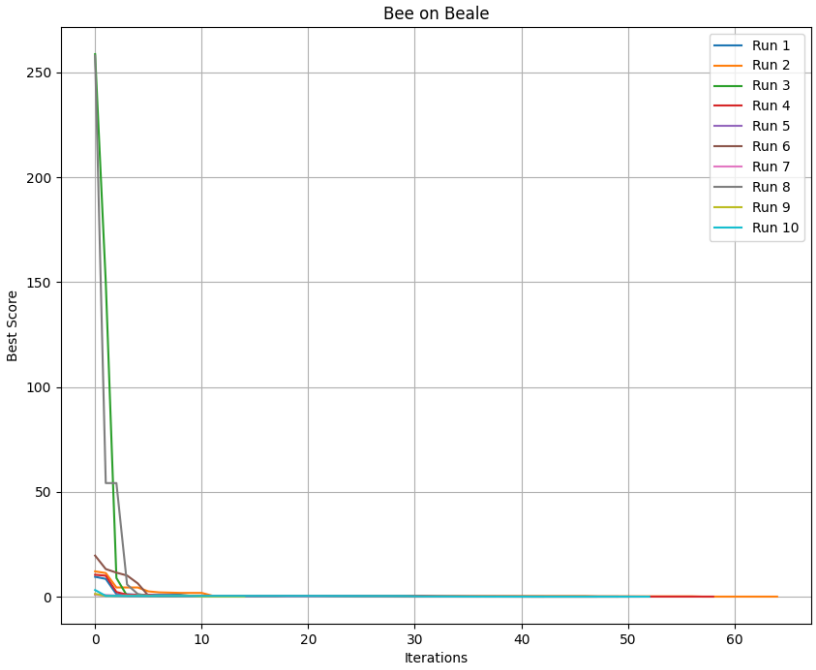

图 33: 蜜蜂优化算法 Beale 函数迭代结果图

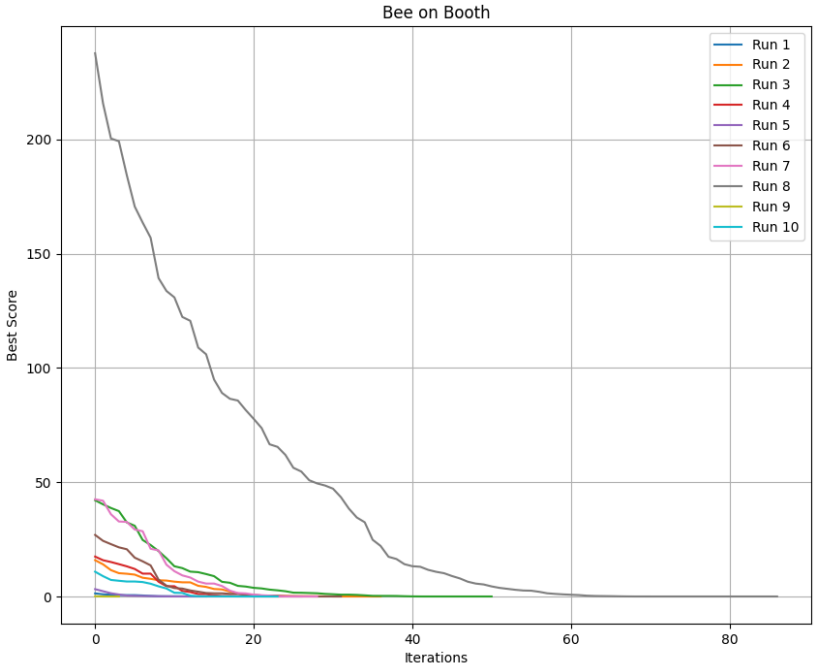

图 34: 蜜蜂优化算法 Booth 函数迭代结果图

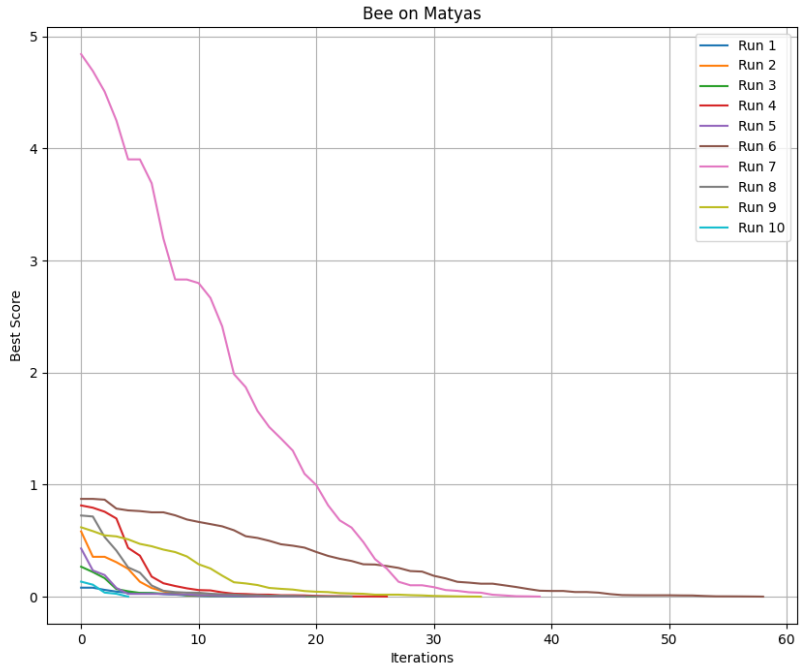

图 35: 蜜蜂优化算法 Matyas 函数迭代结果图

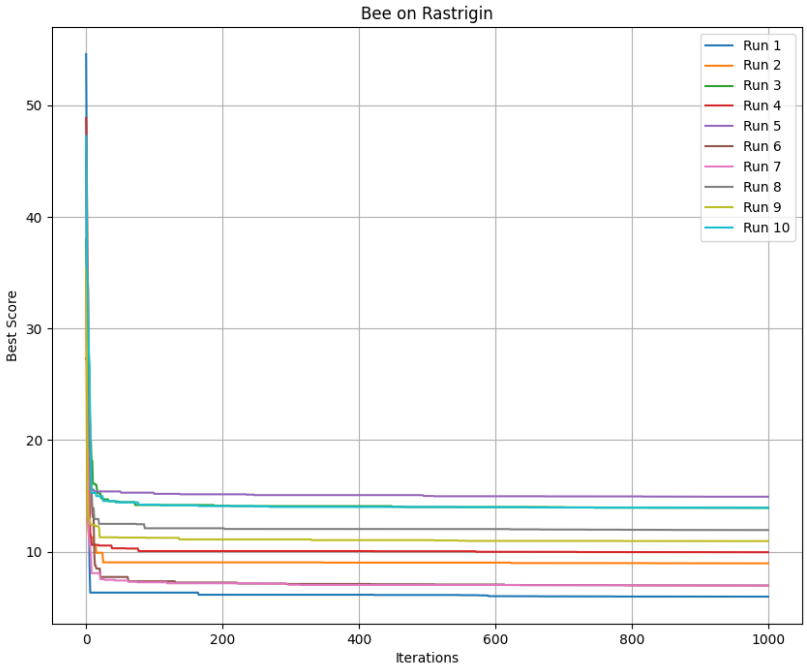

图 36: 蜜蜂优化算法 Rastrigin 函数迭代结果图

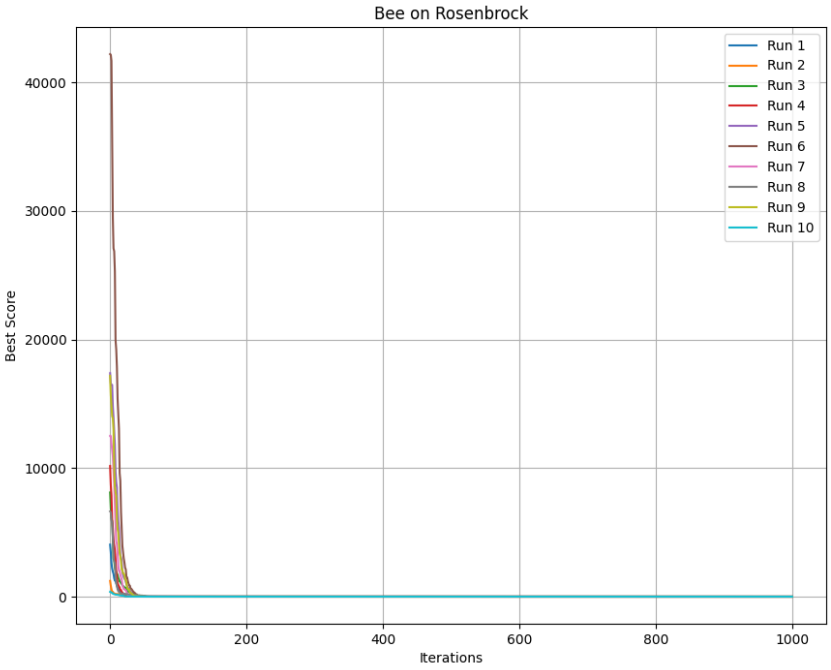

图 37: 蜜蜂优化算法 Rosenbrock 函数迭代结果图

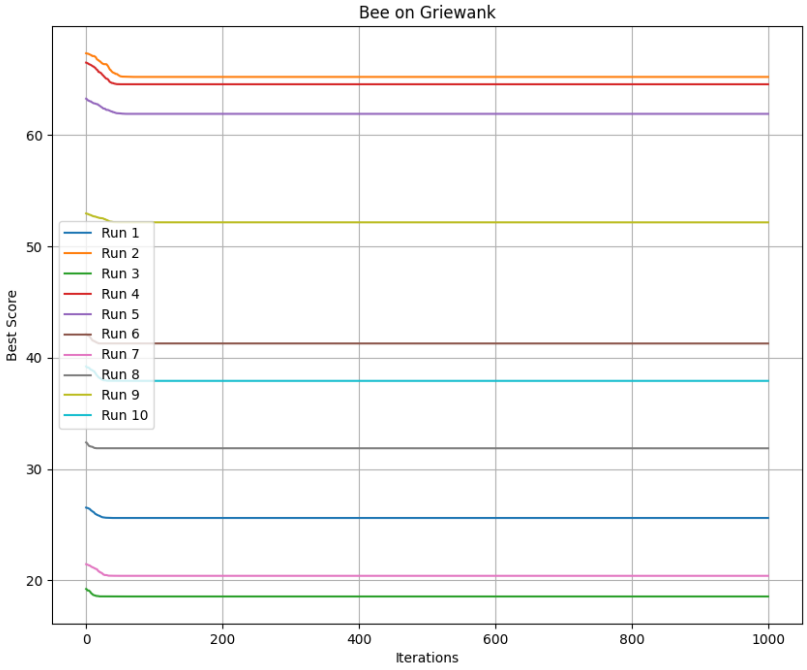

图 38: 蜜蜂优化算法 Griewank 函数迭代结果图

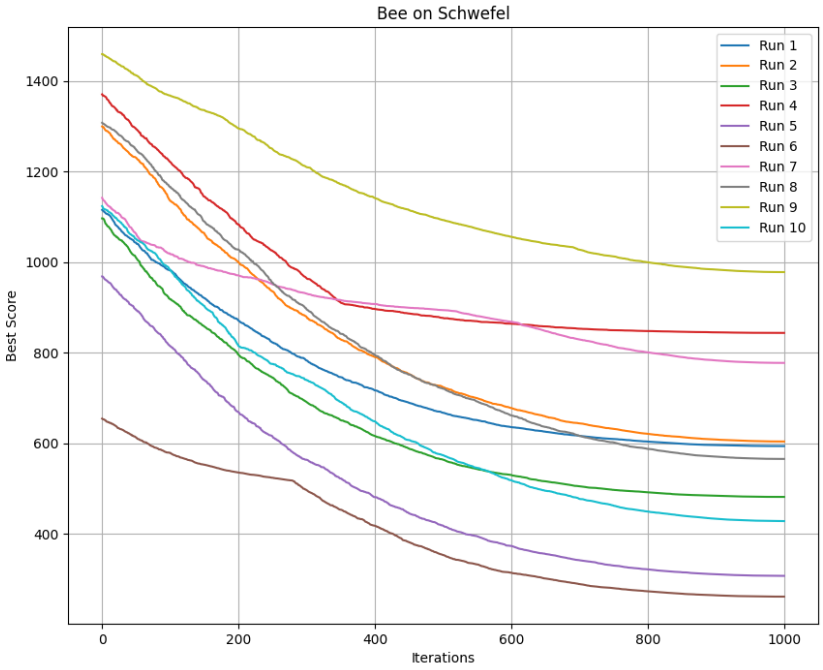

图 39: 蜜蜂优化算法 Schwefel 函数迭代结果图

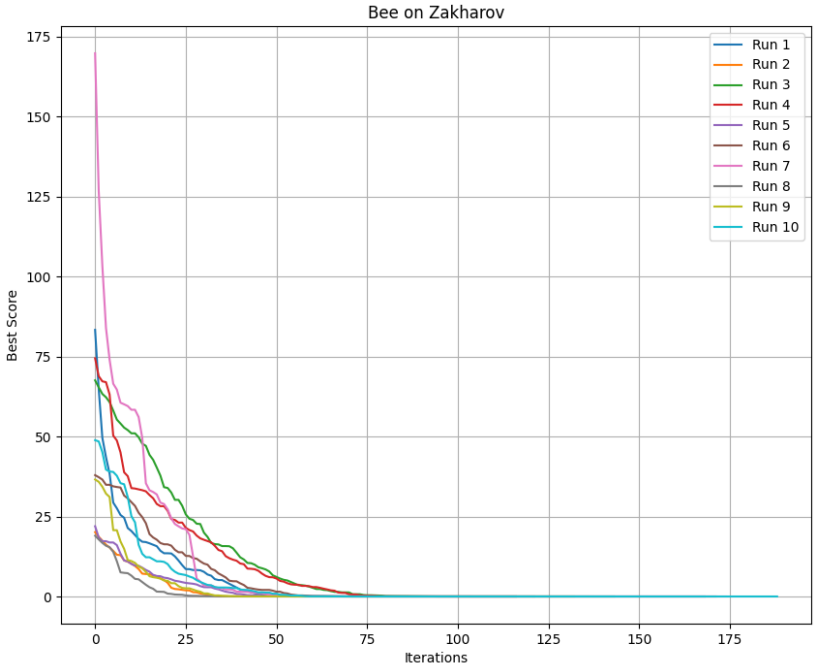

图 40: 蜜蜂优化算法 Zakharov 函数迭代结果图

4.5 鸽子优化算法

表 5: 鸽子优化算法在各函数上的迭代次数

Times	**Sphere**	**Ackley**	**Beale**	**Booth**	**Matyas**	**Rastrigin**	**Rosenbrock**	**Griewank**	**Schwefel**	**Zakharov**
Run 1	1000	1000	6	6	4	1000	1000	1000	1000	1000
Run 2	1000	1000	7	7	3	1000	1000	1000	1000	1000
Run 3	1000	1000	9	8	6	1000	1000	1000	1000	1000
Run 4	1000	1000	7	8	5	1000	1000	1000	1000	1000
Run 5	1000	1000	8	7	5	1000	1000	1000	1000	1000
Run 6	1000	1000	7	7	6	1000	1000	1000	1000	1000
Run 7	1000	1000	6	6	5	1000	1000	1000	1000	1000
Run 8	1000	1000	10	10	5	1000	1000	1000	1000	1000
Run 9	1000	1000	8	8	5	1000	1000	1000	1000	1000
Run 10	1000	1000	7	7	6	1000	1000	1000	1000	1000
Average	1000	1000	7.1	7.4	5.0	1000	1000	1000	1000	1000

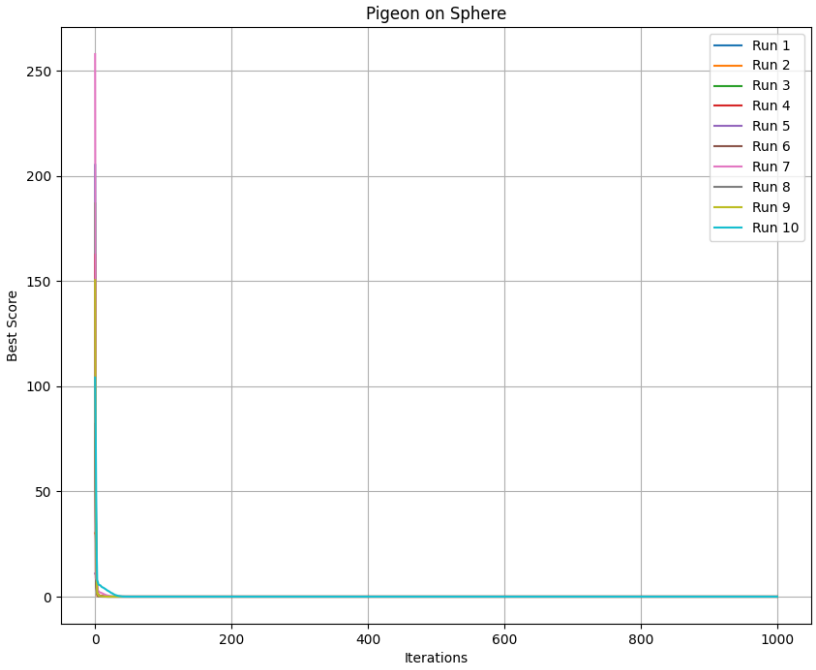

图 41: 鸽子优化算法 Sphere 函数迭代结果图

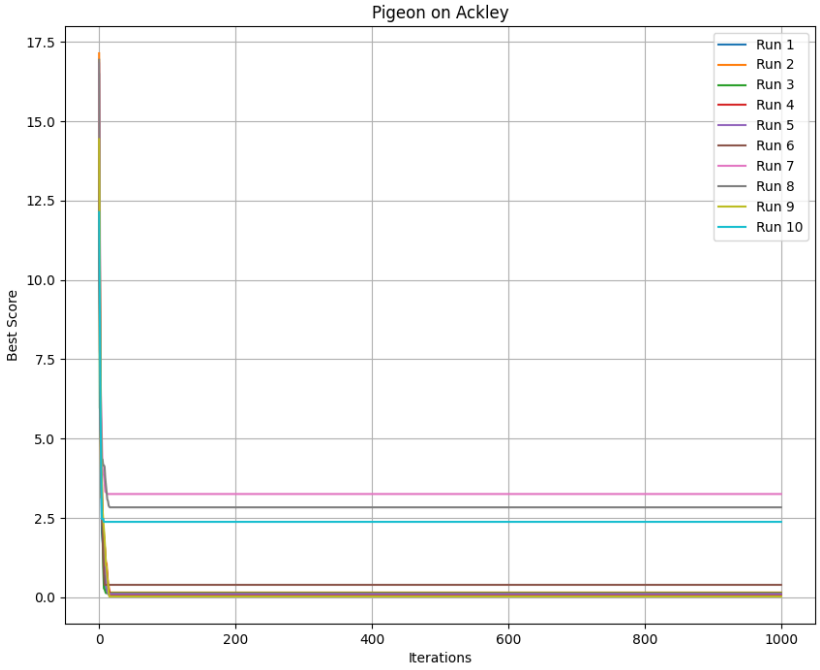

图 42: 鸽子优化算法 Ackley 函数迭代结果图

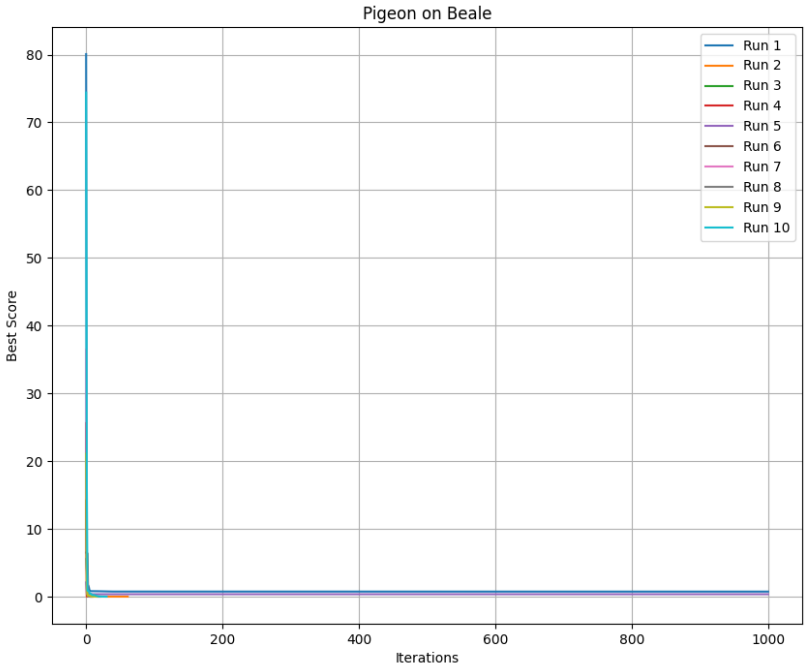

图 43: 鸽子优化算法 Beale 函数迭代结果图

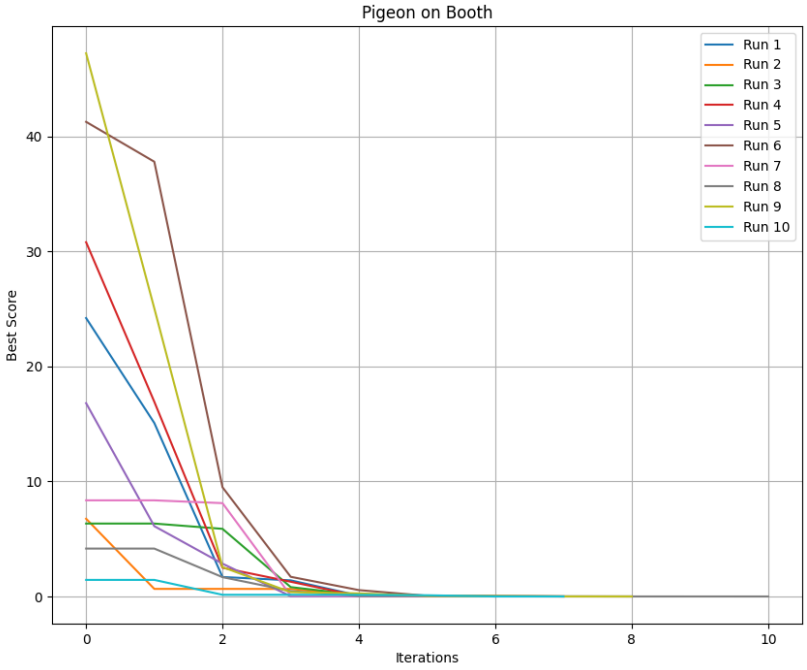

图 44: 鸽子优化算法 Booth 函数迭代结果图

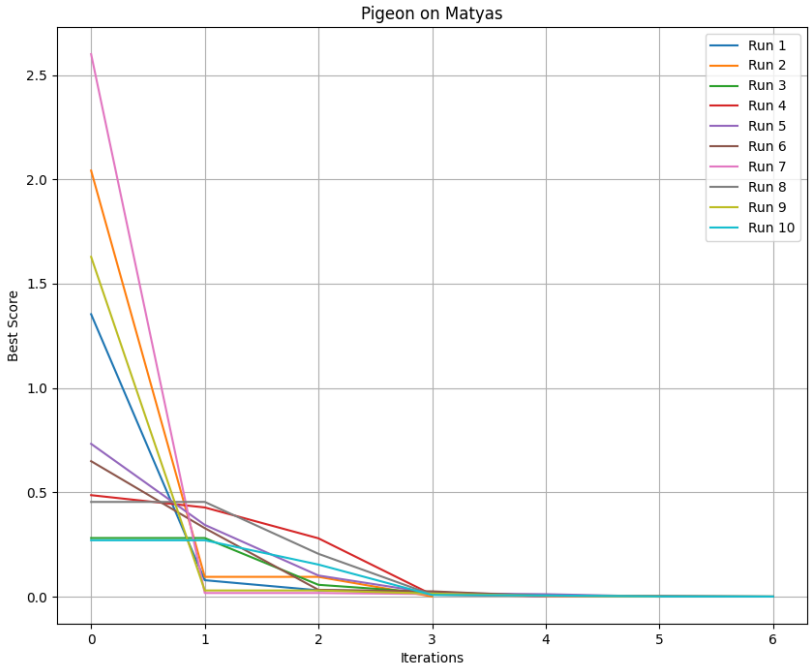

图 45: 鸽子优化算法 Matyas 函数迭代结果图

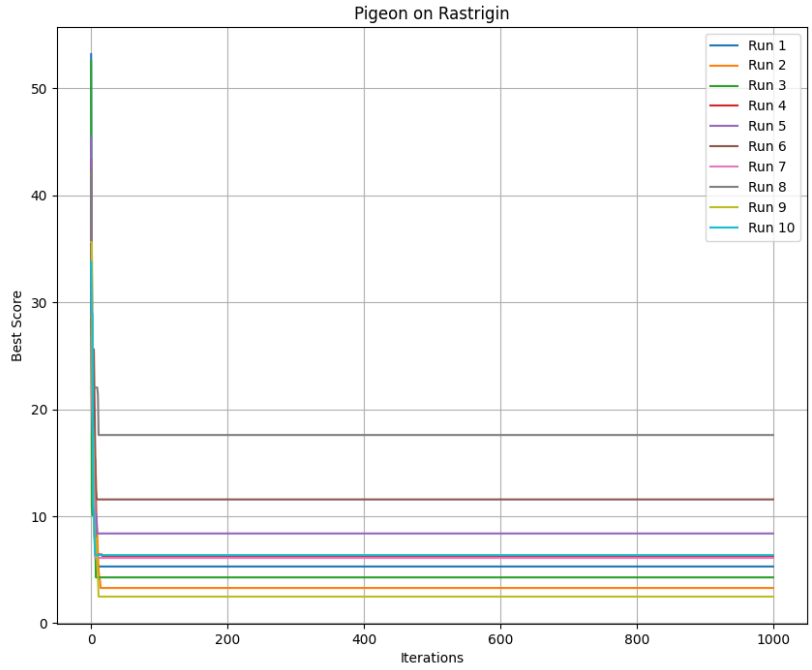

图 46: 鸽子优化算法 Rastrigin 函数迭代结果图

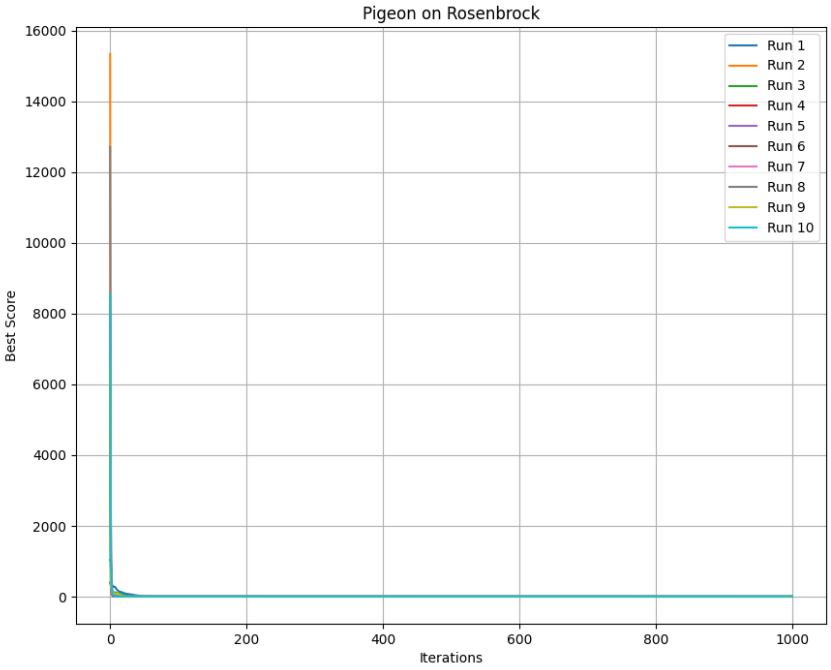

图 47: 鸽子优化算法 Rosenbrock 函数迭代结果图

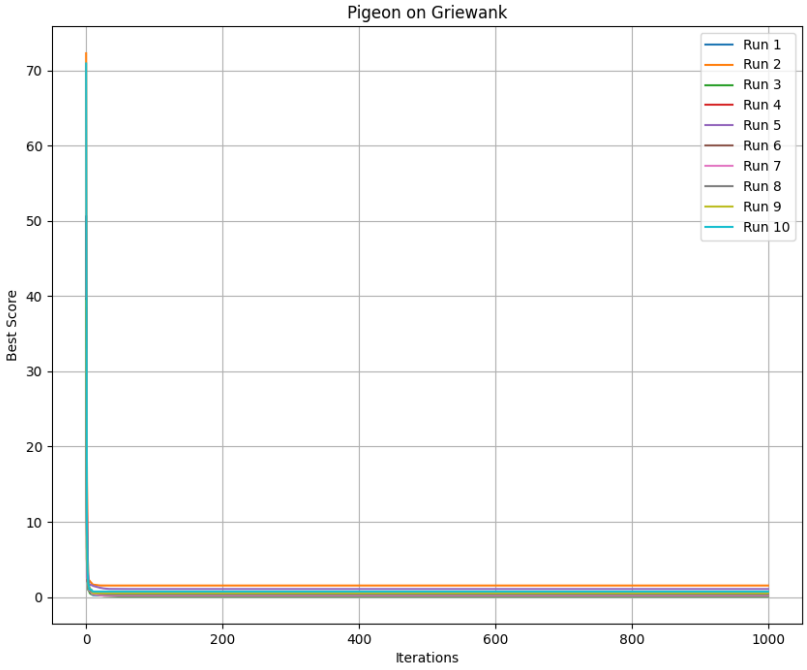

图 48: 鸽子优化算法 Griewank 函数迭代结果图

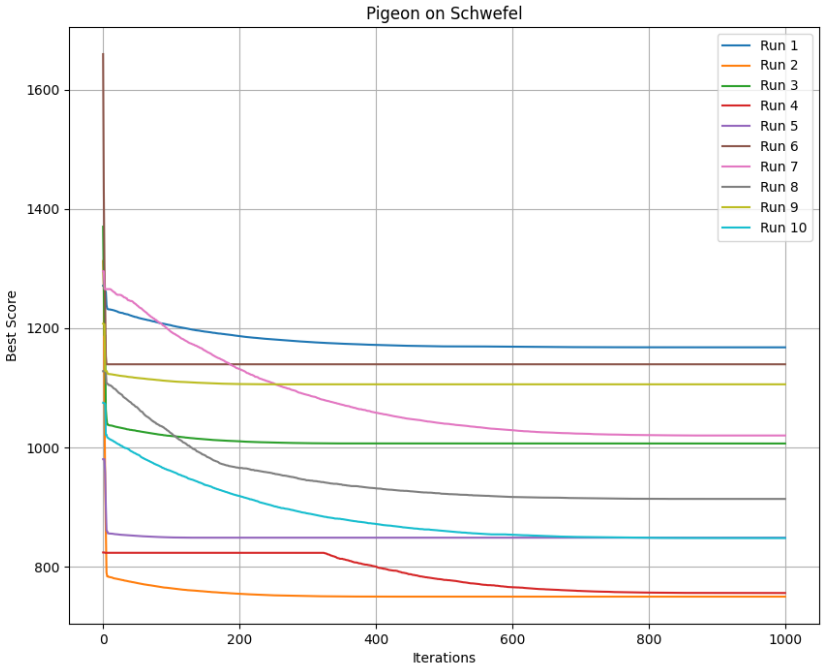

图 49: 鸽子优化算法 Schwefel 函数迭代结果图

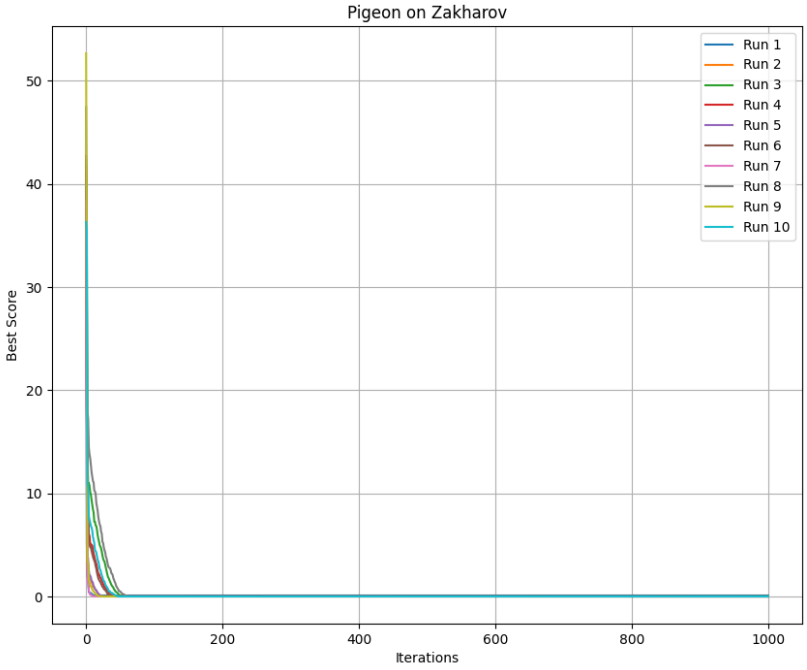

图 50: 鸽子优化算法 Zakharov 函数迭代结果图

5 实验总结

通过实验对五种进化算法（狼群优化算法、屎壳郎优化算法、粒子群优化算法、蜜蜂优化算法和鸽子优化算法）在十个经典优化问题上的性能进行了系统分析，结果显示各算法在适应不同问题的能力上表现差异明显。

狼群优化算法在 Sphere 函数上平均仅需约 11.8 次迭代达到收敛，这表明其在单峰函数的基本优化能力较强，但在复杂函数（如 Rastrigin 和 Rosenbrock）上普遍需要 1000 次迭代，反映出在跳出局部最优方面存在一定局限性。

屎壳郎优化算法表现出良好的动态调整能力，在简单问题如 Matyas 函数上平均仅需 58.8 次迭代，但在复杂函数（如 Schwefel 和 Griewank）中始终未能收敛，表明其全局搜索能力有待提高。相比之下，其在 Beale 函数上的收敛次数（589.9 次）低于其他多峰问题，表明其在中等复杂度问题中具备一定优势。

粒子群优化算法在 Sphere 和 Ackley 函数上均能在 50 次迭代以内找到接近最优解（分别为 25.7 和 44 次迭代），其在低维问题上的稳定表现得益于粒子间信息共享机制。但其在高维多模态问题上（如 Rastrigin 和 Schwefel）需要耗费近 1000 次迭代，显示了局部开发能力的不足。

蜜蜂优化算法的实验数据表明其在小规模、多模态问题（如 Beale 函数，46.7 次迭代）上具备较强的全局搜索能力，但在高维复杂问题（如 Rosenbrock 和 Schwefel）中未能有效收敛，所有实验均达到最大迭代次数（1000 次）。这可能与蜜蜂算法对局部开发的重视不足有关。

鸽子优化算法在简单函数如 Beale 和 Booth 上表现优异，仅需 7 次左右的平均迭代即可达到最优解，但在大多数复杂函数（如 Sphere 和 Ackley）上均需要 1000 次迭代。其全局导航机制使得初期搜索效率较高，但缺乏后期的精细调整能力。

因此，不同算法的性能受其设计特点影响显著。狼群算法在平衡探索与开发方面表现均衡，适用于多模态问题；屎壳郎算法和粒子群算法在局部优化和动态环境中具有优势；蜜蜂算法在全局搜索方面具备潜力，而鸽子算法则更适合简单目标优化任务。

6 参考文献

[1] WU H S, ZHANG F M. Wolf pack algorithm for unconstrained global optimization[J]. Mathematical Problems in Engineering, 2014, 2014(1): 465082.

[2] XUE J, SHEN B. Dung beetle optimizer: A new meta-heuristic algorithm for global optimization [J]. The Journal of Supercomputing, 2023, 79(7): 7305-7336.

[3] KENNEDY J, EBERHART R. Particle swarm optimization[C]//Proceedings of ICNN'95-international conference on neural networks: vol. 4. 1995: 1942-1948.

[4] YUCE B, PACKIANATHER M S, MASTROCINQUE E, et al. Honey bees inspired optimization method: the bees algorithm[J]. Insects, 2013, 4(4): 646-662.

[5] DUAN H, QIAO P. Pigeon-inspired optimization: a new swarm intelligence optimizer for air robot path planning[J]. International journal of intelligent computing and cybernetics, 2014, 7(1): 24-37.