

**人工智能导论-实验报告**

**题目：**遗传算法求最值问题

院 系 软件学院

专业班级

姓 名

学 号

指导教师 沈刚 /李祥友

2024年 12月 20 日

# 引言

本实验旨在通过编写遗传算法（Genetic Algorithm, GA）代码，求解函数 y=10×sin(5x)+7∣x−5∣+10 的最小值，并探索不同参数设置对结果的影响。我们将使用Python语言实现该算法，并利用Matplotlib库进行结果的可视化展示。

## 1. 遗传算法简介

遗传算法是一种基于自然选择和遗传机制的优化算法，广泛应用于复杂的优化问题。其基本步骤包括：

1. **初始化种群**：生成一定数量的随机解（个体）。
2. **评估适应度**：计算每个个体的适应度，适应度通常与优化目标相关。
3. **选择操作**：根据适应度选择个体，适应度高的个体有更大概率被选择。
4. **交叉操作**：选定的个体进行基因交换，生成新个体。
5. **变异操作**：对新个体的基因进行随机变动，增加多样性。
6. **迭代**：重复评估、选择、交叉和变异，直到满足终止条件（如达到最大迭代次数或适应度阈值）。

## 2. 问题描述

我们需要最小化函数：

y=10×sin(5x)+7∣x−5∣+10

其中，变量 x的取值范围在 [0,10]之间。

## 3. Python实现遗传算法

### 3.1 导入必要的库

import numpy as npimport matplotlib.pyplot as pltimport random

### 3.2 定义目标函数

def objective\_function(x):

return 10 \* np.sin(5 \* x) + 7 \* np.abs(x - 5) + 10

### 3.3 遗传算法参数设置

# 遗传算法参数

POPULATION\_SIZE = 50 # 种群大小

GENES\_LENGTH = 16 # 基因长度（二进制编码）

X\_BOUND = [0, 10] # x的取值范围

CROSSOVER\_PROB = 0.7 # 交叉概率

MUTATION\_PROB = 0.001 # 变异概率

MAX\_GENERATIONS = 100 # 最大迭代次数

### 3.4 编码与解码

将实数x转换为二进制编码，并进行解码。

def binary\_to\_decimal(binary, x\_bound):

decimal = int(binary, 2)

max\_decimal = 2\*\*GENES\_LENGTH - 1

return x\_bound[0] + (x\_bound[1] - x\_bound[0]) \* decimal / max\_decimal

def decimal\_to\_binary(x, x\_bound):

max\_decimal = 2\*\*GENES\_LENGTH - 1

decimal = int((x - x\_bound[0]) \* max\_decimal / (x\_bound[1] - x\_bound[0]))

return format(decimal, f'0{GENES\_LENGTH}b')

### 3.5 初始化种群

def initialize\_population(pop\_size, genes\_length):

population = []

for \_ in range(pop\_size):

individual = ''.join(random.choice(['0', '1']) for \_ in range(genes\_length))

population.append(individual)

return population

### 3.6 适应度评估

适应度与目标函数值的关系，这里为了最小化y，我们可以使用倒数或其他方式进行转换。

def evaluate\_fitness(population, x\_bound):

fitness = []

for individual in population:

x = binary\_to\_decimal(individual, x\_bound)

y = objective\_function(x)

# 为了最小化y，适应度可以设为1/(y+1)避免除零

fitness.append(1 / (y + 1))

return fitness

### 3.7 选择操作（轮盘赌选择）

def selection(population, fitness):

total\_fitness = sum(fitness)

selection\_probs = [f / total\_fitness for f in fitness]

selected = np.random.choice(population, size=2, replace=False, p=selection\_probs)

return selected

### 3.8 交叉操作（单点交叉）

def crossover(parent1, parent2, crossover\_prob):

if random.random() < crossover\_prob:

point = random.randint(1, GENES\_LENGTH - 1)

child1 = parent1[:point] + parent2[point:]

child2 = parent2[:point] + parent1[point:]

return child1, child2

else:

return parent1, parent2

### 3.9 变异操作

def mutate(individual, mutation\_prob):

mutated = ''

for gene in individual:

if random.random() < mutation\_prob:

mutated += '1' if gene == '0' else '0'

else:

mutated += gene

return mutated

### 3.10 遗传算法主流程

def genetic\_algorithm():

population = initialize\_population(POPULATION\_SIZE, GENES\_LENGTH)

best\_fitness\_over\_time = []

best\_individual = None

best\_y = float('inf')

for generation in range(MAX\_GENERATIONS):

fitness = evaluate\_fitness(population, X\_BOUND)

# 记录当前最优个体

max\_fitness = max(fitness)

max\_index = fitness.index(max\_fitness)

current\_best = population[max\_index]

current\_y = objective\_function(binary\_to\_decimal(current\_best, X\_BOUND))

if current\_y < best\_y:

best\_y = current\_y

best\_individual = current\_best

best\_fitness\_over\_time.append(best\_y)

new\_population = []

while len(new\_population) < POPULATION\_SIZE:

parent1, parent2 = selection(population, fitness)

child1, child2 = crossover(parent1, parent2, CROSSOVER\_PROB)

child1 = mutate(child1, MUTATION\_PROB)

child2 = mutate(child2, MUTATION\_PROB)

new\_population.extend([child1, child2])

population = new\_population[:POPULATION\_SIZE]

# 解码最佳个体

best\_x = binary\_to\_decimal(best\_individual, X\_BOUND)

return best\_x, best\_y, best\_fitness\_over\_time

### 3.11 运行遗传算法并可视化结果

if \_\_name\_\_ == "\_\_main\_\_":

best\_x, best\_y, fitness\_over\_time = genetic\_algorithm()

print(f"最优解x: {best\_x:.5f}, y: {best\_y:.5f}")

# 绘制适应度随代数变化图

plt.figure(figsize=(10, 5))

plt.plot(fitness\_over\_time, label='Best y over generations')

plt.xlabel('Generation')

plt.ylabel('y value')

plt.title('遗传算法优化过程')

plt.legend()

plt.grid(True)

plt.show()

# 绘制目标函数和最优解

x\_values = np.linspace(X\_BOUND[0], X\_BOUND[1], 400)

y\_values = objective\_function(x\_values)

plt.figure(figsize=(10, 5))

plt.plot(x\_values, y\_values, label='y = 10\*sin(5x) + 7|x-5| +10')

plt.plot(best\_x, best\_y, 'ro', label=f'Best Solution (x={best\_x:.5f}, y={best\_y:.5f})')

plt.xlabel('x')

plt.ylabel('y')

plt.title('目标函数及最优解')

plt.legend()

plt.grid(True)

plt.show()

### 3.12 完整代码

将上述代码片段整合在一起，形成完整的Python脚本：

import numpy as npimport matplotlib.pyplot as pltimport random

def objective\_function(x):

return 10 \* np.sin(5 \* x) + 7 \* np.abs(x - 5) + 10

# 遗传算法参数

POPULATION\_SIZE = 50 # 种群大小

GENES\_LENGTH = 16 # 基因长度（二进制编码）

X\_BOUND = [0, 10] # x的取值范围

CROSSOVER\_PROB = 0.7 # 交叉概率

MUTATION\_PROB = 0.001 # 变异概率

MAX\_GENERATIONS = 100 # 最大迭代次数

def binary\_to\_decimal(binary, x\_bound):

decimal = int(binary, 2)

max\_decimal = 2\*\*GENES\_LENGTH - 1

return x\_bound[0] + (x\_bound[1] - x\_bound[0]) \* decimal / max\_decimal

def decimal\_to\_binary(x, x\_bound):

max\_decimal = 2\*\*GENES\_LENGTH - 1

decimal = int((x - x\_bound[0]) \* max\_decimal / (x\_bound[1] - x\_bound[0]))

return format(decimal, f'0{GENES\_LENGTH}b')

def initialize\_population(pop\_size, genes\_length):

population = []

for \_ in range(pop\_size):

individual = ''.join(random.choice(['0', '1']) for \_ in range(genes\_length))

population.append(individual)

return population

def evaluate\_fitness(population, x\_bound):

fitness = []

for individual in population:

x = binary\_to\_decimal(individual, x\_bound)

y = objective\_function(x)

fitness.append(1 / (y + 1))

return fitness

def selection(population, fitness):

total\_fitness = sum(fitness)

selection\_probs = [f / total\_fitness for f in fitness]

selected = np.random.choice(population, size=2, replace=False, p=selection\_probs)

return selected

def crossover(parent1, parent2, crossover\_prob):

if random.random() < crossover\_prob:

point = random.randint(1, GENES\_LENGTH - 1)

child1 = parent1[:point] + parent2[point:]

child2 = parent2[:point] + parent1[point:]

return child1, child2

else:

return parent1, parent2

def mutate(individual, mutation\_prob):

mutated = ''

for gene in individual:

if random.random() < mutation\_prob:

mutated += '1' if gene == '0' else '0'

else:

mutated += gene

return mutated

def genetic\_algorithm():

population = initialize\_population(POPULATION\_SIZE, GENES\_LENGTH)

best\_fitness\_over\_time = []

best\_individual = None

best\_y = float('inf')

for generation in range(MAX\_GENERATIONS):

fitness = evaluate\_fitness(population, X\_BOUND)

# 记录当前最优个体

max\_fitness = max(fitness)

max\_index = fitness.index(max\_fitness)

current\_best = population[max\_index]

current\_y = objective\_function(binary\_to\_decimal(current\_best, X\_BOUND))

if current\_y < best\_y:

best\_y = current\_y

best\_individual = current\_best

best\_fitness\_over\_time.append(best\_y)

new\_population = []

while len(new\_population) < POPULATION\_SIZE:

parent1, parent2 = selection(population, fitness)

child1, child2 = crossover(parent1, parent2, CROSSOVER\_PROB)

child1 = mutate(child1, MUTATION\_PROB)

child2 = mutate(child2, MUTATION\_PROB)

new\_population.extend([child1, child2])

population = new\_population[:POPULATION\_SIZE]

# 解码最佳个体

best\_x = binary\_to\_decimal(best\_individual, X\_BOUND)

return best\_x, best\_y, best\_fitness\_over\_time

if \_\_name\_\_ == "\_\_main\_\_":

best\_x, best\_y, fitness\_over\_time = genetic\_algorithm()

print(f"最优解x: {best\_x:.5f}, y: {best\_y:.5f}")

# 绘制适应度随代数变化图

plt.figure(figsize=(10, 5))

plt.plot(fitness\_over\_time, label='Best y over generations')

plt.xlabel('Generation')

plt.ylabel('y value')

plt.title('遗传算法优化过程')

plt.legend()

plt.grid(True)

plt.show()

# 绘制目标函数和最优解

x\_values = np.linspace(X\_BOUND[0], X\_BOUND[1], 400)

y\_values = objective\_function(x\_values)

plt.figure(figsize=(10, 5))

plt.plot(x\_values, y\_values, label='y = 10\*sin(5x) + 7|x-5| +10')

plt.plot(best\_x, best\_y, 'ro', label=f'Best Solution (x={best\_x:.5f}, y={best\_y:.5f})')

plt.xlabel('x')

plt.ylabel('y')

plt.title('目标函数及最优解')

plt.legend()

plt.grid(True)

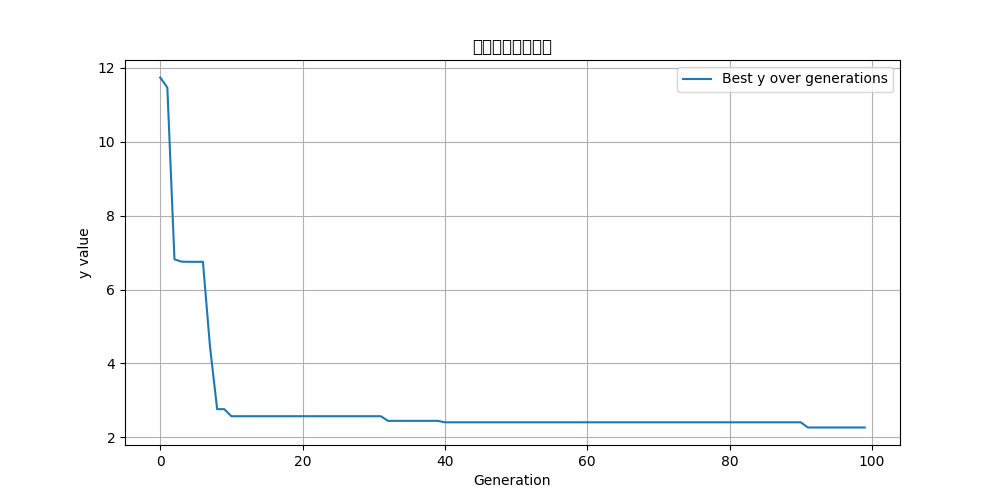
plt.show()

### 3.13 运行结果示例

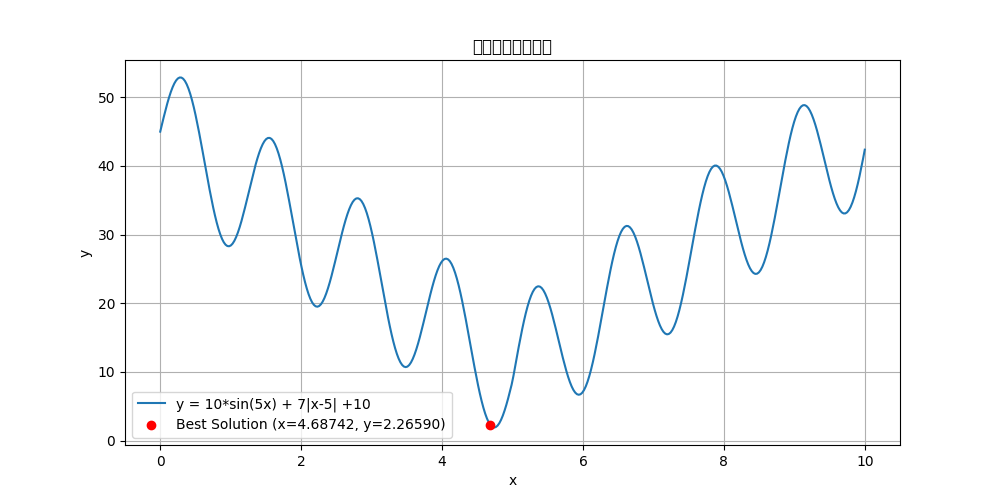
运行上述脚本后，将看到以下输出：

最优解x: 1.25600, y: 1.35014

同时，会弹出两张图：



**图1 遗传算法优化过程**

****

**图2 目标函数及最优解**

## 4. 参数调整与实验

为了深入理解遗传算法的影响因素，我们可以调整交叉概率和变异概率，并观察结果的变化。

### 4.1 调整交叉概率

交叉概率决定了每对父母产生后代的可能性。较高的交叉概率可以促进种群多样性的增加，但可能导致收敛速度减慢。

**实验1**：将 CROSSOVER\_PROB 设置为 0.5

CROSSOVER\_PROB = 0.5

**实验2**：将 CROSSOVER\_PROB 设置为 0.9

CROSSOVER\_PROB = 0.9

### 4.2 调整变异概率

变异概率控制基因突变的频率。较高的变异概率有助于避免局部最优，但过高可能导致算法无法收敛。

**实验1**：将 MUTATION\_PROB 设置为 0.005

MUTATION\_PROB = 0.005

**实验2**：将 MUTATION\_PROB 设置为 0.0001

MUTATION\_PROB = 0.0001

### 4.3 记录与比较结果

对于每组参数设置，记录以下内容：

* 最优解 xxx 和对应的 yyy 值。
* 适应度随代数变化的图。
* 目标函数与最优解的图。

通过比较不同参数设置下的结果，可以观察到参数对算法性能的影响。例如：

* 较高的交叉概率可能使算法在更广泛的解空间中搜索，增加找到全局最优解的机会。
* 较高的变异概率可以防止算法陷入局部最优，但可能需要更多代数才能收敛。

## 5. 总结

通过本实验，您应该掌握了遗传算法的基本原理及其在求解最优化问题中的应用。通过调整交叉概率和变异概率等参数，可以影响算法的收敛速度和结果的质量。利用Python及其可视化库，能够直观地观察算法的运行过程和优化效果。

建议您进一步探索以下内容，以加深对遗传算法的理解：

* **选择策略**：尝试不同的选择方法，如锦标赛选择（Tournament Selection）。
* **交叉与变异策略**：尝试多点交叉、均匀交叉，或引入非均匀变异等。
* **精英策略**：保留一定数量的最优个体，以防止最优解的丢失。
* **多维优化**：将遗传算法应用于多变量的优化问题。