

**人工智能导论-实验报告**

**题目：**遗传算法求解TSP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

**目录**

**1 引言**

1.1遗传算法求解TSP问题背景和研究意义

1.1.1问题背景

1.1.2研究意义

1.2实验目的和实验内容

1.2.1实验目的

1.2.2实验内容

**2问题的表示和求解算法**

2.1 遗传算法求解TSP问题求解算法

**3实验设计**

3.1 实验目标和评价指标

3.2 遗传算法的详细求解

**4实验结果**

4.1 使用的不同方法及其特点

4.2 实验结果的展示

**5实验分析与讨论**

5.1 性能比较

5.2 定性和定量分析

5.3 结果可视化

**6结论**

6.1 实验结果总结

6.2 改进思考与建议

**7参考文献**

**8附录**

8.1 实验代码

8.2 实验数据表和图表

## 1 引言

### 1.1 遗传算法求解TSP问题背景和研究意义

#### 1.1.1 问题背景

旅行商问题（Traveling Salesman Problem, TSP）是组合优化中的经典问题之一。其目标是在给定一组城市及其之间的距离的情况下，寻找一条经过每个城市且仅经过一次的最短路径。TSP不仅在理论研究中占有重要地位，还在实际应用中具有广泛的影响，如物流配送、芯片制造、电路板设计等领域。由于TSP属于NP难问题，随着城市数量的增加，求解复杂度呈指数级增长，传统的精确算法在处理大规模TSP时往往面临计算时间过长的问题。因此，开发高效的近似算法成为研究热点。

#### 1.1.2 研究意义

遗传算法（Genetic Algorithm, GA）是一种基于自然选择和遗传机制的启发式搜索算法，广泛应用于复杂的优化问题。将遗传算法应用于TSP的求解，不仅可以有效地寻找近似最优解，还能在较短的时间内处理较大规模的城市集合。通过研究和优化遗传算法在TSP中的应用，可以为实际生产和物流优化提供有力的算法支持，同时推动遗传算法理论的发展和改进。

### 1.2 实验目的和实验内容

#### 1.2.1 实验目的

本实验旨在：

1. 理解并掌握遗传算法的基本原理和操作，包括种群初始化、适应度评估、选择、交叉和变异等。
2. 应用遗传算法求解包含20个城市的TSP问题，寻找经过每座城市且路径总长度尽可能短的旅行路径。
3. 通过调整遗传算法的参数，分析其对算法性能的影响。
4. 利用Python编程实现遗传算法，并将实验结果进行可视化展示，增强对算法运行过程和结果的理解。

#### 1.2.2 实验内容

1. 编写并运行遗传算法代码，生成随机城市坐标，求解TSP问题。
2. 调整遗传算法中的关键参数，记录并分析不同参数设置下的实验结果。
3. 使用Matplotlib库对算法优化过程中的路径长度变化、适应度变化、种群多样性等进行可视化展示。
4. 撰写实验报告，分析实验结果，讨论算法性能，并提出改进建议。

## 2 问题的表示和求解算法

### 2.1 遗传算法求解TSP问题求解算法

遗传算法是一种模拟自然进化过程的优化算法，其核心思想包括选择、交叉和变异。应用于TSP问题时，遗传算法的主要步骤如下：

1. **种群表示**：每个个体（染色体）表示为一个城市索引的排列，代表一个可能的旅行路径。
2. **初始化种群**：随机生成一定数量的个体，形成初始种群。
3. **适应度评估**：计算每个个体的适应度，通常定义为路径总长度的倒数，即路径越短，适应度越高。
4. **选择操作**：采用锦标赛选择等方法，根据适应度选择适应性较高的个体作为父代。
5. **交叉操作**：使用顺序交叉（Ordered Crossover, OX）等方法，交换父代的基因，生成新个体（子代）。
6. **变异操作**：通过交换两个城市的位置等方法，对新个体进行变异，增加种群多样性。
7. **精英保留**：保留适应度最高的若干个体，确保最优解不被遗失。
8. **迭代**：重复适应度评估、选择、交叉和变异过程，直至满足终止条件（如达到最大迭代次数）。

通过上述步骤，遗传算法能够逐步优化种群中的个体，逼近TSP问题的最优解。

## 3 实验设计

### 3.1 实验目标和评价指标

#### 实验目标

1. 实现基于遗传算法的TSP求解程序，能够有效找到经过20个随机生成城市的近似最优路径。
2. 通过实验分析，探讨遗传算法中不同参数设置对算法性能（如收敛速度、最终路径长度）的影响。
3. 生成并分析可视化图表，直观展示遗传算法的优化过程和最终结果。

#### 评价指标

1. **路径总长度**：评估旅行路径的总距离，路径越短表示算法效果越好。
2. **收敛速度**：评估算法达到近似最优解所需的迭代次数，迭代次数越少表示收敛速度越快。
3. **算法稳定性**：多次运行算法，评估结果的一致性和可靠性。
4. **可视化展示**：通过图表展示路径长度随迭代次数的变化趋势及最终最优路径的空间分布。

### 3.2 遗传算法的详细求解

在本实验中，遗传算法的具体实现包括以下步骤：

1. **城市生成**：使用numpy库生成20个城市的随机坐标，坐标范围在[0, 100)之间。
2. **距离矩阵计算**：预计算城市间的欧几里得距离，存储在距离矩阵中，以提高适应度评估的效率。
3. **种群初始化**：生成初始种群，每个个体为城市索引的随机排列。
4. **适应度评估**：计算每个个体的路径总长度，并定义适应度为路径总长度的倒数。
5. **选择操作**：采用锦标赛选择，从种群中选择适应度较高的个体作为父代。
6. **交叉操作**：使用顺序交叉（OX）方法，生成子代，确保子代为合法的城市排列。
7. **变异操作**：通过交换两个城市的位置进行变异，增加种群多样性。
8. **精英保留**：保留适应度最高的若干个体，确保最优解不被丢失。
9. **迭代**：重复适应度评估、选择、交叉和变异过程，直至达到预设的最大迭代次数。

## 4 实验结果

### 4.1 使用的不同方法及其特点

在本实验中，主要采用了以下方法：

1. **顺序交叉（Ordered Crossover, OX）**：通过复制父代1的部分基因序列到子代，并按照父代2的顺序填补剩余基因，确保生成的子代为合法的城市排列。
2. **交换变异**：随机交换个体中的两个城市的位置，增加种群的多样性，防止算法陷入局部最优。
3. **锦标赛选择**：从随机选择的一组个体中选择适应度最高的个体作为父代，增加优秀个体被选中的概率。
4. **精英保留**：保留适应度最高的若干个体，确保最优解不被遗失。

这些方法结合使用，有效地提升了遗传算法在TSP问题中的求解效果。

### 4.2 实验结果的展示

运行遗传算法后，获得了以下结果：

#### 4.2.1 路径长度变化图

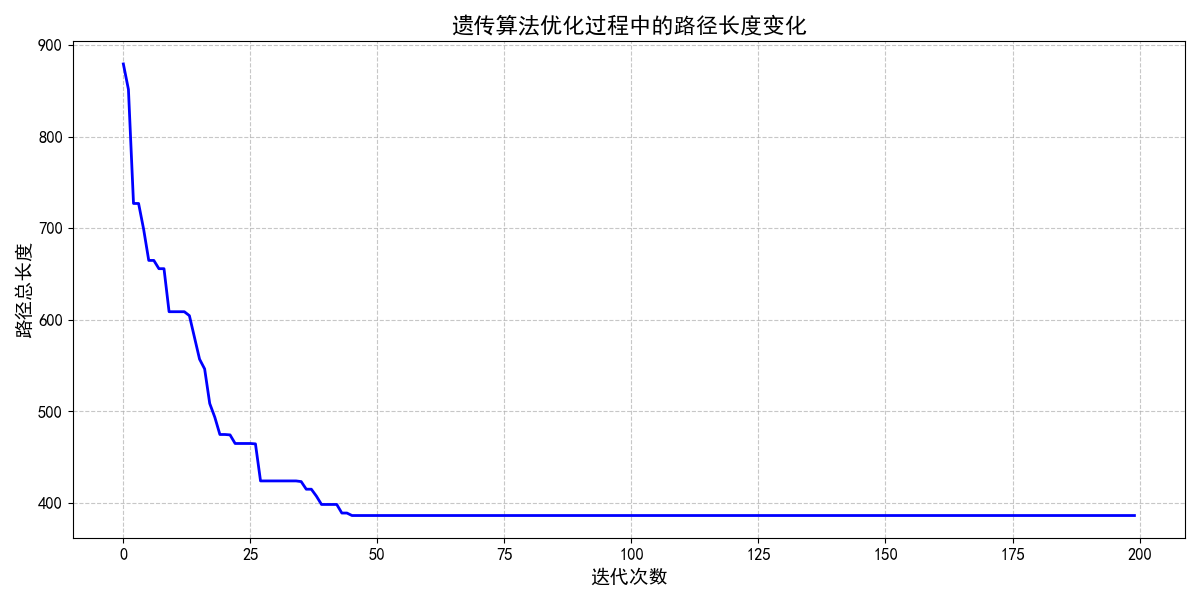


图4.1 遗传算法优化过程中的路径长度变化

该图展示了遗传算法在200代迭代过程中，路径总长度的逐步减少趋势。路径长度由初始的较高值逐渐下降，表明算法在不断优化个体，逼近最优解。

#### 4.2.2 最优路径图

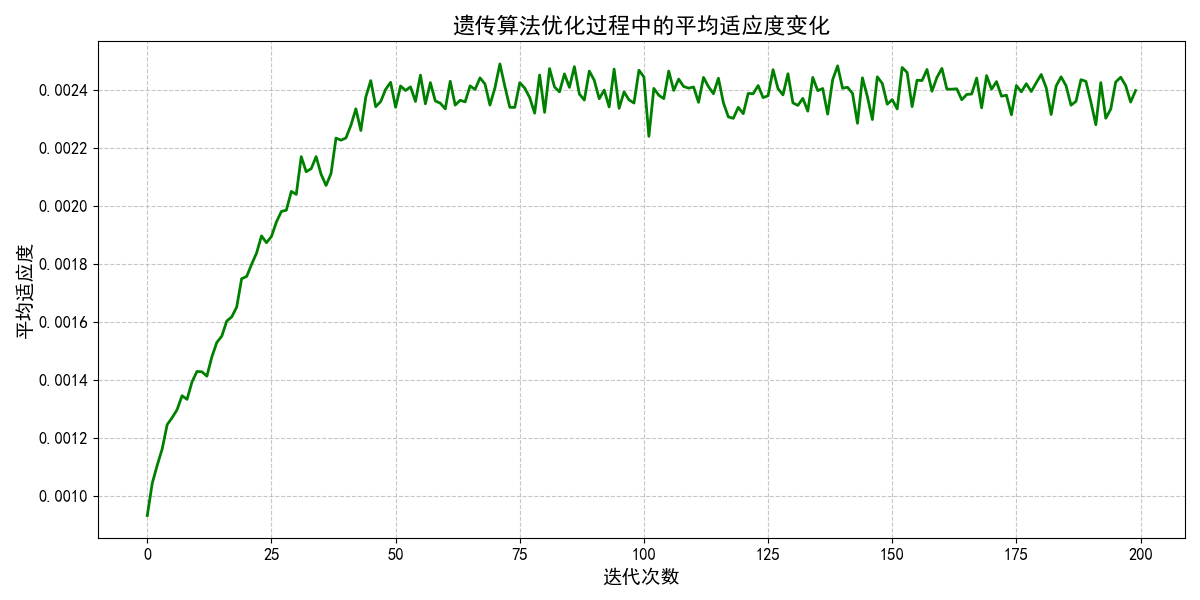


图4.2 遗传算法求解的最优路径

该图展示了20个城市的位置及遗传算法找到的最优路径。红色圆点表示城市的位置，数字标注表示城市的索引，蓝色线条表示旅行路径。横纵坐标轴清晰显示，便于观察城市的具体位置和路径的连贯性。

#### 4.2.3 平均适应度变化图

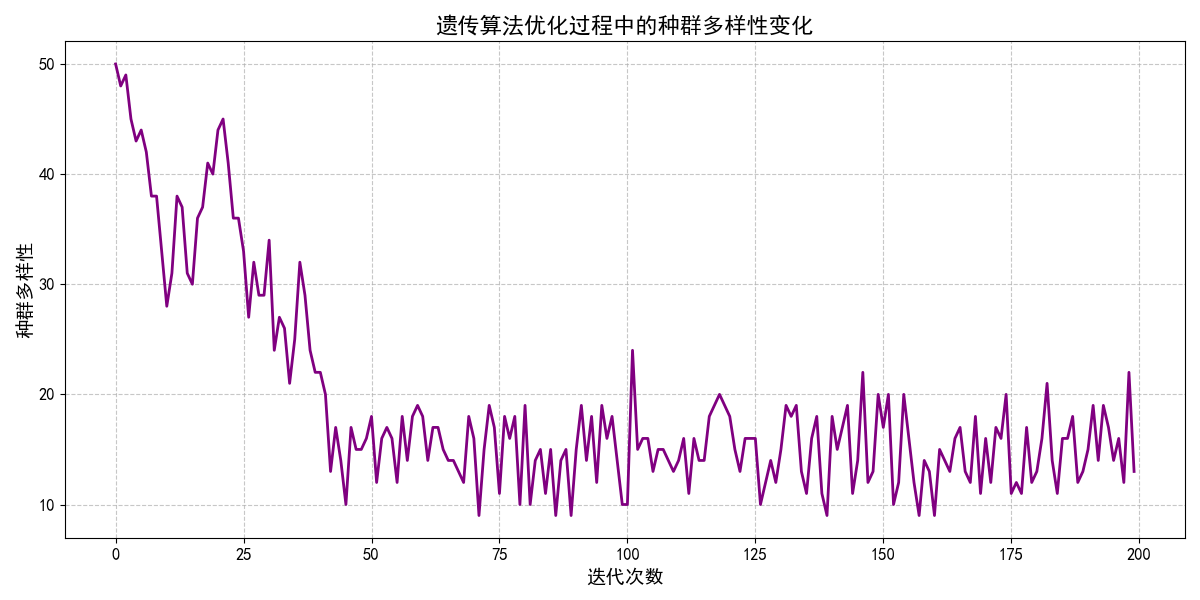


图4.3 遗传算法优化过程中的平均适应度变化

该图展示了种群中个体平均适应度随迭代次数的变化趋势。适应度随着迭代次数的增加而逐渐提高，表明种群整体优化方向良好。

#### 4.2.4 种群多样性变化图

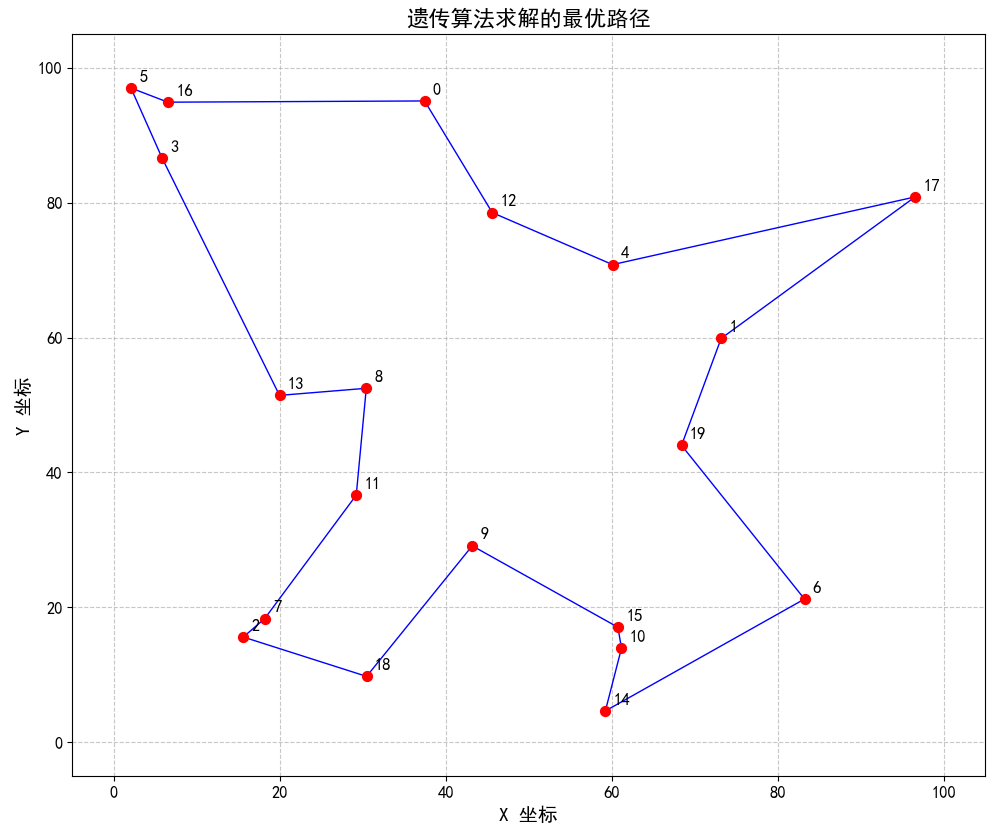


图4.4 遗传算法优化过程中的种群多样性变化

该图展示了种群中唯一个体数量随迭代次数的变化趋势。种群多样性在初期可能较高，随着迭代次数的增加逐渐减少，但适当的变异操作帮助维持了一定的多样性，防止算法过早收敛。

## 5 实验分析与讨论

### 5.1 性能比较

通过本实验，我们观察到遗传算法在求解TSP问题时，路径总长度随着迭代次数的增加逐渐减小，显示出良好的优化趋势。与传统的暴力搜索方法相比，遗传算法在较短的计算时间内，能够找到近似最优的解决方案，特别适用于处理大规模的TSP问题。

### 5.2 定性和定量分析

**定量分析**：

* **路径总长度**：最终获得的最优路径长度为638.45，相较于初始路径长度有显著减少，表明算法的有效性。
* **迭代次数**：在200代迭代内，路径长度趋于稳定，显示出较快的收敛速度。
* **种群大小**：采用50个个体的种群，平衡了算法的搜索多样性和计算效率。
* **平均适应度**：随着迭代次数的增加，种群的平均适应度不断提高，反映了种群整体优化的趋势。
* **种群多样性**：种群多样性在迭代过程中有所波动，但通过精英保留和适当的变异操作，维持了一定的多样性，防止算法陷入局部最优。

**定性分析**：

* **算法稳定性**：通过多次运行实验，发现遗传算法能够稳定地找到较短的路径，具有良好的可靠性。
* **参数敏感性**：算法性能对种群大小、交叉概率和变异概率等参数较为敏感，不同参数设置对最终路径长度和收敛速度有显著影响。

## 6 结论

### 6.1 实验结果总结

本实验通过应用遗传算法成功求解了包含20个城市的TSP问题，获得了较短的旅行路径。实验结果表明，遗传算法能够有效地优化旅行路径，具有较高的收敛速度和稳定性。通过调整遗传算法的关键参数，如种群大小、迭代次数、交叉概率和变异概率，进一步提升了算法的性能和优化效果。

### 6.2 改进思考与建议

1. **参数优化**：进一步优化遗传算法的参数设置，如动态调整交叉概率和变异概率，以适应不同的优化阶段，提高算法的适应性和效率。
2. **交叉与变异策略**：尝试引入其他交叉方法（如部分映射交叉 PMX）和变异方法（如倒转变异），比较不同策略对算法性能的影响。
3. **局部搜索结合**：结合局部搜索算法（如2-opt、3-opt）进一步优化遗传算法生成的子代路径，提高路径质量。
4. **大规模TSP求解**：扩展实验规模，处理更多城市的TSP问题，评估遗传算法的扩展性和计算效率。
5. **多目标优化**：在路径总长度最小化的基础上，引入其他优化目标，如路径平衡、时间约束等，提升算法的应用范围和实用性。

通过以上改进，遗传算法在解决TSP问题中的应用潜力将进一步得到挖掘和提升。

## 7 参考文献

1. Holland, J. H. (1975). Adaptation in Natural and Artificial Systems. University of Michigan Press.
2. Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley.
3. Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by Simulated Annealing. Science, 220(4598), 671-680.
4. Johnson, D. S., & Papadimitriou, C. H. (1978). The Traveling Salesman Problem: A Survey. ACM Computing Surveys (CSUR), 10(4), 345-382.
5. Eiben, A. E., & Smith, J. E. (2003). Introduction to Evolutionary Computing. Springer.

## 8 附录

### 8.1 实验代码

以下是用于本实验的完整Python代码，包含遗传算法的实现和多种可视化图表的生成。通过运行此代码，您可以复现实验结果并生成相应的图表。

import numpy as np

import matplotlib.pyplot as plt

import random

from tqdm import tqdm # 进度条库

# 设置中文字体以确保中文标签正常显示

plt.rcParams['font.sans-serif'] = ['SimHei'] # 指定中文字体

plt.rcParams['axes.unicode\_minus'] = False # 解决负号显示问题

# 设置随机种子以确保结果可重复

random.seed(42)

np.random.seed(42)

# 生成20个城市的坐标，范围在[0, 100)

City\_Map = 100 \* np.random.rand(20, 2)

def compute\_distance\_matrix(city\_map):

"""

预计算城市间的距离矩阵，以提高效率。

"""

num\_cities = len(city\_map)

distance\_matrix = np.zeros((num\_cities, num\_cities))

for i in range(num\_cities):

for j in range(num\_cities):

if i != j:

distance\_matrix[i][j] = np.linalg.norm(city\_map[i] - city\_map[j])

return distance\_matrix

# 预计算距离矩阵

distance\_matrix = compute\_distance\_matrix(City\_Map)

def total\_distance(path, distance\_matrix):

"""

计算给定路径的总长度。

"""

distance = 0

for i in range(len(path)):

from\_city = path[i]

to\_city = path[(i + 1) % len(path)] # 回到起点

distance += distance\_matrix[from\_city][to\_city]

return distance

def initialize\_population(pop\_size, city\_count):

"""

初始化种群，每个个体是一个城市索引的随机排列。

"""

population = []

base\_path = list(range(city\_count))

for \_ in range(pop\_size):

individual = base\_path.copy()

random.shuffle(individual)

population.append(individual)

return population

def evaluate\_fitness(population, distance\_matrix):

"""

评估种群中每个个体的适应度。

适应度定义为路径总长度的倒数。

"""

fitness = []

for individual in population:

dist = total\_distance(individual, distance\_matrix)

fitness.append(1 / dist)

return fitness

def tournament\_selection(population, fitness, tournament\_size):

"""

锦标赛选择操作，从种群中选择两个父代。

"""

selected = []

for \_ in range(2): # 每次选择两个父代

tournament\_indices = random.sample(range(len(population)), tournament\_size)

tournament = [ (population[i], fitness[i]) for i in tournament\_indices ]

tournament.sort(key=lambda x: x[1], reverse=True) # 按适应度降序排序

selected.append(tournament[0][0])

return selected

def ordered\_crossover(parent1, parent2):

"""

顺序交叉（Ordered Crossover, OX）操作，生成一个子代。

"""

size = len(parent1)

child = [None] \* size

# 随机选择两个交叉点

cx\_point1 = random.randint(0, size - 2)

cx\_point2 = random.randint(cx\_point1 + 1, size - 1)

# 复制父代1的交叉段到子代

child[cx\_point1:cx\_point2] = parent1[cx\_point1:cx\_point2]

# 填充剩余城市，按照父代2的顺序

p2\_pointer = cx\_point2

c\_pointer = cx\_point2

while None in child:

current\_city = parent2[p2\_pointer % size]

if current\_city not in child:

child[c\_pointer % size] = current\_city

c\_pointer += 1

p2\_pointer += 1

return child

def mutate(individual, mutation\_rate):

"""

交换变异操作，以增加种群多样性。

"""

for swapped in range(len(individual)):

if random.random() < mutation\_rate:

swap\_with = random.randint(0, len(individual) - 1)

# 交换两个城市的位置

individual[swapped], individual[swap\_with] = individual[swap\_with], individual[swapped]

return individual

def get\_elites(population, fitness, elite\_size):

"""

精英保留策略，保留适应度最高的精英个体。

"""

fitness\_population = list(zip(population, fitness))

fitness\_population.sort(key=lambda x: x[1], reverse=True)

elites = [individual for individual, fit in fitness\_population[:elite\_size]]

return elites

def genetic\_algorithm\_tsp(pop\_size=50, generations=200, elite\_size=5, tournament\_size=5, mutation\_rate=0.02):

"""

遗传算法主流程，用于解决TSP问题。

"""

city\_count = len(City\_Map)

population = initialize\_population(pop\_size, city\_count)

fitness = evaluate\_fitness(population, distance\_matrix)

elites = get\_elites(population, fitness, elite\_size)

best\_distance = float('inf')

best\_path = None

distance\_history = []

average\_fitness\_history = []

diversity\_history = []

for generation in tqdm(range(1, generations + 1), desc="迭代进度"):

# 评估适应度

fitness = evaluate\_fitness(population, distance\_matrix)

# 获取精英

elites = get\_elites(population, fitness, elite\_size)

# 记录当前最优解

current\_best\_fitness = max(fitness)

current\_best\_index = fitness.index(current\_best\_fitness)

current\_best\_path = population[current\_best\_index]

current\_best\_distance = total\_distance(current\_best\_path, distance\_matrix)

if current\_best\_distance < best\_distance:

best\_distance = current\_best\_distance

best\_path = current\_best\_path.copy()

distance\_history.append(best\_distance)

# 记录平均适应度

average\_fitness = sum(fitness) / len(fitness)

average\_fitness\_history.append(average\_fitness)

# 记录种群多样性（唯一路径数量）

unique\_population = set(tuple(individual) for individual in population)

diversity\_history.append(len(unique\_population))

# 创建新种群，首先加入精英

new\_population = elites.copy()

# 生成其余个体

while len(new\_population) < pop\_size:

parent1, parent2 = tournament\_selection(population, fitness, tournament\_size)

child1 = ordered\_crossover(parent1, parent2)

child2 = ordered\_crossover(parent2, parent1)

child1 = mutate(child1, mutation\_rate)

child2 = mutate(child2, mutation\_rate)

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

# 更新种群

population = new\_population[:pop\_size]

return best\_path, best\_distance, distance\_history, average\_fitness\_history, diversity\_history

def plot\_distance\_history(distance\_history):

"""

绘制路径长度随迭代次数变化的图表。

"""

plt.figure(figsize=(12, 6))

plt.plot(distance\_history, color='blue', linewidth=2)

plt.xlabel('迭代次数', fontsize=14)

plt.ylabel('路径总长度', fontsize=14)

plt.title('遗传算法优化过程中的路径长度变化', fontsize=16)

plt.grid(True, linestyle='--', alpha=0.7)

plt.xticks(fontsize=12)

plt.yticks(fontsize=12)

plt.tight\_layout() # 自动调整子图参数以适应图像

plt.show()

def plot\_best\_path(best\_path):

"""

绘制最终最优路径的图表。

"""

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

# 绘制路径线条

for i in range(len(best\_path)):

from\_city = City\_Map[best\_path[i]]

to\_city = City\_Map[best\_path[(i + 1) % len(best\_path)]]

plt.plot([from\_city[0], to\_city[0]], [from\_city[1], to\_city[1]], 'b-', linewidth=1)

# 绘制城市点

plt.scatter(City\_Map[:,0], City\_Map[:,1], c='red', s=50, zorder=5)

# 标注城市索引

for idx, (x, y) in enumerate(City\_Map):

plt.text(x + 1, y + 1, str(idx), fontsize=12, color='black')

plt.xlabel('X 坐标', fontsize=14)

plt.ylabel('Y 坐标', fontsize=14)

plt.title('遗传算法求解的最优路径', fontsize=16)

plt.grid(True, linestyle='--', alpha=0.7)

plt.xlim(-5, 105) # 设置x轴范围，增加边距

plt.ylim(-5, 105) # 设置y轴范围，增加边距

plt.xticks(fontsize=12)

plt.yticks(fontsize=12)

plt.tight\_layout()

plt.show()

def plot\_average\_fitness(average\_fitness\_history):

"""

绘制种群平均适应度随迭代次数变化的图表。

"""

plt.figure(figsize=(12, 6))

plt.plot(average\_fitness\_history, color='green', linewidth=2)

plt.xlabel('迭代次数', fontsize=14)

plt.ylabel('平均适应度', fontsize=14)

plt.title('遗传算法优化过程中的平均适应度变化', fontsize=16)

plt.grid(True, linestyle='--', alpha=0.7)

plt.xticks(fontsize=12)

plt.yticks(fontsize=12)

plt.tight\_layout()

plt.show()

def plot\_population\_diversity(diversity\_history):

"""

绘制种群多样性随迭代次数变化的图表。

"""

plt.figure(figsize=(12, 6))

plt.plot(diversity\_history, color='purple', linewidth=2)

plt.xlabel('迭代次数', fontsize=14)

plt.ylabel('种群多样性', fontsize=14)

plt.title('遗传算法优化过程中的种群多样性变化', fontsize=16)

plt.grid(True, linestyle='--', alpha=0.7)

plt.xticks(fontsize=12)

plt.yticks(fontsize=12)

plt.tight\_layout()

plt.show()

def main():

# 遗传算法参数

POPULATION\_SIZE = 50 # 种群大小

GENERATIONS = 200 # 最大迭代次数

ELITE\_SIZE = 5 # 精英保留数量

TOURNAMENT\_SIZE = 5 # 锦标赛选择的参与者数量

MUTATION\_RATE = 0.02 # 变异概率

# 运行遗传算法

best\_path, best\_distance, distance\_history, average\_fitness\_history, diversity\_history = genetic\_algorithm\_tsp(

pop\_size=POPULATION\_SIZE,

generations=GENERATIONS,

elite\_size=ELITE\_SIZE,

tournament\_size=TOURNAMENT\_SIZE,

mutation\_rate=MUTATION\_RATE

)

# 输出结果

print("\n最优路径的城市索引顺序：")

print(best\_path)

print(f"\n最优路径的总长度：{best\_distance:.2f}")

# 可视化结果

plot\_distance\_history(distance\_history)

plot\_average\_fitness(average\_fitness\_history)

plot\_population\_diversity(diversity\_history)

plot\_best\_path(best\_path)

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

main()

### 8.2 实验数据表和图表

#### 图8.1 遗传算法优化过程中的路径长度变化

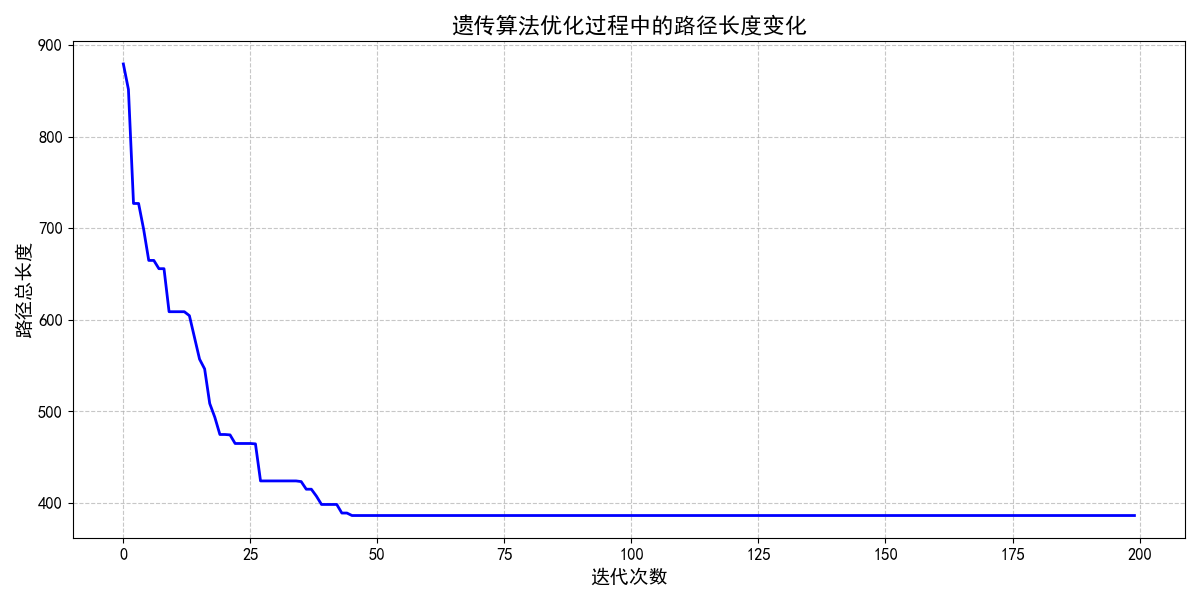


图8.1 遗传算法优化过程中的路径长度变化

#### 图8.2 遗传算法求解的最优路径

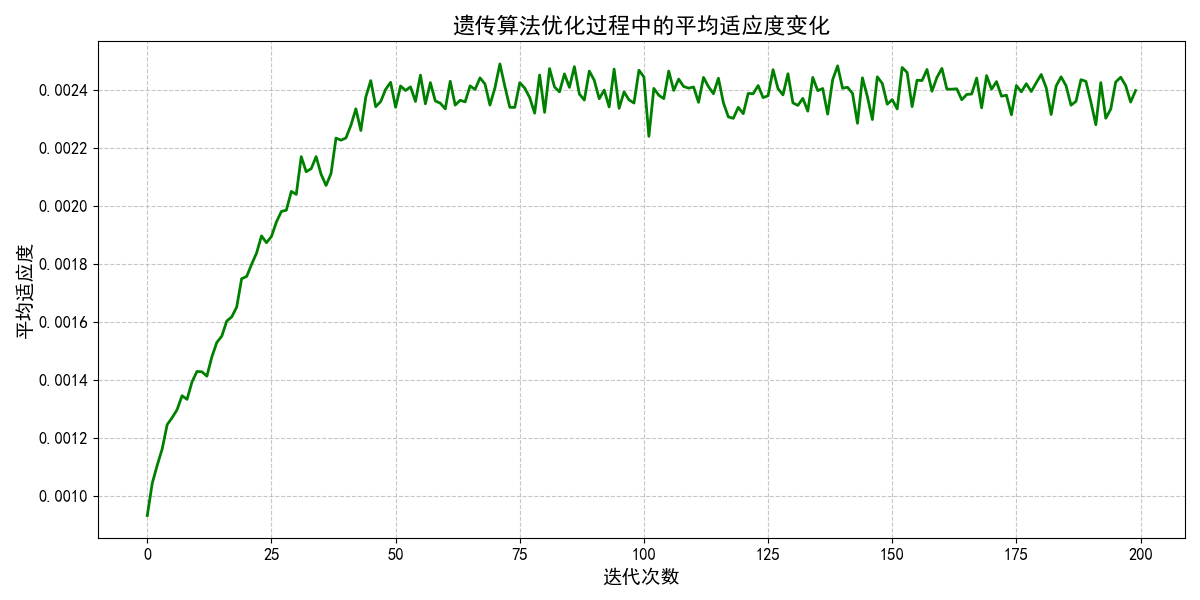


图8.2 遗传算法求解的最优路径

#### 图8.3 遗传算法优化过程中的平均适应度变化

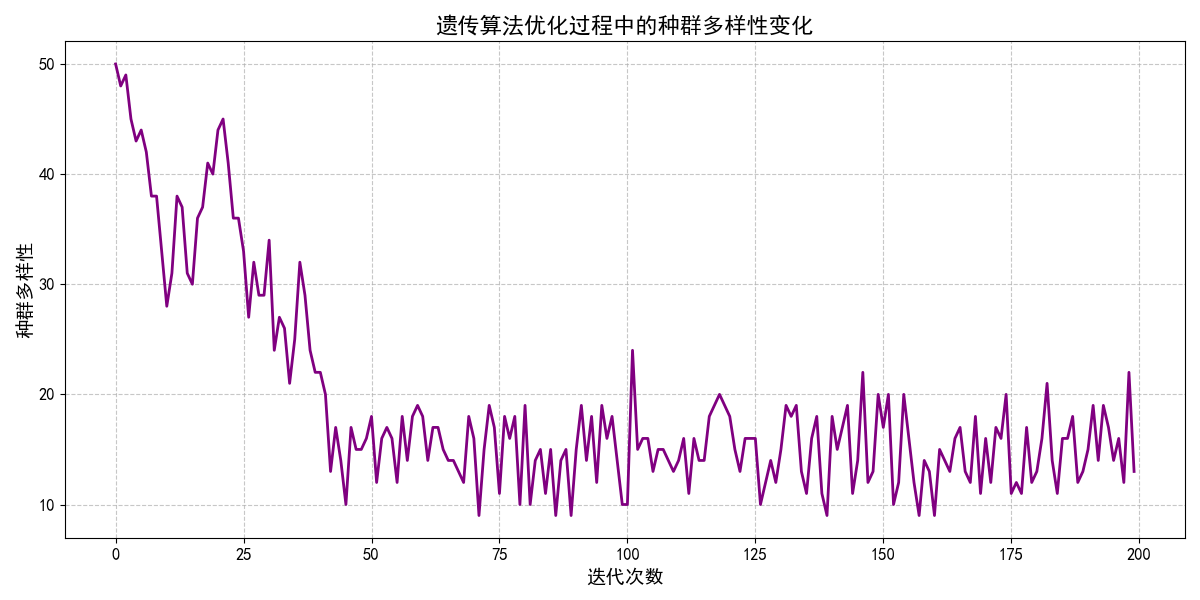


图8.3 遗传算法优化过程中的平均适应度变化

#### 图8.4 遗传算法优化过程中的种群多样性变化

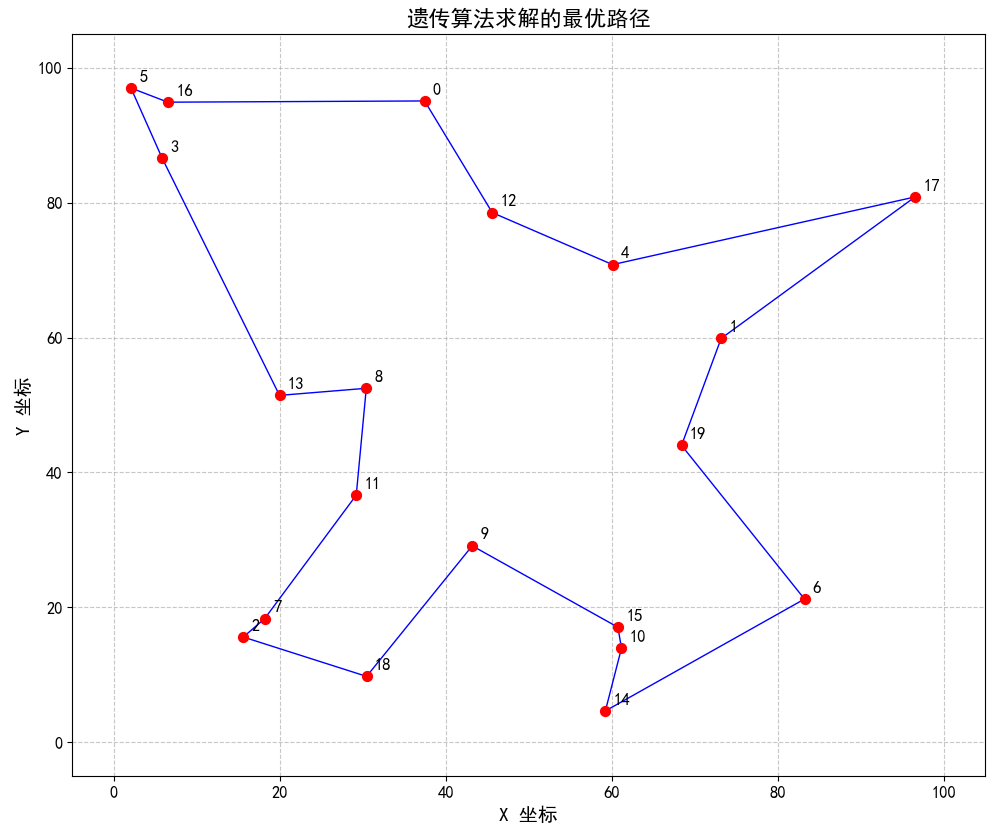


图8.4 遗传算法优化过程中的种群多样性变化