中山大学计算机学院

人工智能

本科生实验报告

课程名称：Artificial Intelligence

学号	23336179	姓名	马福泉

题目一：15-puzzle

一、 实验题目

2.启发式搜索–15-Puzzle

利用A*算法和IDA*算法解决15-Puzzle问题, 可自定义启发式函数. Puzzle问题的输入数据类型为二维嵌套list, 空位置用 `0`表示. 输出的解数据类型为 `list`, 是移动数字方块的次序.

若选择A*算法, 则函数名为 `A_star`; 若选择IDA*算法, 则函数名为 `IDA_star`.

例子: 输入

```
puzzle = [[1,2,3,4],[5,6,7,8],[9,10,11,12],[0,13,14,15]]
```

则调用 `A_star(puzzle)`或 `IDA_star(puzzle)`后输出解

```
[13,14,15]
```

测例:

```
1.[[1,2,4,8],[5,7,11,10],[13,15,0,3],[14,6,9,12]]
2.[[14,10,6,0],[4,9,1,8],[2,3,5,11],[12,13,7,15]]
3.[[5,1,3,4],[2,7,8,12],[9,6,11,15],[0,13,10,14]]
4.[[6,10,3,15],[14,8,7,11],[5,1,0,2],[13,12,9,4]]
5.[[11,3,1,7],[4,6,8,2],[15,9,10,13],[14,12,5,0]]
6.[[0,5,15,14],[7,9,6,13],[1,2,12,10],[8,11,4,3]]
```

二、 实验内容

1. 算法原理

1) 状态表示：使用 16 元素的元组（4x4 矩阵展开）表示拼图状态，例如(1,2,3,4,5,...,15,0)，其中 0 代表空格
状态变换：通过移动空格与相邻数字交换位置来生成新状态
目标检测：检查当前状态是否与预定义的目标状态 GOAL_STATE 完全一致

2) 算法框架

代码实现了标准 A*算法框架，包含三个核心组件：
开放集(Open Set)：使用优先队列（最小堆）存储待探索节点，按 f 值排序
闭合集(Closed Set)：字典结构记录已探索节点的最佳路径信息
启发式函数：曼哈顿距离作为 h(n)和错位块数估计剩余步数。（分别在写在 A_star_manhattan.py 和 A_star_misplace.py 两个代码）
初始化开放集(起始状态)

while 开放集不为空:
    取出 f 值最小的状态
    如果是目标状态，重建路径返回
    生成所有合法邻居状态
    对每个邻居：
        计算新 g 值（当前步数+1）
        如果是未探索状态或找到更优路径：
            计算 f 值(g+h)
            加入开放集
    将当前状态加入闭合集

3) 路径优化与重建

路径优化：在将邻居加入开放集前，检查闭合集中是否已存在更优路径
路径重建：到达目标后，通过回溯父状态指针重建完整解路径

4) IDA*对比 A*

① 迭代加深机制

通过逐步增加深度限制（bound）进行多次搜索
初始 bound = 初始状态的启发式值
每次迭代后 bound = 前次搜索中发现的最小超限 f 值

② 深度优先的搜索方式

采用递归 DFS 而非 A*的优先队列
仅维护当前路径而非全部开放集/闭合集

③ 隐式的边界控制

自动丢弃所有 f > bound 的节点
通过回溯自然实现剪枝

好处：
完备性：保证找到解（如果存在）
最优性：找到的解决方案步数最少
空间效率：相比传统 A*算法，内存使用量显著降低
渐进性：随着问题难度增加，性能下降较为平缓

2. 关键代码展示

1）曼哈顿距离启发函数：

```python
def get_manhattan_distance(state):
    """计算当前状态到目标状态的曼哈顿距离总和"""
    distance = 0
    for i, num in enumerate(state):
        if num != 0:  # 跳过空格
            goal_row, goal_col = GOAL_POSITIONS[num]
            curr_row, curr_col = i // 4, i % 4
            distance += abs(goal_row - curr_row) + abs(goal_col - curr_col)
    return distance
```

2）错位块启发函数

```python
def get_misplaced_tiles(state):
    """计算错位块数量"""
    misplaced = 0
    for i, num in enumerate(state):
        if num != 0 and num != GOAL_STATE[i]:
            misplaced += 1
    return misplaced
```

3）A*算法

```python
def a_star_search(initial_state):
    """使用A*算法解决15数码问题"""
    if initial_state == GOAL_STATE:
        return [initial_state]

    # 优先队列 - (f值, 移动步数, 状态, 父状态)
    open_set = [(get_manhattan_distance(initial_state), 0, initial_state, None)]
    # 使用字典记录已访问状态的最佳路径信息
    closed_set = {}

    while open_set:
        # 获取f值最小的状态
        _, moves, current_state, parent = heapq.heappop(open_set)
        # 如果已经处理过此状态且有更好的路径，则跳过
        if current_state in closed_set and closed_set[current_state][0] <= moves:
            continue
        # 记录当前状态的最佳路径信息
        closed_set[current_state] = (moves, parent)
        # 检查是否达到
        if current_state == GOAL_STATE:
            # 重建路径
            path = []
            while current_state:
                path.append(current_state)
                _, current_state = closed_set.get(current_state, (0, None))
            return path[::-1]  # 反转路径
        # 探索邻居状态
        for neighbor in get_neighbors(current_state):
            new_moves = moves + 1
            # 如果已经访问过且没有更好的路径，则跳过
            if neighbor in closed_set and closed_set[neighbor][0] <= new_moves:
                continue
            # 计算f值 = g值(移动步数) + h值(曼哈顿距离)
            f_value = new_moves + get_manhattan_distance(neighbor)
            heapq.heappush(open_set, (f_value, new_moves, neighbor, current_state))
```

4)IDA*算法

```python
def ida_star_search(initial_state):
    """使用IDA*算法解决15数码问题"""
    if initial_state == GOAL_STATE:
        return [initial_state]

    # 初始的深度限制为启发式函数值（曼哈顿距离）
    bound = get_manhattan_distance(initial_state)
    path = [initial_state]
    visited = {}  # 存储访问过的状态

    while True:
        # 进行深度受限的搜索
        t = search(path, 0, bound, visited)
        if t == "FOUND":
            return path  # 找到解决方案
        if t == float('inf'):
            return None  # 无解
        bound = t  # 更新深度限制
        visited.clear()
```

3. 创新点&优化

1）详细的移动描述：

```
def describe_move(prev_state, next_state):
    # ...计算移动方向和数字...
    return f"数字 {moved_number} 向{direction}移动"
```

2）可视化输出：

```
def print_board(state):
    for i in range(0, 16, 4):
        print(' '.join(f'{state[i+j]:2d}' for j in range(4)))
```

3）可以检查无解情况

（1）计算逆序数（inversions）：
将拼图状态展开为一个一维列表（从左到右、从上到下），忽略空格（0）。
统计所有数字对 (i, j)（其中 i < j 但 state[i] > state[j]）的数量，即逆序数。
确定空格所在行(从下往上数,即最下面一行是第 1 行,最上面一行是第 4 行)。
（2）判断可解性：
如果空格在偶数行（从下往上数的第 2 或第 4 行），则逆序数必须是奇数才能有解。
如果空格在奇数行（从下往上数的第 1 或第 3 行），则逆序数必须是偶数才能有解

```python
def is_solvable(state):
    # 计算逆序数
    inversions = 0
    for i in range(len(state)):
        if state[i] == 0:
            continue  # 跳过空格
        for j in range(i+1, len(state)):
            if state[j] == 0:
                continue  # 跳过空格
            if state[i] > state[j]:
                inversions += 1

    # 获取空格所在行（从下往上数）
    blank_idx = state.index(0)
    blank_row = 4 - (blank_idx // 4)  # 从下往上数的行号（1-4）

    # 判断是否有解
    if blank_row % 2 == 0:  # 空格在偶数行
        return inversions % 2 == 1
    else:  # 空格在奇数行
        return inversions % 2 == 0
```

```
初始状态:
-----------
 2  1  3  4
 5  6  7  8
 9 10 11 12
13 14 15  0
-----------

此状态无解!
```

三、 实验结果及分析

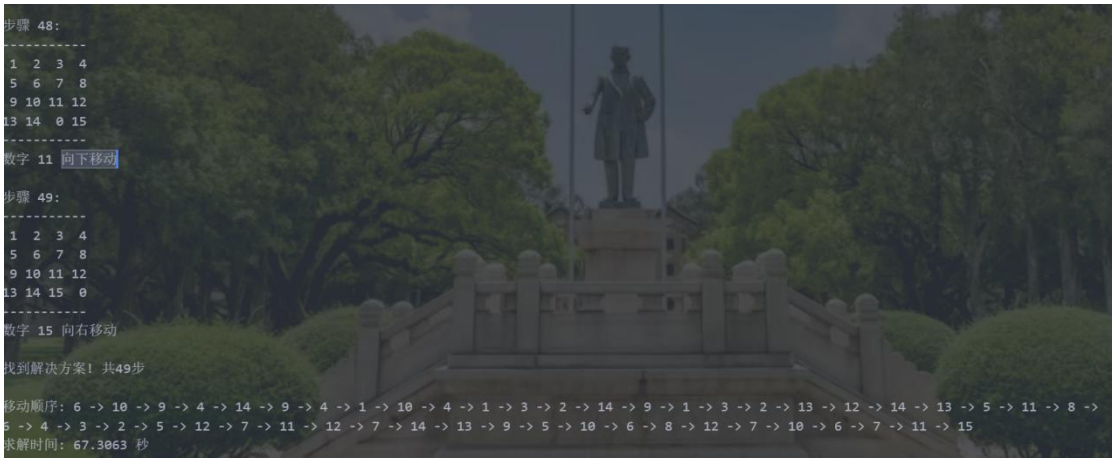
1. 实验结果展示示例（修改 initial_state_2d = INITIAL_STATE_1 即可选择不同样例）

（1）A_star_manhattan.py

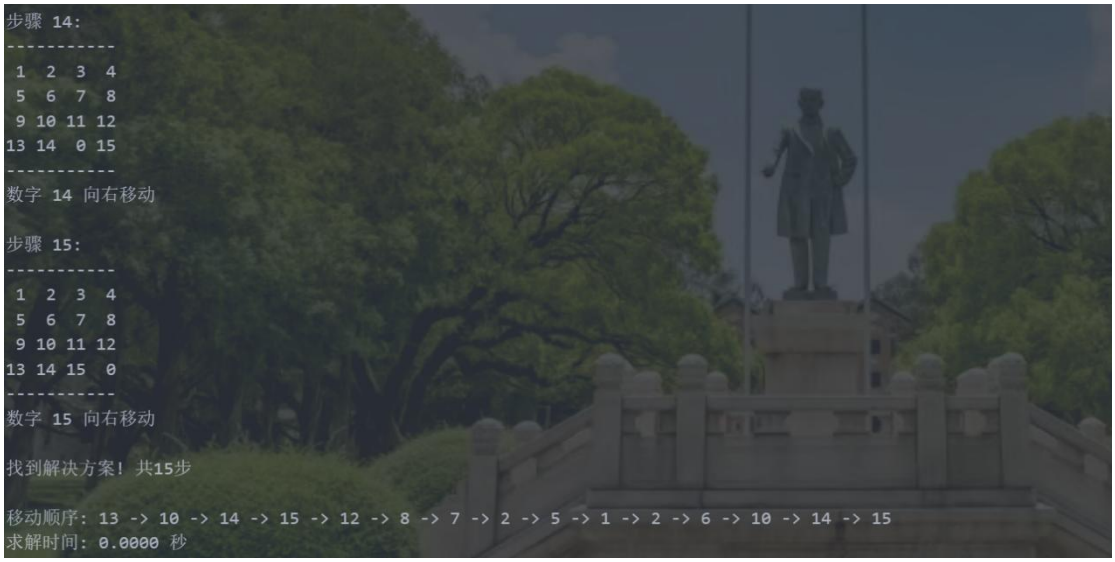
样例一（中间过程忽略）


步骤 21:
1 2 3 4
5 6 7 8
9 10 11 0
13 14 15 12
数字 11 向右移动
步骤 22:
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0
数字 12 向下移动
找到解决方案！共22步
移动顺序: 15 -> 6 -> 9 -> 15 -> 11 -> 10 -> 3 -> 11 -> 10 -> 3 -> 8 -> 4 -> 3 -> 7 -> 6 -> 9 -> 14 -> 13 -> 9 -> 10 -> 11 -> 12
求解时间: 0.0032 秒

样例二


步骤 48:
1 2 3 4
5 6 7 8
9 10 11 12
13 14 0 15
数字 11 向下移动
步骤 49:
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0
数字 15 向右移动
找到解决方案！共49步
移动顺序: 6 -> 10 -> 9 -> 4 -> 14 -> 9 -> 4 -> 1 -> 10 -> 4 -> 1 -> 3 -> 2 -> 14 -> 9 -> 1 -> 3 -> 2 -> 13 -> 12 -> 14 -> 13 -> 5 -> 11 -> 8 ->
6 -> 4 -> 3 -> 2 -> 5 -> 12 -> 7 -> 11 -> 12 -> 7 -> 14 -> 13 -> 9 -> 5 -> 10 -> 6 -> 8 -> 12 -> 7 -> 10 -> 6 -> 7 -> 11 -> 15
求解时间: 67.3063 秒

样例三


步骤 14:
1 2 3 4
5 6 7 8
9 10 11 12
13 14 0 15
数字 14 向右移动
步骤 15:
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0
数字 15 向右移动
找到解决方案！共15步
移动顺序: 13 -> 10 -> 14 -> 15 -> 12 -> 8 -> 7 -> 2 -> 5 -> 1 -> 2 -> 6 -> 10 -> 14 -> 15
求解时间: 0.0000 秒

样例四

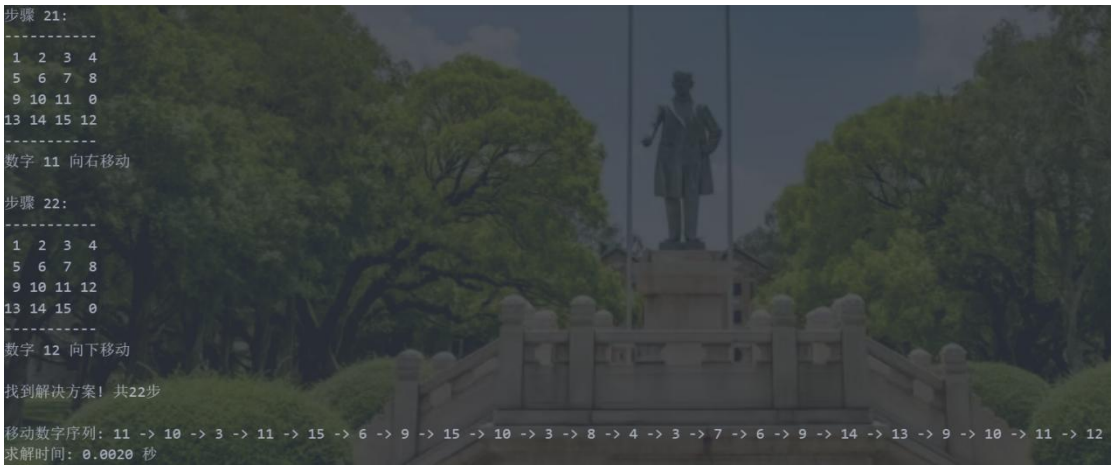
```
步骤 47:
-----------
 1  2  3  4
 5  6  7  8
 9 10 11  0
13 14 15 12
-----------
数字 11 向右移动

步骤 48:
-----------
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15  0
-----------
数字 12 向下移动

找到解决方案！共48步

移动顺序: 9 -> 12 -> 13 -> 5 -> 1 -> 9 -> 7 -> 11 -> 2 -> 4 -> 12 -> 13 -> 9 -> 7 -> 11 -> 2 -> 15 -> 3 -> 2 -> 15 -> 4 -> 11 -> 15 -> 8 -> 14
-> 1 -> 5 -> 9 -> 13 -> 15 -> 7 -> 14 -> 10 -> 6 -> 1 -> 5 -> 9 -> 13 -> 14 -> 10 -> 6 -> 2 -> 3 -> 4 -> 8 -> 7 -> 11 -> 12
求解时间: 91.3360 秒
```

样例五和样例六用 A*算法需要保存所有扩展过的节点，对于 15 数码问题这样的大型问题，状态空间太大导致内存不足。所以在 IDA*算法中解决

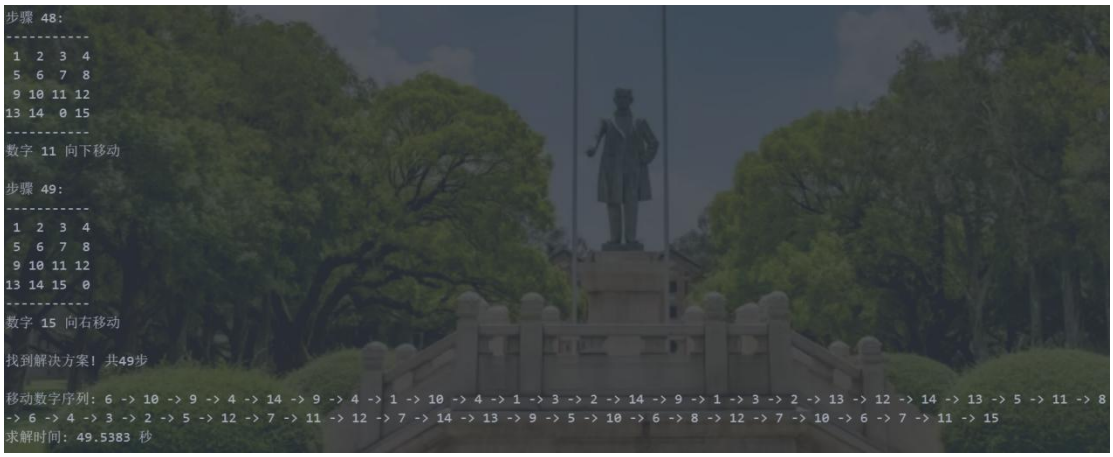
样例七

2） IDA_star_manhattan.py

样例一（中间过程忽略）



样例二


步骤 48:
----
1 2 3 4
5 6 7 8
9 10 11 12
13 14 0 15

数字 11 向下移动
步骤 49:
----
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0

数字 15 向右移动
找到解决方案！共49步
移动数字序列: 6 -> 10 -> 9 -> 4 -> 14 -> 9 -> 4 -> 1 -> 10 -> 4 -> 1 -> 3 -> 2 -> 14 -> 9 -> 1 -> 3 -> 2 -> 13 -> 12 -> 14 -> 13 -> 5 -> 11 -> 8 -> 6 -> 4 -> 3 -> 2 -> 5 -> 12 -> 7 -> 11 -> 12 -> 7 -> 14 -> 13 -> 9 -> 5 -> 10 -> 6 -> 8 -> 12 -> 7 -> 10 -> 6 -> 7 -> 11 -> 15
求解时间: 49.5383 秒

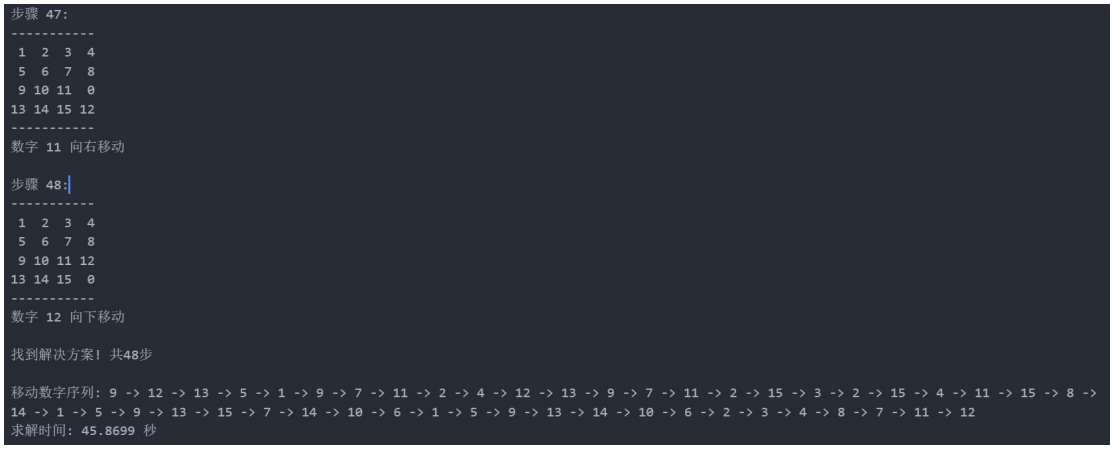
样例三


步骤 14:
----
1 2 3 4
5 6 7 8
9 10 11 12
13 14 0 15

数字 14 向右移动
步骤 15:
----
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0

数字 15 向右移动
找到解决方案！共15步
移动数字序列: 13 -> 10 -> 14 -> 15 -> 12 -> 8 -> 7 -> 2 -> 5 -> 1 -> 2 -> 6 -> 10 -> 14 -> 15
求解时间: 0.0000 秒
[Done] exited with code=0 in 0.087 seconds

样例四


步骤 47:
----
1 2 3 4
5 6 7 8
9 10 11 0
13 14 15 12

数字 11 向右移动
步骤 48:
----
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0

数字 12 向下移动
找到解决方案！共48步
移动数字序列: 9 -> 12 -> 13 -> 5 -> 1 -> 9 -> 7 -> 11 -> 2 -> 4 -> 12 -> 13 -> 9 -> 7 -> 11 -> 2 -> 15 -> 3 -> 2 -> 15 -> 4 -> 11 -> 15 -> 8 -> 14 -> 1 -> 5 -> 9 -> 13 -> 15 -> 7 -> 14 -> 10 -> 6 -> 1 -> 5 -> 9 -> 13 -> 14 -> 10 -> 6 -> 2 -> 3 -> 4 -> 8 -> 7 -> 11 -> 12
求解时间: 45.8699 秒

样例五

```
步骤 57:
-----------
 1  2  3  4
 5  6  7  8
 9 10 11  0
13 14 15 12
-----------
数字 8 向下移动
步骤 58:
-----------
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15  0
-----------
数字 12 向下移动

找到解决方案！共58步
移动数字序列: 13 -> 10 -> 8 -> 6 -> 9 -> 12 -> 5 -> 13 -> 10 -> 8 -> 12 -> 15 -> 14 -> 5 -> 13 -> 12 -> 15 -> 14 -> 5 -> 13 -> 14 -> 9 -> 4 -> 11
-> 3 -> 1 -> 6 -> 4 -> 11 -> 3 -> 1 -> 6 -> 4 -> 2 -> 7 -> 4 -> 2 -> 7 -> 8 -> 10 -> 12 -> 15 -> 10 -> 8 -> 7 -> 11 -> 3 -> 5 -> 9 -> 10 -> 11
-> 3 -> 6 -> 2 -> 3 -> 7 -> 8 -> 12
求解时间: 36.5566 秒

[Done] exited with code=0 in 36.668 seconds
```

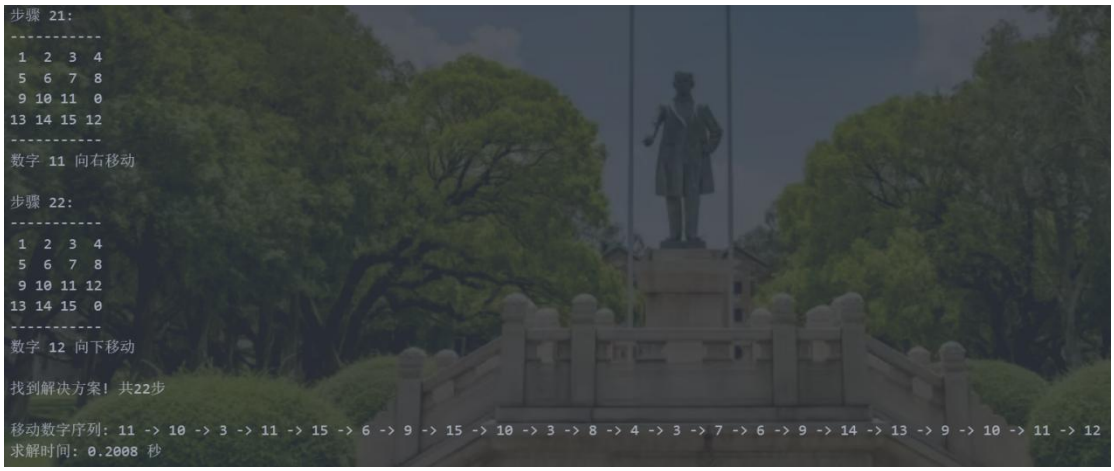
样例六

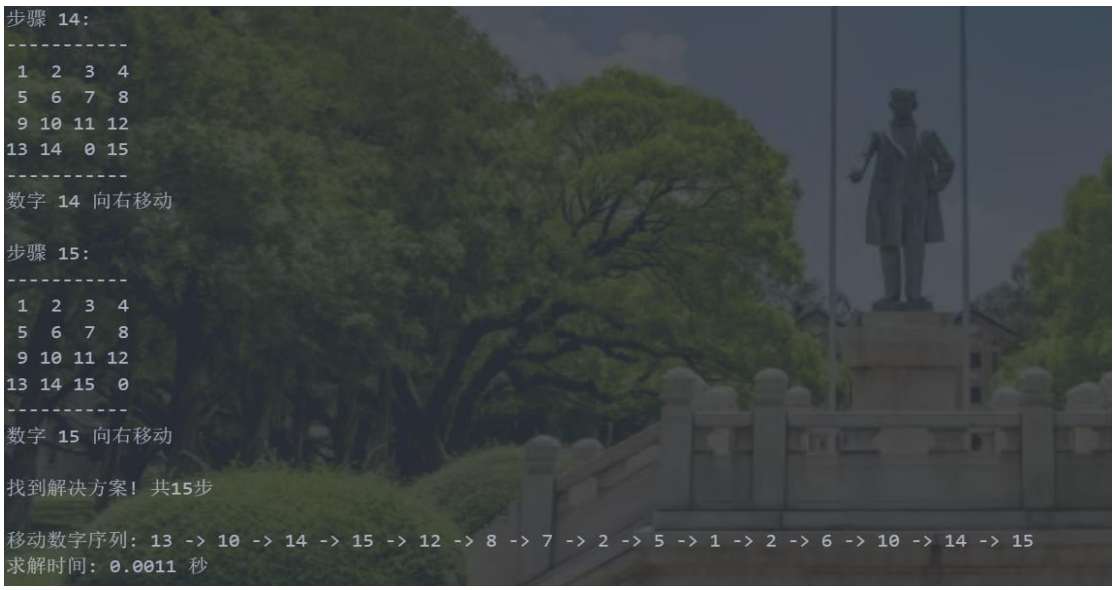
```
步骤 61:
-----------
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14  0 15
-----------
数字 14 向右移动
步骤 62:
-----------
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15  0
-----------
数字 15 向右移动

找到解决方案！共62步
移动数字序列: 7 -> 9 -> 2 -> 1 -> 9 -> 2 -> 5 -> 7 -> 2 -> 5 -> 1 -> 11 -> 8 -> 9 -> 5 -> 1 -> 6 -> 12 -> 10 -> 3 -> 4 -> 8 -> 11 -> 10 -> 12 ->
13 -> 3 -> 4 -> 8 -> 12 -> 13 -> 15 -> 14 -> 3 -> 4 -> 8 -> 12 -> 13 -> 15 -> 14 -> 7 -> 2 -> 1 -> 5 -> 10 -> 11 -> 13 -> 15 -> 14 -> 7 -> 3 ->
4 -> 8 -> 12 -> 15 -> 14 -> 11 -> 10 -> 9 -> 13 -> 14 -> 15
求解时间: 317.2771 秒
```

下面采用错位块启发函数 2A_star_misplace.py，样例仅举几例（中间过程忽略）

```
步骤 21:
-----------
 1  2  3  4
 5  6  7  8
 9 10 11  0
13 14 15 12
-----------
数字 11 向右移动

步骤 22:
-----------
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15  0
-----------
数字 12 向下移动

找到解决方案！共22步

移动数字序列: 15 -> 6 -> 9 -> 15 -> 11 -> 10 -> 3 -> 11 -> 10 -> 3 -> 8 -> 4 -> 3 -> 7 -> 6 -> 9 -> 14 -> 13 -> 9 -> 10 -> 11 -> 12
求解时间: 0.1767 秒
```

IDA_star_misplace.py





2. 评测指标展示及分析

对比上述样例运行数据
（1）曼哈顿距离通过计算每个数字到目标位置的横向和纵向步数之和，提供了更精确的代价估计，能有效引导搜索方向，显著减少 A 或 IDA 需要扩展的节点数，尤其适用于复杂状态的 15-puzzle 问题。相比之下，错位块函数仅统计位置错误的数字数量，虽然计算更快，但启发信息过于粗糙，常低估实际代价，导致搜索效率低下，仅在简单问题适用。
（2）A 算法通过优先队列动态选择最优节点，借助开放集和闭合集确保完备性和最优性，适合中等规模问题，但内存消耗较大（需存储所有待扩展节点）。IDA 则以迭代加深方式逐步放宽深度限制，结合深度优先搜索和剪枝策略，仅维护当前路径，内存占用极低（O(d)），适合大规模或深度未知的问题，但可能因重复搜索牺牲部分时间效率。两者均依赖启发函数，但 IDA 在内存受限时更具优势，而 A 在解较浅时通常更快。

四 参考资料

（1）部分 debug，优化建议参考了 ai 大模型的建议。
（2）阅读参考了以下文章：
A*与 IDA*算法在 15-puzzle 问题中的优化与比较-CSDN 博客

15 Puzzle（15 数码，IDA* 及 N 数码的 有解无解的判读） - 代码先锋网

题目二：遗传算法解决 TSP

一、 实验题目

利用遗传算法求解 TSP 问题

- 在National Traveling Salesman Problems (uwaterloo.ca) （https://www.math.uwaterloo.ca/tsp/world/countries.html）中任选两个TSP问题的数据集。
- 建议：
  - 对于规模较大的TSP问题，遗传算法可能需要运行几分钟甚至几个小时的时间才能得到一个比较好的结果. 因此建议先用城市数较小的数据集测试算法正确与否，再用城市数较大的数据集来评估算法性能。
  - 由于遗传算法是基于随机搜索的算法，只运行一次算法的结果并不能反映算法的性能. 为了更好地分析遗传算法的性能，应该以不同的初始随机种子或用不同的参数(例如种群数量，变异概率等)多次运行算法，这些需要在实验报告中呈现.

二、 实验内容

1. 算法原理

1) 问题建模与表示

每个城市用二维坐标(x,y)表示

路径表示为一个城市的排列序列

使用欧几里得距离计算城市间距离，并预先构建距离矩阵提高效率

2)遗传算法框架

(1) 初始化种群：

随机生成多个可行解（路径）构成初始种群

种群大小(pop_size)影响搜索空间和计算效率

(2) 适应度评估：

计算每个个体的适应度值：采用倒数形式计算

这种设计保证路径越短适应度越高，且避免负值

(3) 选择操作：

使用轮盘赌选择（比例选择）：每个个体被选中的概率与其适应度成正比

(4) 交叉操作:采用顺序交叉(Order Crossover, OX)

随机选择两个切割点，保留父代 1 的片段

按父代 2 顺序填充剩余城市，保持排列有效性

(5) 变异操作：用交换变异：以 mutation_rate 概率随机交换路径中两个城市位置

引入随机扰动，维持种群多样性

(6) 精英保留策略：每代保留最优的 elitism_ratio 比例个体

防止优秀解在进化过程中丢失

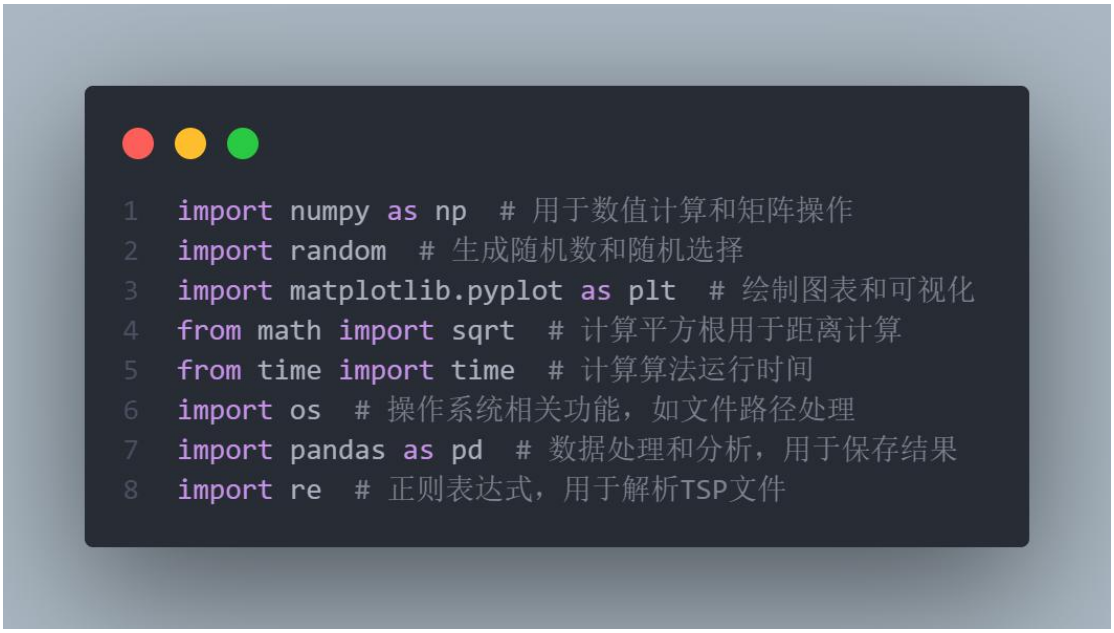
(7) 终止条件：

达到最大迭代次数(num_generations)

(也可设置收敛条件,如适应度不再提升）

2. 关键代码展示

所需模块



```python
import numpy as np  # 用于数值计算和矩阵操作
import random  # 生成随机数和随机选择
import matplotlib.pyplot as plt  # 绘制图表和可视化
from math import sqrt  # 计算平方根用于距离计算
from time import time  # 计算算法运行时间
import os  # 操作系统相关功能，如文件路径处理
import pandas as pd  # 数据处理和分析，用于保存结果
import re  # 正则表达式，用于解析TSP文件
```

核心算法函数，具体代码过长，见文件 tsp.py

```
class TSPSolver:
```

```
        def __init__(self, cities):
            # 初始化城市数据和距离矩阵

        def _create_distance_matrix(self):
            # 构建距离矩阵

        def _initialize_population(self, pop_size):
            # 初始化种群

        def _calculate_fitness(self, individual):
            # 计算适应度
```

```
        def _roulette_wheel_selection(self, population, fitness_values, num_parents):
            # 轮盘赌选择

        def _order_crossover(self, parent1, parent2):
            # 顺序交叉

        def _swap_mutation(self, individual, mutation_rate):
            # 交换变异

        def solve(self, params):
            # 主求解流程

        def visualize(self, route, history, title=None, result_dir=None):
            # 结果可视化
```

3. 创新点&优化

1) 可视化

```python
def visualize(self, route, history, title=None, result_dir=None):
        plt.figure(figsize=(15, 5))  # 创建大图

        # 第一个子图：收敛记录
        plt.subplot(1, 2, 1)
        plt.plot(history['best'], label='Best Distance')  # 最佳距离曲线
        plt.plot(history['avg'], label='Average Distance')  # 平均距离曲线
        plt.xlabel('Generation')  # x轴标签
        plt.ylabel('Distance')  # y轴标签
        plt.title('Convergence History')  # 标题
        plt.legend()  # 显示图例

        # 第二个子图：TSP路径
        plt.subplot(1, 2, 2)
        # 获取坐标
        x = [self.cities[i][0] for i in route] + [self.cities[route[0]][0]]
        y = [self.cities[i][1] for i in route] + [self.cities[route[0]][1]]
        plt.plot(x, y, 'o-')  # 绘制路径
        plt.title(f'TSP Route (Distance: {history["best"][-1]:.2f})')
        plt.tight_layout()  # 调整布局
        if title and result_dir:
            plt.savefig(os.path.join(result_dir, f"{title}.png"))  # 保存图片
        plt.show()  # 显示图片
        plt.figure(figsize=(15, 5))
        plt.subplot(1, 2, 1)
        plt.plot(history['best'], label='Best Distance')
        plt.plot(history['avg'], label='Average Distance')
        plt.xlabel('Generation')
        plt.ylabel('Distance')
        plt.title('Convergence History')
        plt.legend()

        plt.subplot(1, 2, 2)
        x = [self.cities[i][0] for i in route] + [self.cities[route[0]][0]]
        y = [self.cities[i][1] for i in route] + [self.cities[route[0]][1]]
        plt.plot(x, y, 'o-')
        plt.title(f'TSP Route (Distance: {history["best"][-1]:.2f})')

        plt.tight_layout()
        if title and result_dir:
            plt.savefig(os.path.join(result_dir, f"{title}.png"))
        plt.show()
```

2) 多次运行统计稳定性指标（均值、标准差）

```python
# 计算多次运行的统计数据
        param_results['min_distance'] = min(param_results['distances'])  # 最小距离
        param_results['max_distance'] = max(param_results['distances'])  # 最大距离
        param_results['avg_distance'] = sum(param_results['distances']) / len(param_results['distances'])  # 平均距离
        param_results['avg_time'] = sum(param_results['times']) / len(param_results['times'])  # 平均执行时间
        # 将果添加到总结果列表中
        results.append(param_results)
        # 打印
        print(f"结果:")
        print(f"  最小距离: {param_results['min_distance']:.2f}")
        print(f"  最大距离: {param_results['max_distance']:.2f}")
        print(f"  平均距离: {param_results['avg_distance']:.2f} ± {param_results['std_distance']:.2f}")
        print(f"  平均时间: {param_results['avg_time']:.2f}秒")
```

3) CSV 结果导出

```python
# 将所有参数组合的结果准备保存到CSV文件
    df_data = []
    for r in results:
        df_data.append({
            'pop_size': r['pop_size'],
            'num_generations': r['num_generations'],
            'mutation_rate': r['mutation_rate'],
            'elitism_ratio': r['elitism_ratio'],
            'min_distance': r['min_distance'],
            'max_distance': r['max_distance'],
            'avg_distance': r['avg_distance'],
            'std_distance': r['std_distance'],
            'avg_time': r['avg_time']
        })

```

三、 实验结果及分析

1. 实验结果展示示例

每个样例进行以下三个参数的运行，每个参数运行三次，取平均值。

pop_size=50, generations=300, mutation_rate=0.01, elitism_ratio=0.1

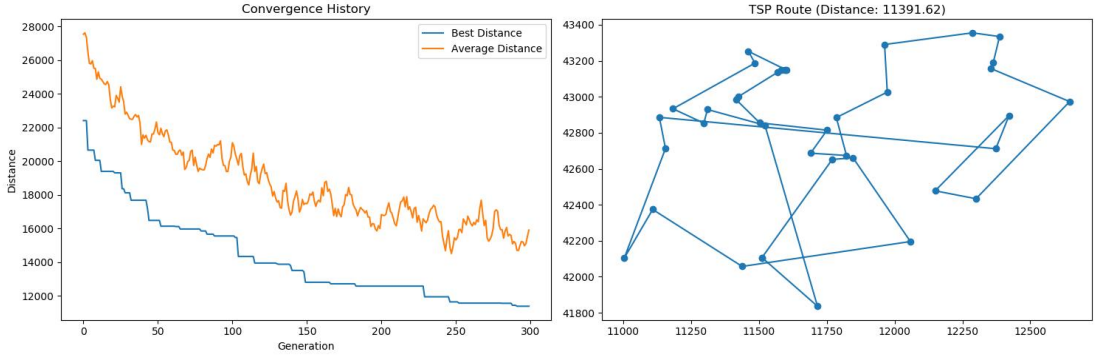
pop_size=100, generations=500, mutation_rate=0.02, elitism_ratio=0.1

pop_size=150, generations=800, mutation_rate=0.05, elitism_ratio=0.2

2. 样例一：吉布提

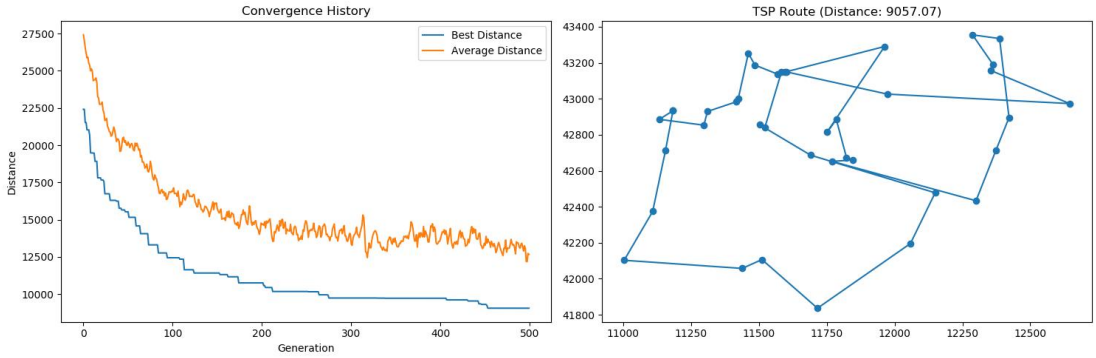
pop_size=50, generations=300, mutation_rate=0.01, elitism_ratio=0.1

```
==========================================
处理数据集: dj38.tsp
==========================================
尝试读取文件: c:\Users\86157\Desktop\AI\LAB\Code\lab4_search\TSP\tsp_data\dj38.tsp
成功读取了 38 个城市坐标

开始参数组合: pop_size=50, generations=300, mutation_rate=0.01, elitism_ratio=0.1
运行 1/3...
运行 1 完成 - 距离: 11167.97, 时间: 0.68秒

==== TSP 求解结果 ====
城市数量: 38
最优路线: [13, 20, 14, 6, 5, 4, 11, 10, 18, 17, 16, 15, 8, 7, 12, 21, 19, 24, 23, 27, 26, 30, 35, 33, 32, 37, 31, 29, 36, 34, 2, 3, 0, 1, 9, 28,
25, 22]
路线距离: 11167.97
求解时间: 0.68 秒
迭代次数: 300
====================

运行 2/3...
运行 2 完成 - 距离: 9551.59, 时间: 0.72秒
运行 3/3...
运行 3 完成 - 距离: 11359.30, 时间: 0.70秒
结果:
  最小距离: 9551.59
  最大距离: 11359.30
  平均距离: 10692.95 ± 810.84
  平均时间: 0.70秒
```

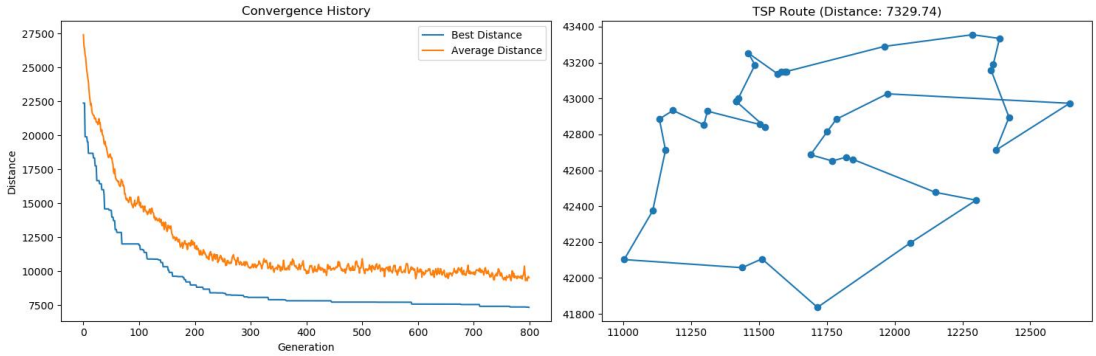


pop_size=100, generations=500, mutation_rate=0.02, elitism_ratio=0.1

```
开始参数组合: pop_size=100, generations=500, mutation_rate=0.02, elitism_ratio=0.1
运行 1/3...
运行 1 完成 - 距离: 9057.07, 时间: 2.39秒

==== TSP 求解结果 ====
城市数量: 38
最优路线: [4, 2, 5, 6, 7, 8, 10, 11, 15, 17, 26, 21, 23, 24, 25, 22, 31, 34, 36, 35, 30, 33, 32, 37, 27, 18, 16, 14, 12, 19, 29, 28, 20, 13, 9,
0, 1, 3]
路线距离: 9057.07
求解时间: 2.39 秒
迭代次数: 500
====================

运行 2/3...
运行 2 完成 - 距离: 9346.72, 时间: 2.40秒
运行 3/3...
运行 3 完成 - 距离: 8260.22, 时间: 2.35秒
结果:
  最小距离: 8260.22
  最大距离: 9346.72
  平均距离: 8888.01 ± 459.39
  平均时间: 2.38秒
```



pop_size=150, generations=800, mutation_rate=0.05, elitism_ratio=0.2

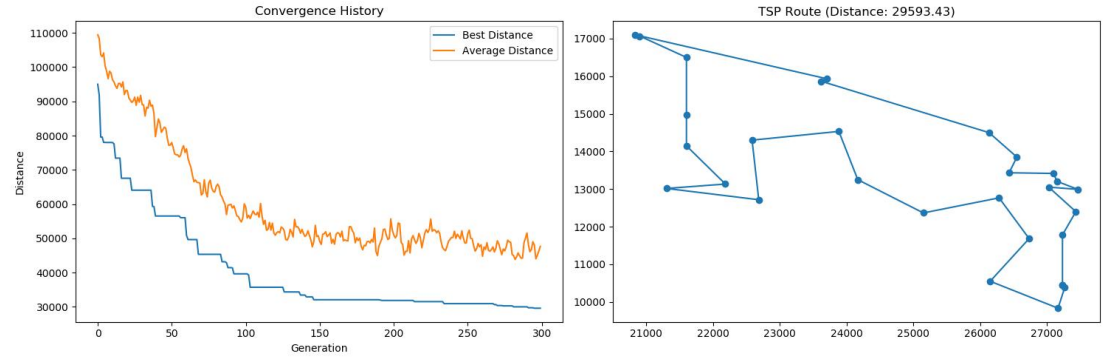


	A	B	C	D	E	F	G	H	I
1	avg_distance	avg_time	elitism_ra...	max_distance	min_distance	mutation_r...	num_generati...	pop_size	std_distance
2	10692.953153922736	0.7022186120351156	0.1	11359.30158313968	9551.588616089259	0.01	300	50	810.8377438980236
3	8888.00627599808	2.37923526763916	0.1	9346.723891567053	8260.22090384739	0.02	500	100	459.3909847596197
4	7457.286978828506	5.167866150538127	0.2	7658.692464458275	7329.735334960925	0.05	800	150	144.09254137315088

样例二：西撒哈拉

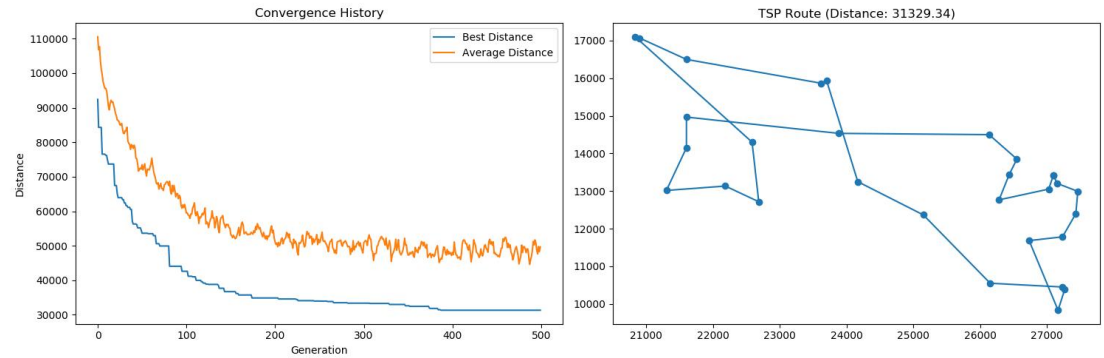
pop_size=50, generations=300, mutation_rate=0.01, elitism_ratio=0.1

```
==========================================
处理数据集: wi29.tsp
==========================================
尝试读取文件: c:\Users\86157\Desktop\AI\LAB\Code\lab4_search\TSP\tsp_data\wi29.tsp
成功读取了 29 个城市坐标

开始参数组合: pop_size=50, generations=300, mutation_rate=0.01, elitism_ratio=0.1
运行 1/3...
运行 1 完成 - 距离: 29593.43, 时间: 0.72秒

==== TSP 求解结果 ====
城市数量: 29
最优路线: [24, 25, 27, 20, 28, 22, 21, 17, 18, 14, 9, 10, 0, 1, 5, 4, 3, 6, 2, 8, 7, 11, 12, 13, 16, 19, 15, 23, 26]
路线距离: 29593.43
求解时间: 0.72 秒
迭代次数: 300
====================

运行 2/3...
运行 2 完成 - 距离: 33268.62, 时间: 0.80秒
运行 3/3...
运行 3 完成 - 距离: 31725.09, 时间: 0.73秒
结果:
  最小距离: 29593.43
  最大距离: 33268.62
  平均距离: 31529.05 ± 1506.78
  平均时间: 0.75秒
```



pop_size=100, generations=500, mutation_rate=0.02, elitism_ratio=0.1

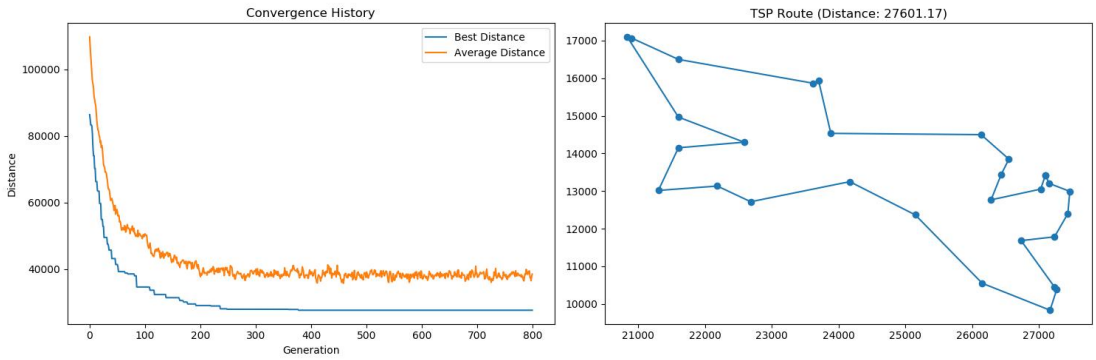
```
开始参数组合: pop_size=100, generations=500, mutation_rate=0.02, elitism_ratio=0.1
运行 1/3...
运行 1 完成 - 距离: 31329.34, 时间: 2.47秒

==== TSP 求解结果 ====
城市数量: 29
最优路线: [21, 20, 16, 17, 18, 14, 11, 4, 3, 2, 6, 8, 7, 0, 1, 5, 9, 10, 12, 13, 15, 24, 26, 23, 19, 25, 27, 28, 22]
路线距离: 31329.34
求解时间: 2.47 秒
迭代次数: 500
====================

运行 2/3...
运行 2 完成 - 距离: 30491.18, 时间: 2.66秒
运行 3/3...
运行 3 完成 - 距离: 29722.33, 时间: 2.41秒
结果:
  最小距离: 29722.33
  最大距离: 31329.34
  平均距离: 30514.28 ± 656.26
  平均时间: 2.51秒
```



pop_size=150, generations=800, mutation_rate=0.05, elitism_ratio=0.2

```
开始参数组合: pop_size=150, generations=800, mutation_rate=0.05, elitism_ratio=0.2
运行 1/3...
运行 1 完成 - 距离: 27601.17, 时间: 5.48秒

==== TSP 求解结果 ====
城市数量: 29
最优路线: [21, 20, 16, 17, 18, 14, 11, 10, 9, 5, 1, 0, 4, 7, 3, 2, 6, 8, 12, 13, 15, 23, 26, 24, 19, 25, 27, 28, 22]
路线距离: 27601.17
求解时间: 5.48 秒
迭代次数: 800
====================

运行 2/3...
运行 2 完成 - 距离: 27601.17, 时间: 5.70秒
运行 3/3...
运行 3 完成 - 距离: 27748.71, 时间: 5.79秒
结果:
  最小距离: 27601.17
  最大距离: 27748.71
  平均距离: 27650.35 ± 69.55
  平均时间: 5.66秒
```

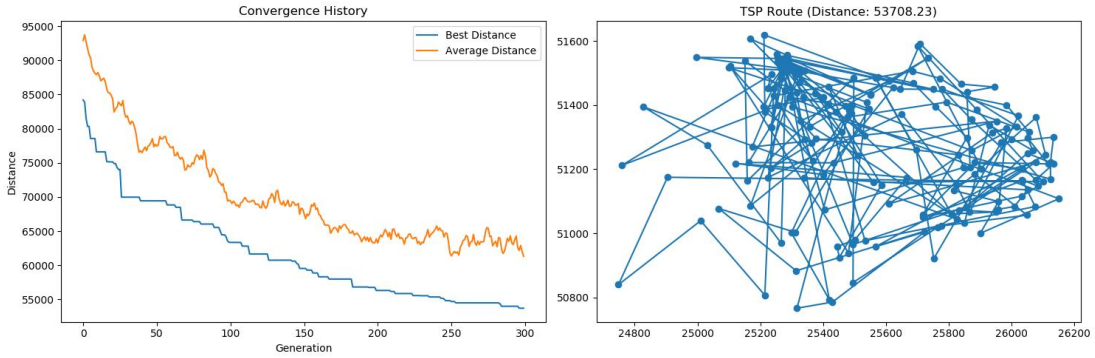
	A	B	C	D	E	F	G	H	I
1	avg_distance	avg_time	elitism_ra...	max_distance	min_distance	mutation_r...	num_generati...	pop_size	std_distance
2	31529.047964452027	0.7495288848876953	0.1	33268.62386641106	29593.425094055616	0.01	300	50	1506.784062046462
3	30514.284399818363	2.51289431254069	0.1	31329.33663120793	29722.333441558363	0.02	500	100	656.2596358663372
4	27650.352375707454	5.65921934445699	0.2	27748.709578134854	27601.17377449375	0.05	800	150	69.54904481495414



样例三：卡塔尔

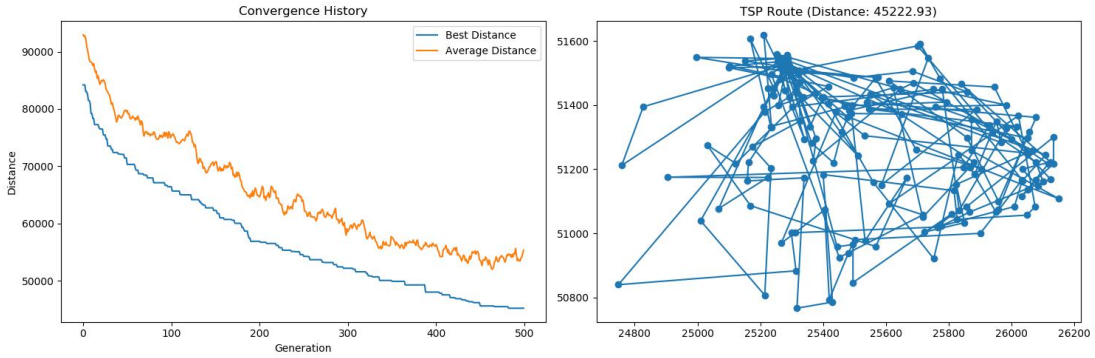
pop_size=50, generations=300, mutation_rate=0.01, elitism_ratio=0.1

```
==========================================
处理数据集: qa194.tsp
==========================================
尝试读取文件: c:\Users\86157\Desktop\AI\LAB\Code\lab4_search\TSP\tsp_data\qa194.tsp
成功读取了 194 个城市坐标

开始参数组合: pop_size=50, generations=300, mutation_rate=0.01, elitism_ratio=0.1
运行 1/3...
运行 1 完成 - 距离: 53708.23, 时间: 8.14秒

==== TSP 求解结果 ====
城市数量: 194
最优路线: [140, 77, 68, 114, 48, 65, 50, 104, 93, 9, 75, 27, 119, 132, 169, 151, 96, 58, 84, 41, 39, 108, 173, 124, 171, 185, 172, 181, 103, 80,
57, 66, 15, 90, 76, 87, 112, 89, 88, 100, 118, 10, 79, 115, 33, 98, 145, 176, 191, 182, 150, 134, 170, 148, 138, 117, 22, 55, 37, 29, 101, 83,
91, 56, 193, 155, 163, 137, 156, 122, 154, 186, 3, 0, 5, 19, 28, 36, 30, 63, 71, 24, 74, 20, 18, 111, 128, 116, 123, 142, 159, 147, 149, 125,
146, 130, 120, 49, 14, 38, 59, 189, 190, 192, 164, 129, 121, 109, 136, 160, 141, 174, 152, 188, 177, 180, 157, 144, 32, 1, 26, 21, 53, 95, 102,
92, 135, 179, 165, 43, 69, 73, 127, 106, 31, 82, 139, 2, 6, 35, 44, 11, 12, 99, 97, 131, 183, 113, 158, 166, 162, 168, 178, 70, 94, 64, 85, 126,
110, 187, 184, 167, 153, 175, 133, 62, 7, 61, 105, 67, 51, 72, 13, 86, 161, 16, 52, 45, 17, 78, 8, 60, 46, 4, 34, 42, 54, 107, 47, 25, 23, 40,
81, 143]
路线距离: 53708.23
求解时间: 8.14 秒
迭代次数: 300
====================

运行 2/3...
运行 2 完成 - 距离: 53848.25, 时间: 8.30秒
运行 3/3...
运行 3 完成 - 距离: 53907.55, 时间: 7.93秒
结果:
  最小距离: 53708.23
  最大距离: 53907.55
  平均距离: 53821.34 ± 83.57
  平均时间: 8.12秒
```



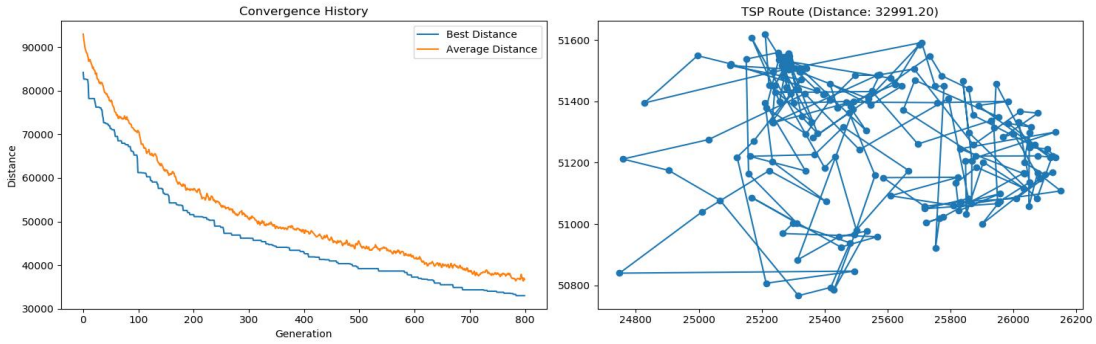
pop_size=100, generations=500, mutation_rate=0.02, elitism_ratio=0.1

```
开始参数组合: pop_size=100, generations=500, mutation_rate=0.02, elitism_ratio=0.1
运行 1/3...
运行 1 完成 - 距离: 45222.93, 时间: 27.61秒

==== TSP 求解结果 ====
城市数量: 194
最优路线: [182, 178, 171, 188, 167, 120, 116, 75, 77, 53, 9, 83, 59, 128, 165, 130, 157, 143, 3, 22, 7, 17, 47, 73, 80, 147, 119, 99, 11, 48,
65, 101, 63, 20, 0, 62, 58, 131, 136, 97, 98, 88, 51, 32, 76, 36, 8, 57, 82, 108, 129, 140, 149, 160, 162, 179, 177, 78, 96, 23, 14, 28, 21, 72,
42, 2, 1, 41, 49, 37, 50, 111, 66, 68, 16, 24, 19, 5, 55, 45, 26, 89, 125, 113, 170, 181, 175, 139, 137, 79, 85, 64, 70, 12, 13, 25, 18, 30, 54,
27, 43, 187, 173, 168, 133, 148, 189, 142, 190, 174, 185, 186, 183, 158, 122, 60, 92, 90, 86, 74, 46, 52, 91, 135, 29, 44, 71, 39, 69, 10, 6,
15, 110, 118, 103, 95, 117, 40, 56, 67, 31, 34, 4, 38, 33, 102, 176, 159, 114, 146, 163, 184, 104, 106, 115, 124, 152, 107, 164, 161, 153, 156,
121, 109, 94, 134, 141, 166, 169, 105, 154, 112, 127, 123, 132, 172, 155, 100, 93, 84, 81, 61, 35, 87, 193, 151, 150, 180, 138, 144, 126, 145,
192, 191]
路线距离: 45222.93
求解时间: 27.61 秒
迭代次数: 500
====================

运行 2/3...
运行 2 完成 - 距离: 46382.85, 时间: 27.98秒
运行 3/3...
运行 3 完成 - 距离: 43068.56, 时间: 32.13秒
结果:
  最小距离: 43068.56
  最大距离: 46382.85
  平均距离: 44891.44 ± 1373.20
  平均时间: 29.24秒
```



pop_size=150, generations=800, mutation_rate=0.05, elitism_ratio=0.2

```
开始参数组合: pop_size=150, generations=800, mutation_rate=0.05, elitism_ratio=0.2
运行 1/3...
运行 1 完成 - 距离: 32991.20, 时间: 66.78秒

==== TSP 求解结果 ====
城市数量: 194
最优路线: [141, 125, 112, 138, 113, 136, 163, 139, 137, 153, 171, 156, 144, 143, 192, 140, 173, 172, 133, 126, 131, 129, 134, 128, 123, 26, 2,
4, 9, 80, 78, 76, 40, 38, 8, 67, 23, 28, 54, 52, 77, 68, 49, 43, 55, 57, 29, 11, 12, 61, 15, 89, 110, 35, 58, 93, 84, 64, 7, 3, 1, 6, 69, 101,
135, 145, 124, 160, 158, 159, 176, 187, 191, 183, 152, 117, 120, 161, 157, 121, 107, 111, 119, 151, 170, 184, 164, 167, 185, 155, 162, 178, 169,
179, 166, 180, 188, 154, 165, 130, 91, 102, 66, 42, 53, 37, 47, 60, 65, 104, 122, 115, 87, 83, 105, 116, 114, 30, 20, 71, 82, 99, 109, 25, 96,
59, 39, 36, 50, 41, 44, 34, 51, 46, 31, 18, 21, 73, 33, 48, 32, 14, 79, 86, 100, 118, 90, 75, 13, 70, 17, 24, 81, 22, 5, 0, 97, 19, 10, 16, 56,
72, 45, 63, 27, 106, 94, 95, 62, 88, 103, 85, 98, 108, 92, 74, 127, 132, 147, 150, 190, 181, 146, 142, 149, 148, 193, 182, 175, 177, 174, 186,
189, 168]
路线距离: 32991.20
求解时间: 66.78 秒
迭代次数: 800
====================

运行 2/3...
运行 2 完成 - 距离: 32719.80, 时间: 76.80秒
运行 3/3...
运行 3 完成 - 距离: 32841.26, 时间: 73.64秒
结果:
  最小距离: 32719.80
  最大距离: 32991.20
  平均距离: 32850.75 ± 111.00
  平均时间: 72.41秒
```



3.

	A	B	C	D	E	F	G	H	I
1	avg_distance	avg_time	elitism_ra...	max_distance	min_distance	mutation_r...	num_generati...	pop_size	std_distance
2	53821.34319248827	8.122824112574259	0.1	53907.54918752869	53708.225975682675	0.01	300	50	83.56871858067174
3	44891.44380874141	29.239842017491657	0.1	46382.846819371676	43068.55956146053	0.02	500	100	1373.2042491138757
4	32850.75319159663	72.40616369247437	0.2	32991.20115925292	32719.801921390397	0.05	800	150	111.00158264617528

2. 评测

从以上结果可知

pop_size=50, generations=300, mutation_rate=0.01, elitism_ratio=0.1

pop_size=100, generations=500, mutation_rate=0.02, elitism_ratio=0.1

pop_size=150, generations=800, mutation_rate=0.05, elitism_ratio=0.2

从上到下

收敛速度：越来越慢慢

解的质量：越来越接近最优解

计算时间：越来越长

四、 参考资料

（1）部分 debug，优化建议参考了 ai 大模型的建议。

（2）阅读参考了以下文章：

TSP 优化：遗传算法实战教程-CSDN 博客

基于遗传算法的 TSP 问题优化（超详细）-CSDN 博客

遗传算法（GA）：C++实现 TSP 问题 - 知乎