



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 | 23336266 | 姓名 | 熊彦钧 |
|----|----------|----|-----|

一、实验题目

- 1.利用启发式搜索解决 15puzzle 问题;
- 2.利用遗传算法求解 TSP 问题。

二、实验内容

一：利用启发式搜索解决 15puzzle 问题

1. 算法原理

启发式搜索是一种利用启发式信息来指导搜索方向的算法。相比于盲目搜索，启发式搜索能更高效地找到解。常见的启发式搜索算法包括 A 算法、IDA 算法等。

其中 A* (A-Star) 算法是一种经典的启发式搜索算法，用于在加权图或状态空间中找到从起点到目标的最优路径。它结合了 Dijkstra 算法和贪心最佳优先搜索的优点，通过引入启发式函数来指导搜索方向，从而显著减少需要探索的节点数量。

2. 关键代码展示

在实验中，我们分别使用了以下几个启发式函数来进行测试，并比较他们之间的效率：

①计算曼哈顿距离：我们通过分别计算每次方格移动后状态的曼哈顿距离（即每一个数字当前位置和目标位置的最小水平+垂直步数之和），并选择曼哈顿距离之和最少的那个状态，经过多次迭代后，在有解的情况下必定能得到最后的目标。其中这一部分的启发式函数代码如下：

```
# def calculate_manhattan_distance(board: np.ndarray, target: np.ndarray) -> int:
#     """计算曼哈顿距离"""
#     target_positions = {}
#     for i in range(16):
#         target_positions[target[i]] = i
#
#     distance = 0
#     for i in range(16):
#         tile_value = board[i]
#         if tile_value == 0:
#             continue
#         correct_pos = target_positions[tile_value]
#         current_row, current_col = i // 4, i % 4
#         correct_row, correct_col = correct_pos // 4, correct_pos % 4
#         distance += abs(current_row - correct_row) + abs(current_col - correct_col)
#
#     return distance
```



②计算欧几里得距离：即计算每个数字当前位置和目标位置之间的直线距离（利用两点之间距离公式），并选择距离之和最小的那个状态进行迭代，在有解的情况下，理论上经过多次迭代必定能得到最后的目标。这一部分的启发式函数代码如下：

```
# def calculate_euclidean_distance(board: np.ndarray, target: np.ndarray) -> float:
#     """计算欧几里得距离"""
#     target_positions = {}
#     for i in range(16):
#         target_positions[target[i]] = i
#
#     distance = 0
#     for i in range(16):
#         tile_value = board[i]
#         if tile_value == 0:
#             continue
#         correct_pos = target_positions[tile_value]
#         current_row, current_col = i // 4, i % 4
#         correct_row, correct_col = correct_pos // 4, correct_pos % 4
#         distance += np.sqrt((current_row - correct_row)**2 + (current_col - correct_col)**2)
#
#     return distance
```

③计算带冲突的曼哈顿距离：带冲突的曼哈顿距离是一种改进的启发式方法，它在标准曼哈顿距离的基础上，额外考虑同行或同列的数字冲突（即若两个数字在同一行/列，他们的目标也在同一行/列，但是顺序相反，则构成冲突，步数需要额外增加2）。我们分别计算每种状态的曼哈顿距离之和+2*冲突对数，并选择求和最少的那个状态进行迭代。在有解的情况下，理论上经过多次迭代必定能得到最后的目标。这一部分的启发式函数代码如下：

```
def calculate_manhattan_with_conflicts(current_board, target_board, target_positions):
    """计算带冲突的曼哈顿距离"""
    base_distance = calculate_manhattan_distance(current_board, target_positions)
    conflict_penalty = 0

    for i in range(4):
        for j in range(4):
            idx = i * 4 + j
            tile_value = current_board[idx]
            if tile_value == 0:
                continue

            target_pos = target_positions[tile_value]
            target_row, target_col = target_pos // 4, target_pos % 4

            # 检查行冲突
            if i == target_row:
                for k in range(j+1, 4):
                    idx2 = i * 4 + k
                    tile_value2 = current_board[idx2]
                    if tile_value2 != 0:
                        if tile_value2 < 16: # 防止数组越界
                            target_pos2 = target_positions[tile_value2]
                            target_row2, target_col2 = target_pos2 // 4, target_pos2 % 4
                            if target_row2 == i and target_col2 < target_col:
                                conflict_penalty += 2

            # 检查列冲突
            if j == target_col:
                for k in range(i+1, 4):
                    idx2 = k * 4 + j
                    tile_value2 = current_board[idx2]
                    if tile_value2 != 0:
                        if tile_value2 < 16: # 防止数组越界
                            target_pos2 = target_positions[tile_value2]
                            target_row2, target_col2 = target_pos2 // 4, target_pos2 % 4
                            if target_col2 == j and target_row2 < target_row:
                                conflict_penalty += 2

    return base_distance + conflict_penalty
```

这几段代码将作为启发式函数，嵌入最终的 A* 算法主体内。整个算法的流程大致如下：

首先我们需要找到空白格的位置，并且使用启发式函数，分别求空白格上下左右移动后的状态的 $h(x)$ ，最终选择 $h(x)$ 最小的那个状态进行迭代。由于代码篇幅较长，这里就不作展示了，详细代码请见附件。

3. 创新点&优化

由于在实验过程中，发现了运行时间过长这一问题，询问了身边的同学，一开始的程序运行时间普遍在三个小时以上，因此本人询问了 AI，获得了一些提高运行速度的方法：

① 使用 numba，通过 @jit 装饰器对关键函数（如曼哈顿距离计算）进行即时编译（JIT），将 Python 代码转换为优化的机器码。

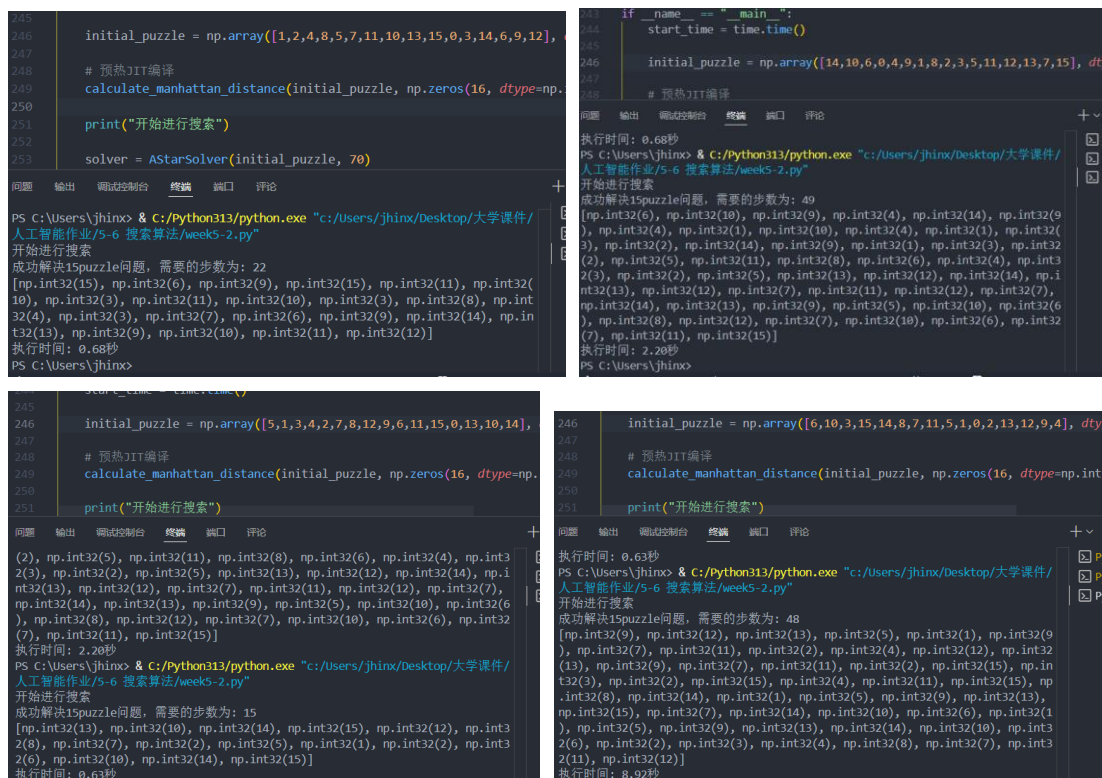
② PuzzleNode 类使用 __slots__ 替代动态属性字典，降低内存开销；

③ 使用哈希存储，并且节点哈希值通过位运算预计算（_calculate_hash），避免每次调用 hash() 时的重复计算。

三、实验结果及分析

1. 实验结果展示示例

对于实验给出的六个初始状态，运行结果分别如下：



```
245 initial_puzzle = np.array([1,2,4,8,5,7,11,10,13,15,0,3,14,6,9,12], dtype=np.int32)
246
247 # 预编译JIT
248 calculate_manhattan_distance(initial_puzzle, np.zeros(16, dtype=np.int32))
249
250 print("开始进行搜索")
251
252 solver = AStarSolver(initial_puzzle, 70)
253
```

PS C:\Users\jinhx> & C:/Python313/python.exe "c:/Users/jinhx/Desktop/大学课件/人工智能作业/5-6 搜索算法/week5-2.py"

开始进行搜索

成功解决15puzzle问题。需要的步数为: 22

[np.int32(15), np.int32(6), np.int32(9), np.int32(15), np.int32(11), np.int32(10), np.int32(3), np.int32(11), np.int32(10), np.int32(3), np.int32(8), np.int32(4), np.int32(3), np.int32(7), np.int32(6), np.int32(9), np.int32(14), np.int32(13), np.int32(9), np.int32(10), np.int32(11), np.int32(12)]

执行时间: 0.68秒

```
245 initial_puzzle = np.array([5,1,3,4,2,7,8,12,9,6,11,15,0,13,10,14], dtype=np.int32)
246
247 # 预编译JIT
248 calculate_manhattan_distance(initial_puzzle, np.zeros(16, dtype=np.int32))
249
250 print("开始进行搜索")
251
```

PS C:\Users\jinhx> & C:/Python313/python.exe "c:/Users/jinhx/Desktop/大学课件/人工智能作业/5-6 搜索算法/week5-2.py"

开始进行搜索

成功解决15puzzle问题。需要的步数为: 15

[np.int32(13), np.int32(10), np.int32(14), np.int32(15), np.int32(12), np.int32(8), np.int32(7), np.int32(2), np.int32(5), np.int32(1), np.int32(2), np.int32(6), np.int32(10), np.int32(14), np.int32(15)]

执行时间: 0.63秒

```
246 initial_puzzle = np.array([6,10,3,15,14,8,7,11,5,1,0,2,13,12,9,4], dtype=np.int32)
247
248 # 预编译JIT
249 calculate_manhattan_distance(initial_puzzle, np.zeros(16, dtype=np.int32))
250
251 print("开始进行搜索")
252
```

PS C:\Users\jinhx> & C:/Python313/python.exe "c:/Users/jinhx/Desktop/大学课件/人工智能作业/5-6 搜索算法/week5-2.py"

开始进行搜索

成功解决15puzzle问题。需要的步数为: 48

[np.int32(9), np.int32(12), np.int32(13), np.int32(5), np.int32(1), np.int32(9), np.int32(7), np.int32(11), np.int32(2), np.int32(4), np.int32(12), np.int32(13), np.int32(9), np.int32(7), np.int32(11), np.int32(2), np.int32(15), np.int32(3), np.int32(2), np.int32(15), np.int32(4), np.int32(11), np.int32(15), np.int32(8), np.int32(14), np.int32(1), np.int32(5), np.int32(9), np.int32(13), np.int32(15), np.int32(7), np.int32(14), np.int32(10), np.int32(6), np.int32(1), np.int32(5), np.int32(9), np.int32(13), np.int32(14), np.int32(10), np.int32(2), np.int32(2), np.int32(3), np.int32(4), np.int32(8), np.int32(7), np.int32(11), np.int32(12)]

执行时间: 8.92秒

```
246 initial_puzzle = np.array([11,3,1,7,4,6,8,2,15,9,10,13,14,12,5,0],
247
248 # 预热JIT编译
249 calculate_manhattan_distance(initial_puzzle, np.zeros(16, dtype=np.
250
251 print("开始进行搜索"))

问题 输出 调试控制台 终端 窗口 评论
PS C:\Users\jhinx> & C:\Python313\python.exe "c:/Users/jhinx/Desktop/大学课件/
人工智能作业/5-6 搜索算法/week5-2.py"
开始进行搜索
成功解决15puzzle问题，需要的步数为：56
[ np.int32(13), np.int32(10), np.int32(8), np.int32(6), np.int32(9), np.int32(1
2), np.int32(5), np.int32(13), np.int32(10), np.int32(8), np.int32(12), np.in
32(15), np.int32(14), np.int32(5), np.int32(13), np.int32(12), np.int32(15), n
p.int32(14), np.int32(5), np.int32(13), np.int32(14), np.int32(9), np.int32(4
), np.int32(11), np.int32(3), np.int32(1), np.int32(6), np.int32(11), np.int3
2(1), np.int32(1), np.int32(6), np.int32(4), np.int32(2), np.int32(8), np.in
32(10), np.int32(12), np.int32(15), np.int32(10), np.int32(8), np.in
32(7), np.int32(4), np.int32(2), np.int32(11), np.int32(3), np.int32(5), np.i
nt32(9), np.int32(10), np.int32(11), np.int32(13), np.int32(6), np.int32(2), np
.int32(3), np.int32(7), np.int32(8), np.int32(12)]
执行时间：41.44秒
PS C:\Users\jhinx>

246 initial_puzzle = np.array([0,5,15,14,7,9,6,13,1,2,12,10,8,11,4,3],
247
248 # 预热JIT编译
249 calculate_manhattan_distance(initial_puzzle, np.zeros(16, dtype=np
250
251 print("开始进行搜索"))

问题 输出 调试控制台 终端 窗口 评论
人工智能作业/5-6 搜索算法/week5-2.py"
开始进行搜索
成功解决15puzzle问题，需要的步数为：62
[ np.int32(7), np.int32(9), np.int32(2), np.int32(1), np.int32(9), np.int32(2),
np.int32(5), np.int32(7), np.int32(2), np.int32(5), np.int32(1), np.int32(11)
, np.int32(8), np.int32(9), np.int32(5), np.int32(1), np.int32(6), np.int32(12
), np.int32(10), np.int32(3), np.int32(4), np.int32(8), np.int32(11), np.int3
2(10), np.int32(12), np.int32(13), np.int32(3), np.int32(4), np.int32(8), np.in
32(12), np.int32(13), np.int32(15), np.int32(14), np.int32(3), np.int32(4), n
p.int32(9), np.int32(12), np.int32(13), np.int32(15), np.int32(14), np.int32(7
), np.int32(2), np.int32(1), np.int32(5), np.int32(10), np.int32(11), np.int3
2(13), np.int32(15), np.int32(14), np.int32(7), np.int32(3), np.int32(4), np.in
32(8), np.int32(12), np.int32(15), np.int32(14), np.int32(11), np.int32(10),
np.int32(9), np.int32(13), np.int32(14), np.int32(15)]
执行时间：91.06秒
PS C:\Users\jhinx>
```

经过运行测试，代码能够解决 15puzzle 问题，实验完成。

2. 评测指标展示及分析

由上面的运行结果可以看到，即使是这里面最复杂的第六个情况，也能在 100 秒以内得出答案，相比优化前动辄几小时的运行时间，代码效率得到了显著提升。

二、利用遗传算法求解 TSP 问题

二、实验内容

1. 算法原理

遗传算法是一种模拟生物进化过程的优化算法，其核心原理是通过模拟自然选择、交叉（重组）和变异等机制，逐步优化种群中的解。

选择操作包括：轮盘赌选择（按适应度比例随机选择）、锦标赛选择（随机选择几个个体，保留最优者）、精英选择（直接保留当前最优个体到下一代）等，其目的是选择适应度高的个体作为父代。

交叉操作包括：部分映射交叉（随机选择两个交叉点，交换父代的片段）、顺序交叉（保留父代 A 的一个子序列，按父代 B 的顺序填充剩余城市）、循环交叉（通过循环继承父代的基因）等。其目的是通过重组父代基因生成子代。

变异操作包括：交换变异（随机交换两个个体）、倒置变异（反转一段子序列）等，其目的是以一定概率（通常较小，如 0.010.01）随机改变个体，增加多样性。

算法流程：

1. 初始化种群。
2. 计算每个个体的适应度。

While 不满足终止条件 do:

选择父代。

通过交叉生成子代。

对子代进行变异。

更新种群（保留精英或替换部分个体）。

3. 输出历史最优解



2.关键代码展示

①数据加载和初始化:

```
def load_tsp_data(file_path):
    """读取TSP问题数据文件,提取城市坐标"""
    header_line = 0
    # 查找城市坐标数据的起始位置
    with open(file_path, 'r') as file:
        for i, line in enumerate(file):
            if line.startswith('NODE_COORD_SECTION'):
                header_line = i + 1

    # 读取数据并转换为numpy数组
    df = pd.read_csv(file_path, skiprows=header_line, header=None, sep=' ')
    df = df.dropna()
    cities_data = df.to_numpy()
    for i in range(len(cities_data)):
        cities_data[i][1] = float(cities_data[i][1])
        cities_data[i][2] = float(cities_data[i][2])
    cities_data = cities_data[:, 1:]
    return cities_data
```

②计算适应度: 计算路径的总距离, 并转换为适应度(距离越短, 适应度越高):

```
def calculate_fitness(cities_data, route):
    """计算路径的适应度值"""
    # 根据路径顺序重新排列城市坐标
    ordered_cities = np.array([cities_data[route[i]] for i in range(len(route))])
    total_distance = 0.0
    for i in range(len(route) - 1):
        total_distance += np.linalg.norm(ordered_cities[i] - ordered_cities[i+1])
    total_distance += np.linalg.norm(ordered_cities[-1] - ordered_cities[0])
    fitness = 1 / (total_distance + 1e-6)
    return fitness
```

③选择操作: 这里采用的是锦标赛选择:

```
def tournament_selection(solutions_population, tournament_size):
    """锦标赛选择法:从种群中随机选择几个个体, 然后选出其中最好的一个"""
    selected_solutions = np.random.choice(solutions_population, size=tournament_size, replace=False)
    selected_solutions = np.array([TspSolution(solution.route, solution.fitness)
                                   for i, solution in enumerate(selected_solutions)],
                                   dtype=object)
    return select_best_solution(selected_solutions)
```

④交叉操作: 这里采用的是顺序交叉:

```
def order_crossover(parent1_route, parent2_route):
    """顺序交叉算子:在保持路径有效性的前提下, 结合两个父代路径的特征"""
    i = np.random.randint(0, len(parent1_route))
    j = np.random.randint(0, len(parent1_route))
    while i == j:
        j = np.random.randint(0, len(parent1_route))
    if i > j:
        i, j = j, i # 确保i是起点, j是终点

    city_mapping = {}
    for k in range(i, j+1):
        city_mapping[parent2_route[k]] = parent1_route[k]

    child_route = np.zeros((len(parent1_route),), dtype=int)
    for k in range(len(parent1_route)):
        if k < i or k > j:
            child_route[k] = parent1_route[k]
        else:
            child_route[k] = city_mapping[parent2_route[k]]
    return child_route
```



⑤变异操作：这里采用的是倒置变异：

```
def apply_inversion_mutation(route, mutation_rate=0.1):
    """倒置变异:以一定概率随机选择路径的一段，将其倒置"""
    for i in range(len(route)):
        if np.random.rand() < mutation_rate:
            # 随机选择两个点
            i = np.random.randint(0, len(route))
            j = np.random.randint(0, len(route))
            while i == j:
                j = np.random.randint(0, len(route))
            if i > j:
                i, j = j, i
            route[i:j+1] = route[i:j+1][::-1]
    return route
```

其中对于整个种群的变异率，我们需要编写一个函数来自适应变异率并且定期重置部分种群以防止种群提前收敛，代码如下：

```
def calculate_adaptive_mutation_rate(iteration, max_iterations):
    """自适应变异率计算:随着迭代进行，变异率逐渐降低，同时定期提高变异率以跳出局部最优"""
    initial_rate = 0.3
    final_rate = 0.001

    alpha = 4.0 # 退火参数
    progress = iteration / max_iterations

    # 每500代提高一次变异率
    if iteration % 500 == 0 and iteration > 0:
        return initial_rate * 0.5
    return final_rate + (initial_rate - final_rate) * np.exp(-alpha * progress)
```

```
def perform_partial_reset(solutions_population, cities_data, reset_percentage=0.3):
    """部分种群重置:淘汰一部分较差的解，引入新的随机解增加多样性"""
    population_size = len(solutions_population)
    reset_count = int(population_size * reset_percentage)

    indices = np.argsort([solution.fitness for solution in solutions_population])
    keep_indices = indices[-(population_size - reset_count):]

    new_routes = np.array([np.random.permutation(len(cities_data)) for _ in range(reset_count)])
    new_solutions = np.array([TspSolution(route, calculate_fitness(cities_data, route))
                              for route in new_routes], dtype=object)

    # 组合保留的解和新生成的解
    new_population = np.zeros(population_size, dtype=object)
    new_population[:reset_count] = new_solutions
    new_population[reset_count:] = solutions_population[keep_indices]

    return new_population
```

⑥遗传算法函数主体：由于篇幅较长，请详见附件

三、实验结果及分析

1. 实验结果展示示例

这里我们在网址 <https://www.math.uwaterloo.ca/tsp/world/countries.html> 上选择了两个较小的数据集（dj38 以及 wi29）。他们的测试结果分别如下：

（1）dj38 数据集的遗传算法求解结果如下：



```
PS C:\Users\jhinx> & C:/Python313/python.exe "c:/Users/jhinx/Desktop/大学课件/人工智能作业/5-6 搜索算法/week5-4.py"
第 0 代: 最佳路径长度 = 22288.018819868754
第 1000 代: 最佳路径长度 = 13958.401751613674
第 2000 代: 最佳路径长度 = 10782.505817246252
第 3000 代: 最佳路径长度 = 7807.425782937127
第 4000 代: 最佳路径长度 = 7243.392221184194
第 5000 代: 最佳路径长度 = 6659.90674138676
第 6000 代: 最佳路径长度 = 6659.906741386758
第 7000 代: 最佳路径长度 = 6659.906741386758
第 8000 代: 最佳路径长度 = 6659.4315339314635
第 9000 代: 最佳路径长度 = 6659.4315339314635
PS C:\Users\jhinx>
```

(2) wi29 数据集的遗传算法求解结果如下:

```
PS C:\Users\jhinx> & C:/Python313/python.exe "c:/Users/jhinx/Desktop/大学课件/人工智能作业/5-6 搜索算法/week5-4.py"
第 0 代: 最佳路径长度 = 86701.48125951276
第 0 代: 最佳路径长度 = 86701.48125951276
第 1000 代: 最佳路径长度 = 38213.59887591963
第 1000 代: 最佳路径长度 = 38213.59887591963
第 2000 代: 最佳路径长度 = 30448.96400566136
第 3000 代: 最佳路径长度 = 28506.642268875352
第 4000 代: 最佳路径长度 = 28506.642268875345
第 5000 代: 最佳路径长度 = 27601.17377549376
第 6000 代: 最佳路径长度 = 27601.17377549375
第 7000 代: 最佳路径长度 = 27601.17377549375
第 8000 代: 最佳路径长度 = 27601.17377549375
第 9000 代: 最佳路径长度 = 27601.17377549375
第 9000 代: 最佳路径长度 = 27601.17377549375
```

经过结果展示, 算法正确性得到保障。

2. 评测指标展示及分析

由于前两个数据集样本量较小, 无法得出比较普遍性的结论, 因此在这里我们额外测试一个数据集 (qa194), 在保持参数不变的情况下, 输出如下:

```
PS C:\Users\jhinx> & C:/Python313/python.exe "c:/Users/jhinx/Desktop/大学课件/人工智能作业/5-6 搜索算法/week5-4.py"
第 0 代: 最佳路径长度 = 84376.46388462752
第 1000 代: 最佳路径长度 = 69407.20038947402
第 2000 代: 最佳路径长度 = 64608.83591805124
第 3000 代: 最佳路径长度 = 59960.874893066924
第 4000 代: 最佳路径长度 = 52753.09013642583
第 5000 代: 最佳路径长度 = 46010.94411372778
第 6000 代: 最佳路径长度 = 39006.3012052602
第 7000 代: 最佳路径长度 = 31205.599406816982
第 8000 代: 最佳路径长度 = 23516.6501573091
第 9000 代: 最佳路径长度 = 17259.001658363573
```

我们发现在程序结束的时候 (即达到了最大的迭代次数), 最佳路径长度存在未收敛的可能性, 因此我们上调迭代次数的限制为 20000 代, 重新测试, 结果如下:

```
PS C:\Users\jhinx> & C:/Python313/python.exe "c:/Users/jhinx/Desktop/大学课件/人工智能作业/5-6 搜索算法/week5-4.py"
第 0 代: 最佳路径长度 = 86100.65866228208
第 1000 代: 最佳路径长度 = 72273.84228201667
第 2000 代: 最佳路径长度 = 68543.7984653188
第 3000 代: 最佳路径长度 = 65917.045208271
第 4000 代: 最佳路径长度 = 63294.69556560092
第 5000 代: 最佳路径长度 = 61558.33957585631
第 6000 代: 最佳路径长度 = 58806.683956291556
第 7000 代: 最佳路径长度 = 54709.62085301513
第 8000 代: 最佳路径长度 = 51315.05669759572
第 9000 代: 最佳路径长度 = 46765.17204222025
第 10000 代: 最佳路径长度 = 42190.758488621235
第 11000 代: 最佳路径长度 = 38562.09391758941
第 12000 代: 最佳路径长度 = 34892.96950393223
第 13000 代: 最佳路径长度 = 29580.98153959938
第 14000 代: 最佳路径长度 = 25552.389213271817
第 15000 代: 最佳路径长度 = 21753.510058346314
第 16000 代: 最佳路径长度 = 18953.98382456202
第 17000 代: 最佳路径长度 = 15746.030995767007
第 18000 代: 最佳路径长度 = 14466.953186955676
第 19000 代: 最佳路径长度 = 12840.16128408778
```

可以看到依然没有出现前面两个数据集类似的收敛情况。于是我们再次上调为 30000 代,



重新测试，结果如下：

```
PS C:\Users\jhinx> & C:/Python313/python.exe "c:/Users/jhinx/Desktop/大学课件/人工智能作业/5-6 搜索算法/week5-4.py"
第 0 代: 最佳路径长度 = 86552.30932904998
第 1000 代: 最佳路径长度 = 70405.73255337961
第 2000 代: 最佳路径长度 = 69343.79205373324
第 3000 代: 最佳路径长度 = 67645.754987828
第 4000 代: 最佳路径长度 = 66688.18077853088
第 5000 代: 最佳路径长度 = 66031.42513644625
第 6000 代: 最佳路径长度 = 63603.70106083046
第 7000 代: 最佳路径长度 = 61390.04598278003
第 8000 代: 最佳路径长度 = 60401.8327702317
第 9000 代: 最佳路径长度 = 57200.656926355834
第 10000 代: 最佳路径长度 = 55076.81958216458
第 11000 代: 最佳路径长度 = 52379.78034389655
第 12000 代: 最佳路径长度 = 50234.51756007046
第 13000 代: 最佳路径长度 = 47588.78881673044
第 14000 代: 最佳路径长度 = 44323.95157808337
第 15000 代: 最佳路径长度 = 40196.967729594304
第 16000 代: 最佳路径长度 = 38288.64938231885
第 17000 代: 最佳路径长度 = 34851.98417094656
第 18000 代: 最佳路径长度 = 32709.75465770301
第 19000 代: 最佳路径长度 = 29867.345166175695
第 20000 代: 最佳路径长度 = 26285.59020195449
第 21000 代: 最佳路径长度 = 23731.842370446688
第 22000 代: 最佳路径长度 = 20536.65655533532
第 23000 代: 最佳路径长度 = 18186.304339336715
第 24000 代: 最佳路径长度 = 16573.64100025483
第 25000 代: 最佳路径长度 = 13977.252595582839
第 26000 代: 最佳路径长度 = 12887.591603845203
第 27000 代: 最佳路径长度 = 11927.283656986425
第 28000 代: 最佳路径长度 = 11365.668537617488
第 29000 代: 最佳路径长度 = 11054.71937251703
第 30000 代: 最佳路径长度 = 10700.248177259919
```

可以看到，所得的结果接近于收敛。

从上面对于同一数据集的三次输出结果我们可以发现：由于变异和交换等操作具有随机性，因此在迭代相同次数的时候，输出的结果不尽相同，并且在数据集基数较大的时候，遗传算法可能存在不收敛的情况。

四、 参考资料

详细代码请见附件