

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

一、 实验题目

遗传算法解决 TSP 问题。

二、 实验内容

1. 算法原理

遗传算法的原理，总得来说，就是把问题的所有可能的解单独看成一个个体，总体上看成一个种群，我们想要的解的性质决定了这个种群的自然选择的方向。这样，我们就可以模拟大自然对这个种群进行自然选择，淘汰不符合要求的个体，选择接近要求的个体。同时模拟种群基因的交叉和变异，不断得到新的个体，经过很多代的种群繁衍，从而不断得到接近要求的问题的解。这总结有可能是近似解，也可能是某一范围内的极值对应的解，也就是不那么精确，但是在调整种群数量，种群的繁衍代数，以及变异概率等参数后，多次模拟，最终是可以得到问题的最佳解的。

旅行商问题，就是给定一定数量的点的坐标，求得一条最短路径把这些点串连起来，这个问题在城市数比较多的时候，可能的解也是成指数增长，利用传统的算法效率就很低了，而且难以得到比较精确的解，这个时候遗传算法就是一个比较好的选择，可以在相对较高的效率下，得到比较精确的解。

2. 关键代码展示（可选）

使用 `python` 实现遗传算法求解旅行商问题的具体方法如下：

- (1) 首先，定义一个算法类，在类中完成算法的所有实现和问题的求解。算法需要导入 `numpy`, `math`, `matplotlib.pyplot` 三个库。`numpy` 的作用是使用里面的 `numpy` 数组比使用 `list` 的效率要更高一些，`matplotlib.pyplot` 用于绘制路线图和收敛曲线。

```
1 import numpy as np
2 import math
3 import matplotlib.pyplot as plt
```



图 1 导入相应库

- (2) 定义全局变量，方便修改算法参数。

```

5   # 初始种群数
6   init_population = 200
7
8   # 变异概率
9   mutation_pro = 0.05
10
11  # 迭代次数
12  iterate_num = 20000
13

```

图 2 定义全局变量

- (3) 定义算法类的构造函数, 用于读取.tsp 文件获取城市信息, 并计算得到两两城市的距离, 初始化种群, 并得到当前初始种群的每条路线的适应度和当前最佳路径数组和长度。适应度是长度的倒数, 适应度越高, 长度越短, 越符合我们的需求。但是值得注意的是, 网站上相关.tsp 文件有的以 EOF 结尾, 有的却没有, 如果使用我下面给出的代码需要把.tsp 文件的 EOF 行删掉, 不留任何多余的行, 或者在我下面的代码的第 28 行 if 条件加上 line != 'EOF'。

```

15  # 解决TSP问题的遗传算法类
16  class GeneticAlgTSP:
17      def __init__(self, filename):
18          # 读取文件数据
19          with open(filename, 'r') as file:
20              lines = file.readlines()
21              coordinate_index = 0
22              i = 0
23              for line in lines:
24                  if line.startswith("DIMENSION"):
25                      self.dimension = int(line.split(":")[1]) # 城市数量n
26                      self.city_coordinates = np.zeros((self.dimension, 2)) # 城市坐标, n×2矩阵, 分别是横坐标和纵坐标
27                      self.city_distances = np.zeros((self.dimension, self.dimension)) # 城市之间的距离, n×n矩阵
28                  if coordinate_index == 1:
29                      city_features = line.strip().split()
30                      self.city_coordinates[i][0] = float(city_features[1]) # 城市横坐标
31                      self.city_coordinates[i][1] = float(city_features[2]) # 城市纵坐标
32                      i += 1
33                  if line.startswith("NODE_COORD_SECTION"):
34                      coordinate_index = 1
35          # 计算得到两两城市之间的距离, 存入city_distances二维数组中
36          for i in range(self.dimension):
37              for j in range(self.dimension):
38                  x = self.city_coordinates[i][0] - self.city_coordinates[j][0]
39                  y = self.city_coordinates[i][1] - self.city_coordinates[j][1]
40                  distance = math.sqrt(x * x + y * y)
41                  self.city_distances[i][j] = distance
42          # 随机初始化种群
43          self.population = np.zeros((init_population, self.dimension)).astype(int)
44          for i in range(len(self.population)):
45              self.population[i] = np.random.choice(range(self.dimension), size=self.dimension, replace=False)
46          # 得到初始种群每个个体(路径)的适应度, 并计算得到当前最佳路径best_route
47          self.get_all_fitness()
48          best_index = np.argmax(self.fitness)
49          self.best_route = self.population[best_index] # 目前已知最佳路径
50

```

图 3 构造函数

- (4) 接下来在类中定义方便计算的小函数, 包括某一条路径的长度计算, 某一条路径的适应度的值 (长度的倒数) 以及种群中每条路径的适应度的值。



```
51      # 计算某一条路径的长度
52  def get_one_distance(self, route):
53      length = 0
54      for i in range(len(route) - 1):
55          length += self.city_distances[route[i]][route[i + 1]]
56      length += self.city_distances[route[-1]][route[0]]
57      return length
58
59      # 计算某一条路径的适应度的值（长度的倒数）
60  def get_one_fitness(self, route):
61      length = 0
62      for i in range(len(route) - 1):
63          length += self.city_distances[route[i]][route[i + 1]]
64      length += self.city_distances[route[-1]][route[0]]
65      return 1 / length
66
67      # 计算种群中每条路径的适应度的值
68  def get_all_fitness(self):
69      self.fitness = np.zeros(len(self.population)) # 种群此时每条路径的适应度的值
70      for i in range(len(self.population)):
71          self.fitness[i] = self.get_one_fitness(self.population[i])
72      return self.fitness
73
```

图 4 计算函数

- (5) 然后定义遗传算法中最关键的选择，遗传，变异的过程函数，我使用的选择方式是轮盘赌选择的方法，以种群中每个个体的适应度占总适应度（种群中每个个体的适应度之和）的比例为概率挑选个体加入新种群。具体实现方法是得到 `probability` 数组作为每个个体的选择概率，依照这个概率数组调用 `np` 中的 `choice` 方法得到使用相应概率随机产生的数组索引，那么这个索引代表的种群个体就成功地选择出来了。循环得到新的种群。

```
74      # 对初始种群进行选择
75  def select(self):
76      # 定义选择后的种群
77      selected_population = np.zeros((len(self.population), self.dimension)).astype(int)
78      # 将每条路径的适应度在总适应度之和的占比作为选择的概率
79      probability = self.fitness / np.sum(self.fitness)
80      for i in range(len(self.population)):
81          # 依照概率选择种群中的一条路线作为新种群的一条路线
82          choice = np.random.choice(range(len(self.population)), p=probability)
83          selected_population[i] = self.population[choice]
84      return selected_population
85
```

图 5 种群选择函数

- (6) 下面实现交叉函数，这应该是算法中最难实现的函数了，而且至关重要，实测不同的交叉函数对算法的性能有决定性的影响。我最开始写的交叉函数，比如最短路径长度是 3w 多的城市群，在算法运行过程中，从最开始的 8w 长度迭代到了 5.8w 后就死活难以再减小了，明显这样的交叉算法是完全不行的。修改后就能非常接近标准答案了，如果不计算的误差，就是正确答案了。总而言之交叉函数是算法中最重要的函数，下面给出我实现的交叉函数：



```
86     # 对某两条路径进行交叉
87     def crossover(self, route1, route2):
88         # 随机选择一个断点
89         point = np.random.randint( low: 1, self.dimension - 1)
90         # 深拷贝原来的路径
91         child1 = np.copy(route1)
92         child2 = np.copy(route2)
93         j = k = 0
94         # 交叉得到新路径
95         for i in range(self.dimension):
96             if route2[i] not in child1[:point]:
97                 child1[point + j] = route2[i]
98                 j += 1
99             if route1[i] not in child2[:point]:
100                 child2[point + k] = route1[i]
101                 k += 1
102         # 将新路径替换原来的路径
103         route1 = child1
104         route2 = child2
105
106     # 对种群中所有个体进行两两交叉
107     def crossover_all(self):
108         for i in range(0, self.population.shape[0] - 1, 2):
109             self.crossover(self.population[i], self.population[i + 1])
110
```

图 6 种群交叉函数

- (7) 然后是种群进化的最后一步，变异，我按照 ppt 上的变异方式，随机选则一条路径的一个片段进行倒置。

```
111     # 对种群进行变异（倒置变异）
112     def mutation(self):
113         # 得到决定是否变异的随机数数组
114         pro_array = np.random.rand(self.population.shape[0])
115         for i in range(self.population.shape[0]):
116             # 在一定的变异概率下
117             if pro_array[i] <= mutation_pro:
118                 # 随机选取变异片段
119                 point1 = np.random.randint( low: 0, self.dimension)
120                 point2 = np.random.randint(point1 + 1, self.dimension + 1)
121                 old_seq = self.population[i][point1:point2]
122                 # 将变异片段倒置
123                 new_seq = old_seq[::-1]
124                 self.population[i][point1:point2] = new_seq
125
```

图 7 种群变异函数

- (8) 然后是算法的主函数，迭代函数，我写了很多版本，一是没有画图输出的，二是最终会画出最佳路线图版本的，三是每迭代 200 次就会更新最佳路线图的，四是在三的基础上，最终会画出最佳路线长度随迭代次数的变化曲线的。下面展示第四版的代码实现：
(迭代本身不复杂，但是输出图形占了很多行代码)



```
由 Xnip 截图

126     def iterate(self, num_iterations, print_best_route=False):
127         plt.ion() # 开启交互模式
128         best_fitness_over_time = [] # 初始化存储最佳适应度值的列表
129
130         for i in range(num_iterations):
131             # 依次进行遗传算法中选择，交叉，变异，得到适应度（路线长度），更新最佳路径的步骤
132             self.population = self.select()
133             self.crossover_all()
134             self.mutation()
135             self.get_all_fitness()
136             best_index = np.argmax(self.fitness)
137             new_best_route = self.population[best_index] # 目前已知最佳路径
138             fitness_old = self.get_one_fitness(self.best_route)
139             fitness_new = self.fitness[best_index]
140             if fitness_old < fitness_new:
141                 self.best_route = new_best_route
142             best_fitness_over_time.append(1 / self.fitness[best_index]) # 存储最佳适应度值的倒数（如果适应度是路线长度）
143
144             if i % 200 == 0: # 每200次迭代更新一次图形
145                 print('迭代次数: ', "{:5d}".format(i), ' ', end='')
146                 print('最短路径值: ', self.get_one_distance(self.best_route))
147                 if print_best_route:
148                     print(self.best_route)
149                     self.plot_best_route() # 调用绘制最佳路径的方法
150                     plt.pause(0.1) # 暂停一段时间，以便图形更新
151                     plt.clf() # 清除当前图形，准备下一次绘制
152
153         plt.ioff() # 关闭交互模式
154         # 输出最终得到的最佳路径城市列表和最短路径值
155         final = self.get_one_fitness(self.best_route)
156         print(list(self.best_route))
157         print(1 / final)
158         self.plot_best_route() # 最后再绘制一次，确保最终结果被显示
159
160         # 绘制适应度曲线
161         plt.figure() # 新建一个图形
162         plt.plot(*args: best_fitness_over_time, label='Best Fitness Over Time')
163         plt.xlabel('Iteration')
164         plt.ylabel('Fitness')
165         plt.title('Fitness Over Time')
166         plt.legend()
167         plt.show() # 显示图形
168
169         # 使用matplotlib绘制最佳路径函数
170     def plot_best_route(self):
171         plt.figure(figsize=(10, 6))
172         # 绘制所有城市的位置
173         plt.scatter(self.city_coordinates[:, 0], self.city_coordinates[:, 1], c='red', label='Cities')
174         # 绘制最佳路径
175         for i in range(-1, len(self.best_route) - 1):
176             start_city = self.city_coordinates[self.best_route[i]]
177             end_city = self.city_coordinates[self.best_route[i + 1]]
178             plt.plot(*args: [start_city[0], end_city[0]], [start_city[1], end_city[1]], 'k-')
179         plt.title('Best Route: ' + str(1 / self.get_one_fitness(self.best_route)))
180         plt.xlabel('X Coordinate')
181         plt.ylabel('Y Coordinate')
182         plt.legend()
183         plt.show()
184
```

图 8 迭代函数

(9) 接下来写出文件的主函数部分，就可以运行代码了。

```
186 if __name__ == '__main__':
187     a = GeneticAlgTSP('wi29.tsp')
188     a.iterate(iterate_num)
189     a.plot_best_route()
190
```

图 9 主函数

3. 创新点&优化（如果有）

- (1) 在类的实现中，我只定义了类的实例变量 `fitness` 数组来存储种群中每个个体的适应度，后来想了一下其实没有必要强调适应度这个概念，直接定义 `length` 数组来存储种群中每条路径的长度就行了，在比较的时候也可以直接比较长度，根本用不着什么适应度，而且我计算适应度是直接把长度的倒数当适应度，而表示长度的时候又是取适应度的倒数，这样会增大计算的误差。所以可以直接从类中剔除适应度这个概念。让代码瘦身，也提高了精确度。
- (2) 我最开始没有加路线图以及收敛曲线的图形显示代码，只有终端输出的数值，显得很抽象，加入了路线图的显示，遗传算法的运行过程显得更加的清晰，容易理解。
- (3) 为了加快代码运行的效率，之前每次都是用 `len(self.population)` 来代表种群个体数，改成直接用全局变量 `init_population` 能更快。

Ps: 上述所有的优化都在最终的 `main.py` 下完成，所以最终上交的代码文件和上面的关键代码展示有区别。

三、 实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）

- (1) Western Sahara 的 29 个城市

在种群数量为 200，变异概率为 0.05，迭代次数为 20000 的情况下，最终结果如下：



The screenshot shows the PyCharm IDE interface. The project is named 'AI_lab5' and contains files: d38.tsp, main.py, q94.tsp, and w129.tsp. The 'main.py' file is open, showing Python code for a Genetic Algorithm TSP solver. The terminal window below shows the iterative process of finding the best route, starting from an initial value of 17400 and converging to 27748.70957813485 after 190 iterations. The code uses a genetic algorithm to iterate through cities and plot the best route.

```
for i in range(-1, len(self.best_route) - 1):
    start_city = self.city_coordinates[self.best_route[i]]
    end_city = self.city_coordinates[self.best_route[i + 1]]
    plt.plot(*args [start_city[0], end_city[0]], [start_city[1], end_city[1]], 'k-')
plt.title('Best Route: ' + str(self.get_one_distance(self.best_route)))
plt.xlabel('X Coordinate')
plt.ylabel('Y Coordinate')
plt.legend()
plt.show()

if __name__ == '__main__':
    a = GeneticAlgTSP('w129.tsp')
    a.iterate(iterate_num)
    a.plot_best_route()
```

```
迭代次数: 17400 最短路径值: 27748.70957813485
迭代次数: 17600 最短路径值: 27748.70957813485
迭代次数: 17800 最短路径值: 27748.70957813485
迭代次数: 18000 最短路径值: 27748.70957813485
迭代次数: 18200 最短路径值: 27748.70957813485
迭代次数: 18400 最短路径值: 27748.70957813485
迭代次数: 18600 最短路径值: 27748.70957813485
迭代次数: 18800 最短路径值: 27748.70957813485
迭代次数: 19000 最短路径值: 27748.70957813485
迭代次数: 19200 最短路径值: 27748.70957813485
迭代次数: 19400 最短路径值: 27748.70957813485
迭代次数: 19600 最短路径值: 27748.70957813485
迭代次数: 19800 最短路径值: 27748.70957813485
[18, 17, 21, 22, 26, 28, 27, 25, 24, 26, 23, 15, 19, 16, 13, 12, 8, 6, 2, 3, 7, 4, 8, 1, 5, 9, 10, 11, 14]
27748.70957813485
```

图 10 29 个城市迭代过程

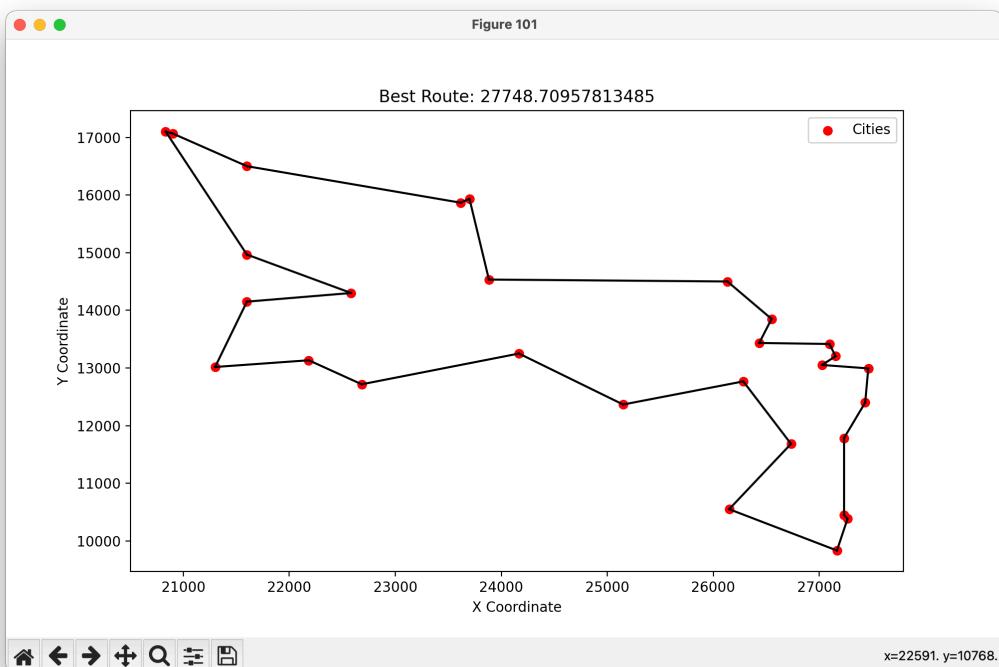


图 11 29 个城市路线图

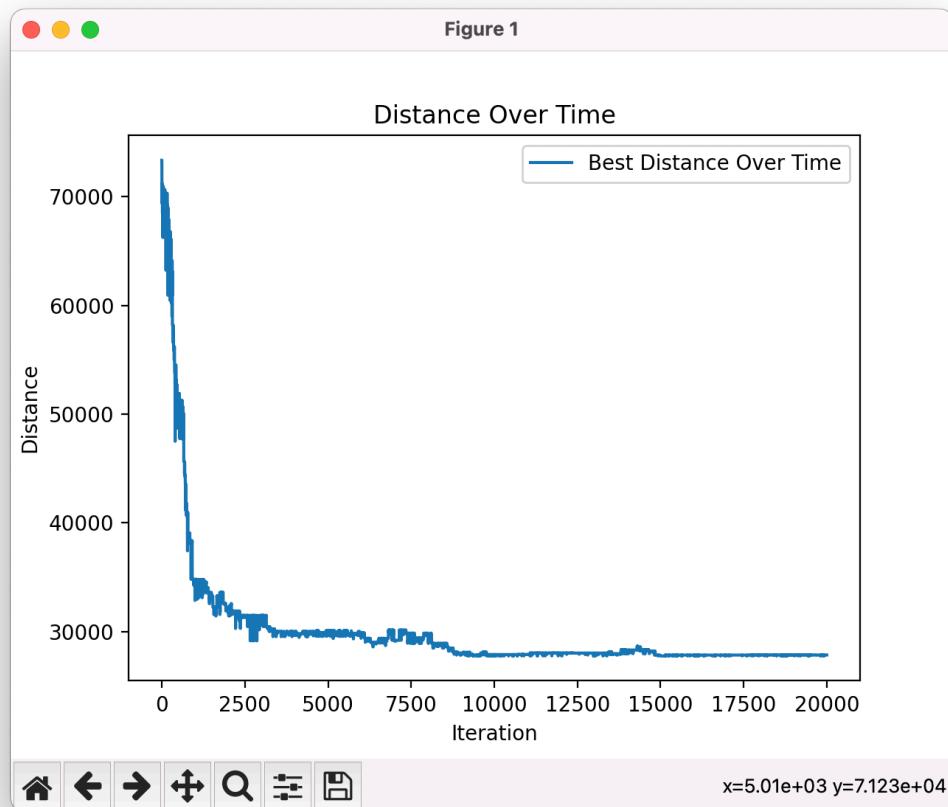
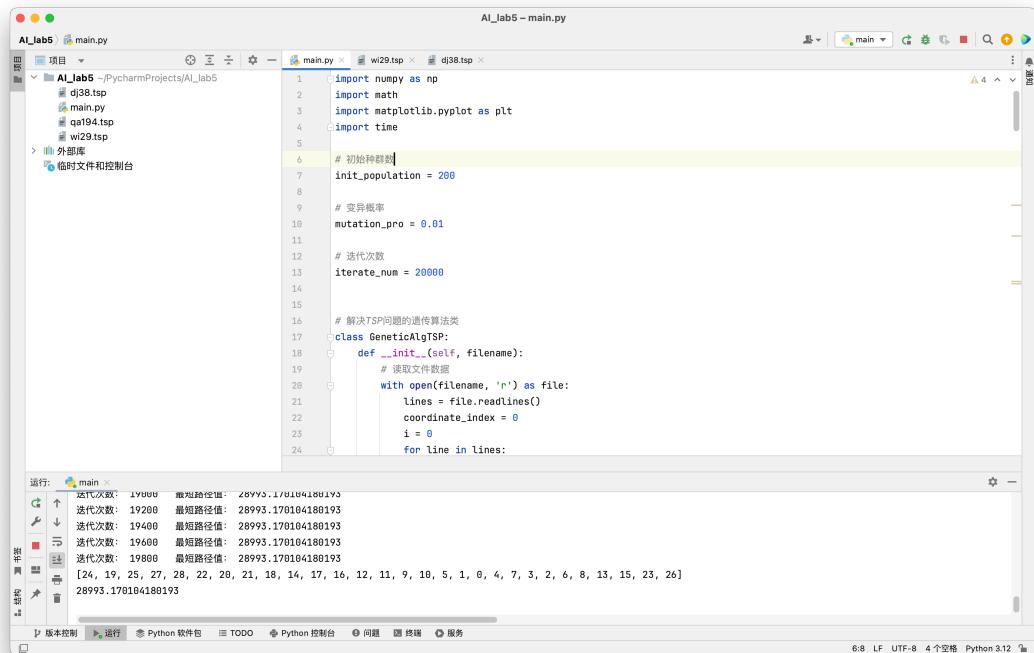


图 12 29 个城市迭代收敛曲线

如果修改种群变异概率为 0.01，结果如下：



```

AI_Lab5 ->PycharmProjects/AI_Lab5
  |- dj38.tsp
  |- main.py
  |- q94.tsp
  |- w19.tsp
>|- 外部库
  |- 临时文件和控制台

AI_Lab5 ->main.py
1  import numpy as np
2  import math
3  import matplotlib.pyplot as plt
4  import time
5
6  # 初始种群数
7  init_population = 200
8
9  # 变异概率
10 mutation_pro = 0.01
11
12 # 迭代次数
13 iterate_num = 20000
14
15
16 # 解决TSP问题的遗传算法类
17 class GeneticAlgTSP:
18     def __init__(self, filename):
19         # 读取文件数据
20         with open(filename, 'r') as file:
21             lines = file.readlines()
22             coordinate_index = 0
23             i = 0
24             for line in lines:

```

运行: main <--> 运行 Python 软件包 TODO Python 控制台 问题 终端 服务

迭代次数: 19000 最短路径值: 28993.170104180193
 迭代次数: 19200 最短路径值: 28993.170104180193
 迭代次数: 19400 最短路径值: 28993.170104180193
 迭代次数: 19600 最短路径值: 28993.170104180193
 迭代次数: 19800 最短路径值: 28993.170104180193
 [24, 19, 25, 27, 28, 22, 29, 21, 18, 14, 17, 16, 12, 11, 9, 10, 5, 1, 0, 4, 7, 3, 2, 6, 8, 13, 15, 23, 26]
 28993.170104180193

图 13 修改变异概率后的迭代结果

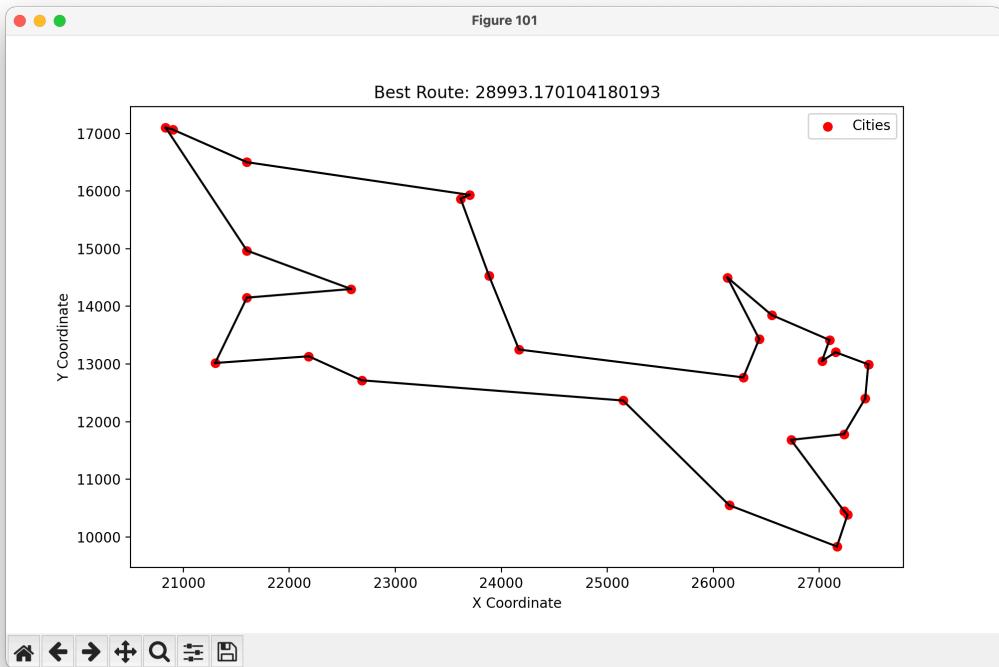
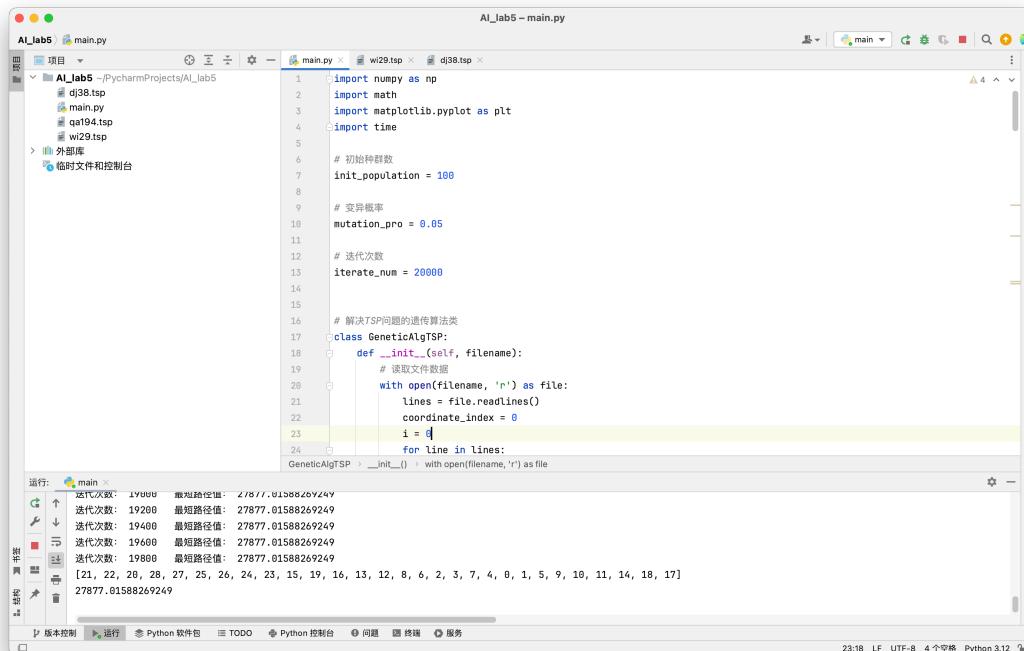


图 14 修改变异概率后的路线图

可见路线有细微差别，明显 0.05 的变异概率要更优。

若修改初始种群数为 100，结果如下：



```

AI_Lab5 - main.py
-----
1  import numpy as np
2  import math
3  import matplotlib.pyplot as plt
4  import time
5
6  # 初始种群数
7  init_population = 100
8
9  # 变异概率
10 mutation_pro = 0.05
11
12 # 迭代次数
13 iterate_num = 20000
14
15
16 # 解决TSP问题的遗传算法类
17 class GeneticAlgorithmTSP:
18     def __init__(self, filename):
19         # 读取文件数据
20         with open(filename, 'r') as file:
21             lines = file.readlines()
22             coordinate_index = 0
23             i = 0
24             for line in lines:
25                 GeneticAlgorithmTSP > __init__() > with open(filename, 'r') as file

```

图 15 修改初始种群数后的迭代结果

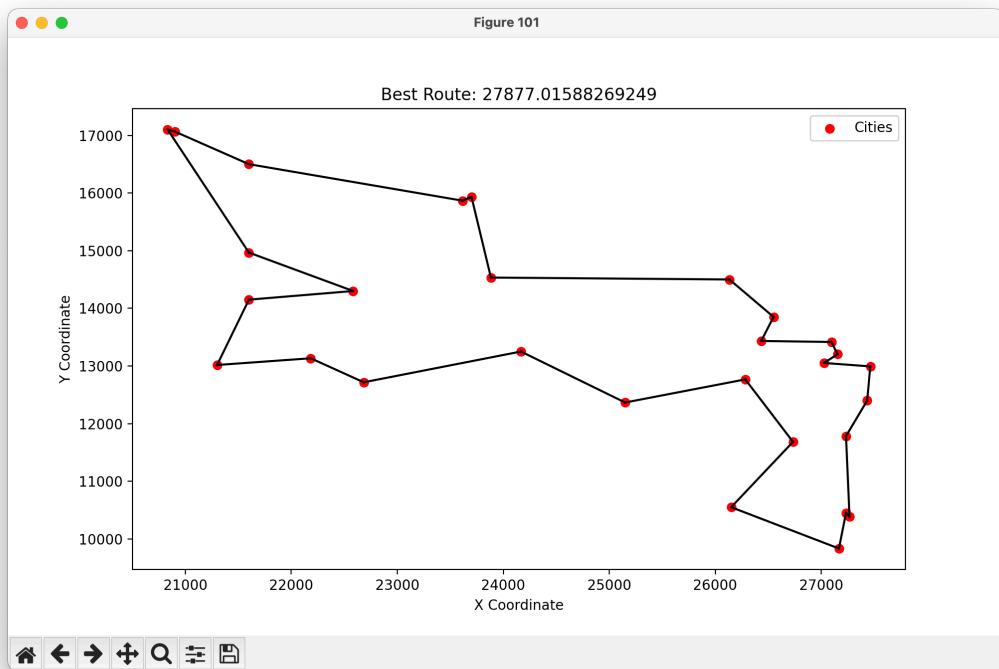


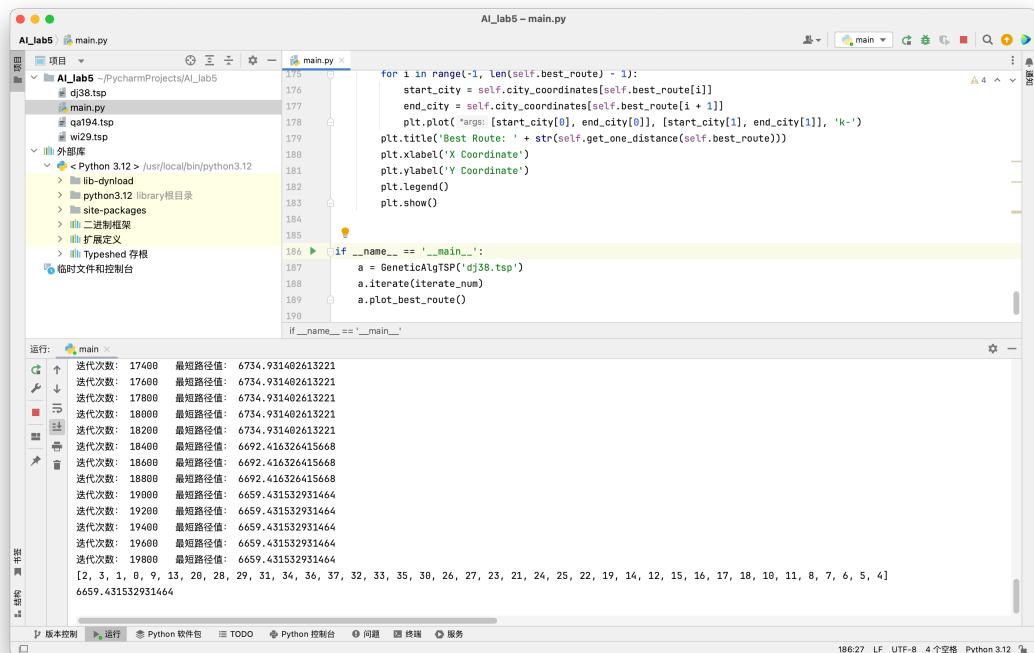
图 16 修改初始种群数后的路线图

可见，初始种群数修改后，程序运行更快了，但是结果也不如修改前。

经过我的反复试验，大概种群数为 200，变异概率为 0.05 为最优，当然，这也与城市数

量有关，不同的 TSP 问题有不同的最佳参数。

(2) Djibouti 的 38 个城市



The screenshot shows the PyCharm IDE interface with the project structure and code editor. The code in main.py is as follows:

```

for i in range(-1, len(self.best_route) - 1):
    start_city = self.city_coordinates[self.best_route[i]]
    end_city = self.city_coordinates[self.best_route[i + 1]]
    plt.plot([start_city[0], end_city[0]], [start_city[1], end_city[1]], 'k-')
plt.title('Best Route: ' + str(self.get_one_distance(self.best_route)))
plt.xlabel('X Coordinate')
plt.ylabel('Y Coordinate')
plt.legend()
plt.show()

if __name__ == '__main__':
    a = GeneticAlgTSP('dj38.tsp')
    a.iterate(iterate_num)
    a.plot_best_route()

```

The '运行' (Run) tool window at the bottom shows the iterative process of the genetic algorithm, starting from an initial route length of 17400 and gradually improving it to a minimum of 19600, eventually reaching a best route length of 6659.431532931464.

图 17 38 个城市迭代过程

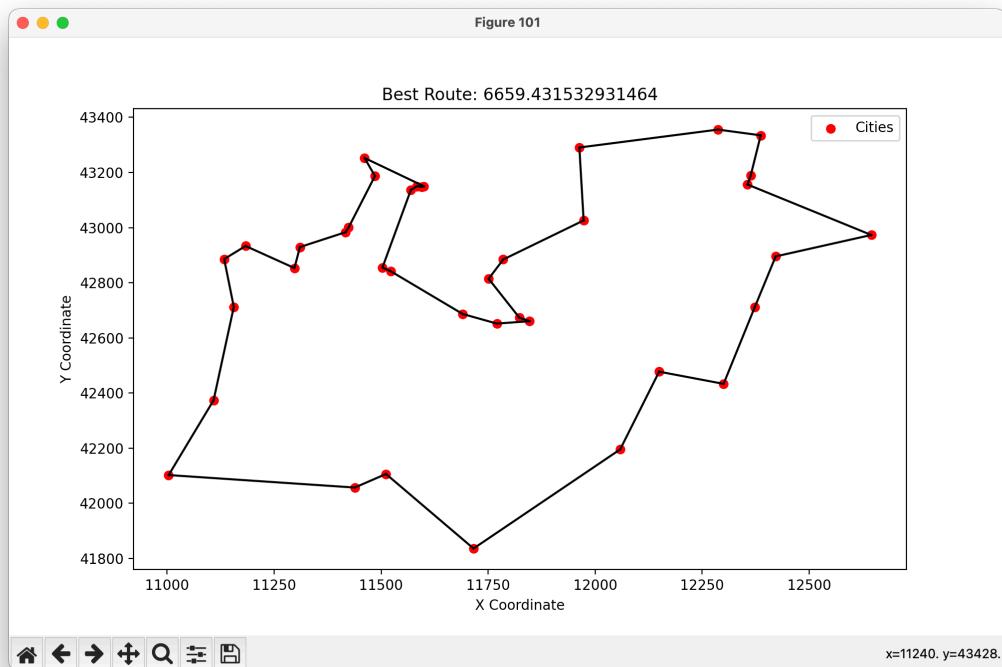


图 18 38 个城市路线图

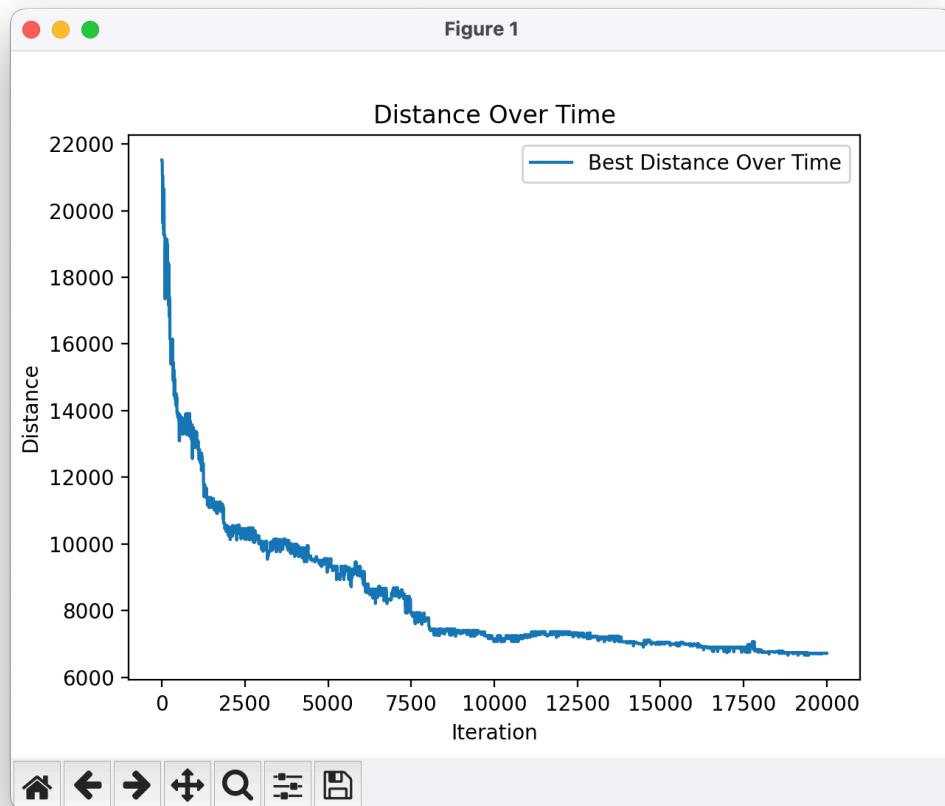


图 19 38 个城市收敛曲线

2. 评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）

对于我的遗传算法，下面对算法的主要操作时间复杂度的评估。

- (1) 选择操作 (`select` 方法): 这个操作的时间复杂度主要取决于种群大小。`(init_population)` 因为它遍历整个种群来计算适应度和选择新种群，所以其时间复杂度大致为 $O(N)$ ，其中 N 是种群的大小。
- (2) 交叉操作 (`crossover_all` 方法): 这个操作遍历种群中的每对个体进行交叉，因此其时间复杂度也是 $O(N)$ ， N 是种群的大小。
- (3) 变异操作 (`mutation` 方法): 这个操作同样遍历整个种群，对每个个体进行可能的变异操作，其时间复杂度也是 $O(N)$ 。

(4) 迭代操作 (`iterate` 方法): 这是遗传算法的主体, 它包含了上述所有操作, 并且会重复执行指定的迭代次数 (`iterate_num`)。因此, 如果上述每个操作的时间复杂度是 $O(N)$, 迭代操作的总时间复杂度将是 $O(N M)$, 其中 M 是迭代次数。

添加测量时间的代码, 种群 200, 20000 次迭代, 29 个城市的运行时间如下: (中途要关掉所有的图片才能使代码继续运行, 实际更快, 大概 300 整秒。体验上大概 3 秒左右完成 200 次迭代。



```

运行: main
迭代次数: 18000 最短路径值: 27601.173774493753
迭代次数: 18200 最短路径值: 27601.173774493753
迭代次数: 18400 最短路径值: 27601.173774493753
迭代次数: 18600 最短路径值: 27601.173774493753
迭代次数: 18800 最短路径值: 27601.173774493753
迭代次数: 19000 最短路径值: 27601.173774493753
迭代次数: 19200 最短路径值: 27601.173774493753
迭代次数: 19400 最短路径值: 27601.173774493753
迭代次数: 19600 最短路径值: 27601.173774493753
迭代次数: 19800 最短路径值: 27601.173774493753
[3, 7, 4, 0, 1, 5, 9, 10, 11, 14, 18, 17, 16, 20, 21, 22, 28, 27, 25, 19, 24, 26, 23, 15, 13, 12, 8, 6, 2]
27601.173774493753
运行时间: 359.784452753067 秒
进程已结束, 退出代码为 0

```

图 20 添加运行时间测量

|-----如有优化, 请重复 1, 2, 分析优化后的算法结果-----|

四、参考资料

将参考的代码, 算法思路的来源表明在此处(可以是网址, 参考文献等)

- [1] 学者网: 5-高级搜索+.pptx, 实验作业 5.pdf, ch4 高级搜索.pdf
- [2] TSP 问题网站: <https://www.math.uwaterloo.ca/tsp/world/countries.html>
- [3] Github 上相关资料: https://github.com/nairoj/tsp_ga/tree/master