实验三　遗传算法

一　实验目的

熟悉和掌握遗传算法的原理和框架，遗传算法的编码和解码，遗传算法的遗传算子；熟悉遗传算法的执行过程，并学会利用遗传算法解决实际应用问题。

二　实验原理

遗传算法是模仿生物遗传学和自然选择机理，通过人工方式构造的一类优化搜索算法，是对生物进化过程进行中的一种仿真，是进化计算的一种重要形式。在用遗传算法求解问题时，问题的每一个可能解都被编码成一个“染色体”，即个体，若干个个体构成了一个群体（所有可行解）。在遗传算法开始时，总是随机产生一些个体（即初始解），根据预定的目标函数对每一个个体进行评估，给出一个适应度值；基于此适应度值，选择一些个体来产生下一代，选择操作体现了“适者生存”的原理；然后选择出来的个体，经过交叉和变异算子进行再组合生成新的一代，这样逐步朝着最优解的方向进化。

三　实验内容

1 给出遗传算法求解 TSP（旅行商问题）的算法步骤

(1) 问题描述

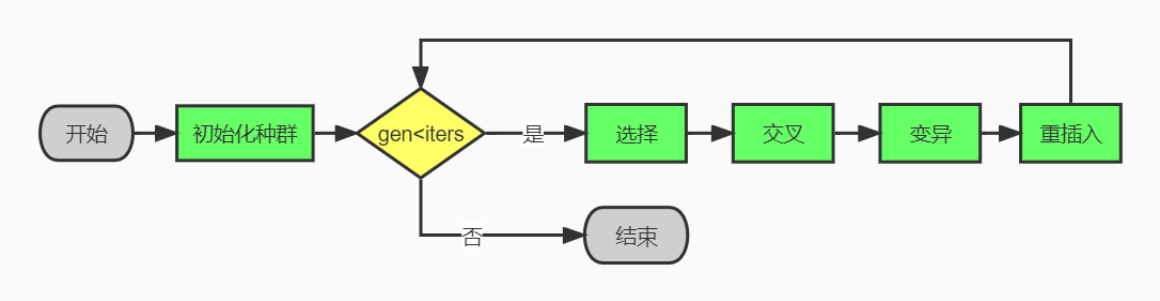
以 14 个城市为例，假定 14 个城市的坐标如下，寻找一条最短的遍历 14 个城市的路径

城市编号	X 坐标	Y 坐标	城市编号	X 坐标	Y 坐标
1	16.47	96.10	8	17.20	96.29
2	16.47	94.44	9	16.30	97.38
3	20.09	92.54	10	14.05	98.12
4	22.39	93.37	11	16.53	97.38
5	25.23	97.24	12	21.52	95.59
6	22.00	96.05	13	19.41	97.13
7	20.47	97.02	14	20.09	92.55

表 1: 14 个城市的位置坐标

(2) 算法流程

通过对初始化的种群进行选择、交叉、变异、重插入操作使得种群进化，从而使得该种群的适应度提高，画出流程图如下：



2 编程实现遗传算法求解 TSP（旅行商问题）

(1) 编码

采用整数编码方式，对于 n 个城市的 TSP 问题，染色体分为 n 段，其中每一段为对应城市的编号，如对 10 个城市的 TSP 问题 $\{1,2,3,4,5,6,7,8,9,10\}$，则 $|1|2|3|4|5|6|7|8|9|10|$ 就是一个合法染色体。

(2) 种群初始化

在完成染色体编码后，必须产生一个初始种群作为起始解，运用随机数初始化一个 $m \times n$ 的整数矩阵，其中每行代表一种路径，初始化 m 条路径。

```
        def initPop(self, NIND, N):
            """
            description: 初始化种群
            param: NIND 种群大小
            param: N 个体染色体长度(城市个数)
            Returns: Chrom 初始种群
            """

            Chrom = np.zeros(shape=(NIND, N), dtype= int)
            for i in  range(NIND):
                Chrom[i] = np.random.permutation(N)

            return Chrom
```

(3) 适应度函数

设 $|k_1|k_2|\cdots|k_n|$ 为一个整数编码的染色体，$D_{k_ik_j}$ 为城市 k_i 到城市 k_j 的距离，则该个体的适应度为

$$fitness = \frac{1}{\sum_{i=1}^{n-1} D_{k_ik_j} + D_{k_nk_1}}$$

即适应度函数为恰好走遍 n 个城市，再回到起点城市的距离的倒数。

```
        def fitness(self, distance):
            """
            description: 适应度函数
            param: distance 个体长度向量(TSP距离)
```

```
        Returns: FitnV 个体适应度向量
        """

        return 1 / distance
```

(4) 选择操作

采用轮赌盘算法从旧种群中以一定的概率选择个体到新种群中，个体被选择到的概率与适应度有关，个体的适应度值越大，被选中的概率越大。

```
    def select(self, Chrom, FitnV, GGAP):
        """
        description: 选择操作
        param: Chrom 种群
        param: FitnV 适应度
        param: GGAP 选择概率
        Returns: SelCh 被选择的个体
        """

        def sus(Nsel, FitnV):
            """
            description:
            param: FitnV 个体适应度向量
            param: Nsel 被选择个体的数目
            Returns: sel_index 选择到的个体索引
            """

            NIND = FitnV.shape[0]
            sel_index = np.zeros(Nsel, dtype= int)          # 保存被选择的个体下标
            cumfit = np.cumsum(FitnV)                       # 累计适应度
            cumfit_one = cumfit / cumfit[-1]                # 映射到[0, 1]
            for i in  range(Nsel):
                rand = np.random.rand()
                for j in  range(NIND):
                    if cumfit_one[j] > rand:
                        sel_index[i] = j
                        break

            return sel_index

        NIND = Chrom.shape[0]                               # 种群个数
        # 被选择的个体数
        Nsel =  int(np. max(np.array([np.ceil(NIND * GGAP), 2])))
        sel_index = sus(Nsel=Nsel, FitnV=FitnV)
        SelCh = Chrom[sel_index, :]

        return SelCh
```

(5) 交叉操作

随机产生两个 $[1,n]$ 中的两个点 r_1，r_2，将 r_1 和 r_2 之间的基因进行交叉，然后在处理冲突。例如 $n=10$、$r_1=4$、$r_2=7$

9	5	1	3	7	4	2	10	8	6
10	5	4	6	3	8	7	2	1	9

交叉后得到

9	5	1	6	3	8	7	10	*	*
10	5	*	3	7	4	2	*	1	9

通过中间部分的映射可以得到

9	5	1	6	3	8	7	10	4	2
10	5	8	3	7	4	2	6	1	9

```
1          def recombin(self, SelCh, Pc):
2              """
3              description: 交叉操作
4              param: SelCh 被选中的个体
5              param: Pc 交叉概率
6              Returns: Selch 交叉后的个体
7              """
8
9              def intercross(A, B):
10                 """
11                 description: 两条染色体交叉
12                 param: A 染色体A
13                 param: B 染色体B
14                 Returns: 交叉后的两条染色体A, B
15                 """
16
17                 L =  len(A)
18                 # 随机生成两个交叉点
19                 r1, r2 = 0, 0
20                 while r1 == r2:
21                     r1 = np.random.randint(low=1, high=L - 1)
22                     r2 = np.random.randint(low=1, high=L - 1)
23                 if r1 > r2:
24                     r1, r2 = r2, r1
25                 # 交叉
26                 for i in  range(r1, r2):
27                     A[i], B[i] = B[i], A[i]
28                 # 解决A冲突
29                 for i in  range(r1):
```

```
                    j = r1
                    while j < r2:
                        if A[i] == A[j]:
                            A[i] = B[j]
                            j = r1
                            continue
                        j += 1
                for i in  range(r2, L):
                    j = r1
                    while j < r2:
                        if A[i] == A[j]:
                            A[i] = B[j]
                            j = r1
                            continue
                        j += 1

                # 解决B冲突
                for i in  range(r1):
                    j = r1
                    while j < r2:
                        if B[i] == B[j]:
                            B[i] = A[j]
                            j = r1
                            continue
                        j += 1
                for i in  range(r2, L):
                    j = r1
                    while j < r2:
                        if B[i] == B[j]:
                            B[i] = A[j]
                            j = r1
                            continue
                        j += 1

                return A, B

            Nsel = SelCh.shape[0]
            for i in  range(Nsel - 1):
                rand = np.random.rand()
                if rand < Pc:
                    SelCh[i, :], SelCh[i + 1,
                                       :] = intercross(SelCh[i, :], SelCh[i + 1, :])

            return SelCh
```

(6) 变异操作

随机选取 $[1, n]$ 中两个点 r_1，r_2，对换他们的位置即可

```
        def mutate(self, SelCh, Pm):
            """
            description: 变异操作
            param: SelCh 被选择的个体
            param: Pm 变异概率
            Returns: SelCh 变异后的个体
            """

            Nsel, L = SelCh.shape
            for i in  range(Nsel):
                rand = np.random.rand()
                if rand < Pm:
                    r1, r2 = 0, 0
                    while r1 == r2:
                        r1 = np.random.randint(low=0, high=L)
                        r2 = np.random.randint(low=0, high=L)
                    if r1 > r2:
                        r1, r2 = r2, r1
                    SelCh[i, r1], SelCh[i, r2] = SelCh[i, r2], SelCh[i, r1]

            return SelCh
```

(7) 进化逆操作

改善遗传算法的局部搜索能力，在选择、交叉、变异之后进行进化逆操作，只有经逆操作后，适应度提高的才接受下来，否则逆操作无效。在 $[1, n]$ 之间随机生成两个点 r_1，r_2，将其对换位置，计算适应度，比原来高，则保留，否则逆操作无效。

```
        def reverse(self, SelCh, Distance):
            """
            description: 进化逆转操作
            param: SelCh 被选择的个体
            param: Distance 距离矩阵
            Returns: SelCh 进化逆转后的个体
            """

            row, L = SelCh.shape
            ObjV = self.path_distance(Chrom=SelCh, Distance=Distance)
            SelCh_1 = SelCh.copy()
            for i in  range(row):
                r1, r2 = 0, 0
                while r1 == r2:
                    r1 = np.random.randint(low=0, high=L)
                    r2 = np.random.randint(low=0, high=L)
                if r1 > r2:
                    r1, r2 = r2, r1
                SelCh_1[i, r1], SelCh_1[i, r2] = SelCh_1[i, r2], SelCh_1[i, r1]
```

```
            ObjV_1 = self.path_distance(Chrom=SelCh_1, Distance=Distance)
            for i in  range(row):
                if ObjV[i] > ObjV_1[i]:
                    SelCh[i, :] = SelCh_1[i, :]

            return SelCh
```

(8) 重插入

将经过选择、交叉、变异、逆操作后的种群重新插入到原来的种群中，但保留父代中的精英。

```
        def reins(self, Chrom, SelCh, ObjV, Distance, opt_path):
            """
            description: 父代精英重插入
            param: Chrom 父代种群
            param: SelCh 子代种群
            param: ObjV 父代适应度
            param: Distance 距离矩阵
            param: opt_path 父代精英
            Returns: SelCh 组合后得到的新种群
            """

            NIND = Chrom.shape[0]
            Nsel = SelCh.shape[0]
            index = np.argsort(ObjV)
            for i in  range(NIND - Nsel):
                SelCh = np.vstack((SelCh, Chrom[index[i]]))
            ObjV_s = self.path_distance(SelCh, Distance)
            # 用父代精英替换最bad的个体
            bad_index = np.argmax(ObjV_s)
            SelCh[bad_index, :] = opt_path[:]
            return SelCh
```

(9) 运行结果

设置一些初始参数如下

```
            if __name__ == "__main__":
                pop_size = 20
                city_size = 14
                position = np.array([[16.47, 96.10],
                                     [16.47, 94.44],
                                     [20.09, 92.54],
                                     [22.39, 93.37],
                                     [25.23, 97.24],
                                     [22.00, 96.05],
                                     [20.47, 97.02],
                                     [17.20, 96.29],
```

```
                                    [16.30, 97.38],
                                    [14.05, 98.12],
                                    [16.53, 97.38],
                                    [21.52, 95.59],
                                    [19.41, 97.13],
                                    [20.09, 92.55]])
    iters = 200
    Pc = 0.7
    Pm = 0.05
    Pa = 0.8
    tsp = TSP(pop_size=pop_size, city_size=city_size,
              position=position, iters=iters, Pc=Pc, Pm=Pm, Pa=Pa)
    tsp.compute_optimal_path()
    print("最优路径: ", tsp.opt_path)
    print("花费代价: ", tsp.preObjV)
    tsp.draw_evolution()
    tsp.draw_distance()
```

得到最终路径为：

$$8 \to 9 \to 0 \to 1 \to 13 \to 2 \to 3 \to 4 \to 5 \to 11 \to 6 \to 12 \to 7 \to 10 \to 8$$

总花费为：29.34052

进化过程图如下：

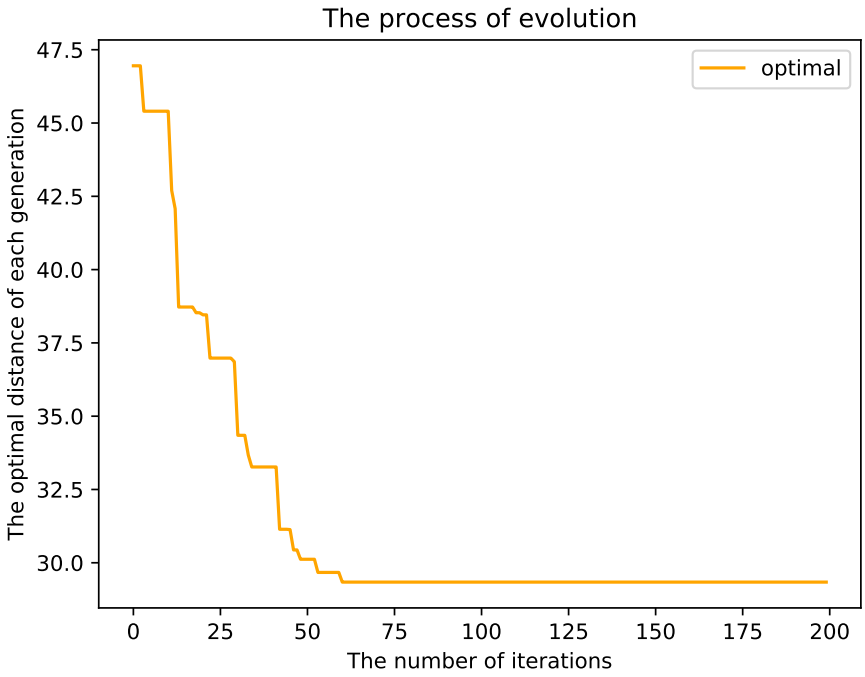
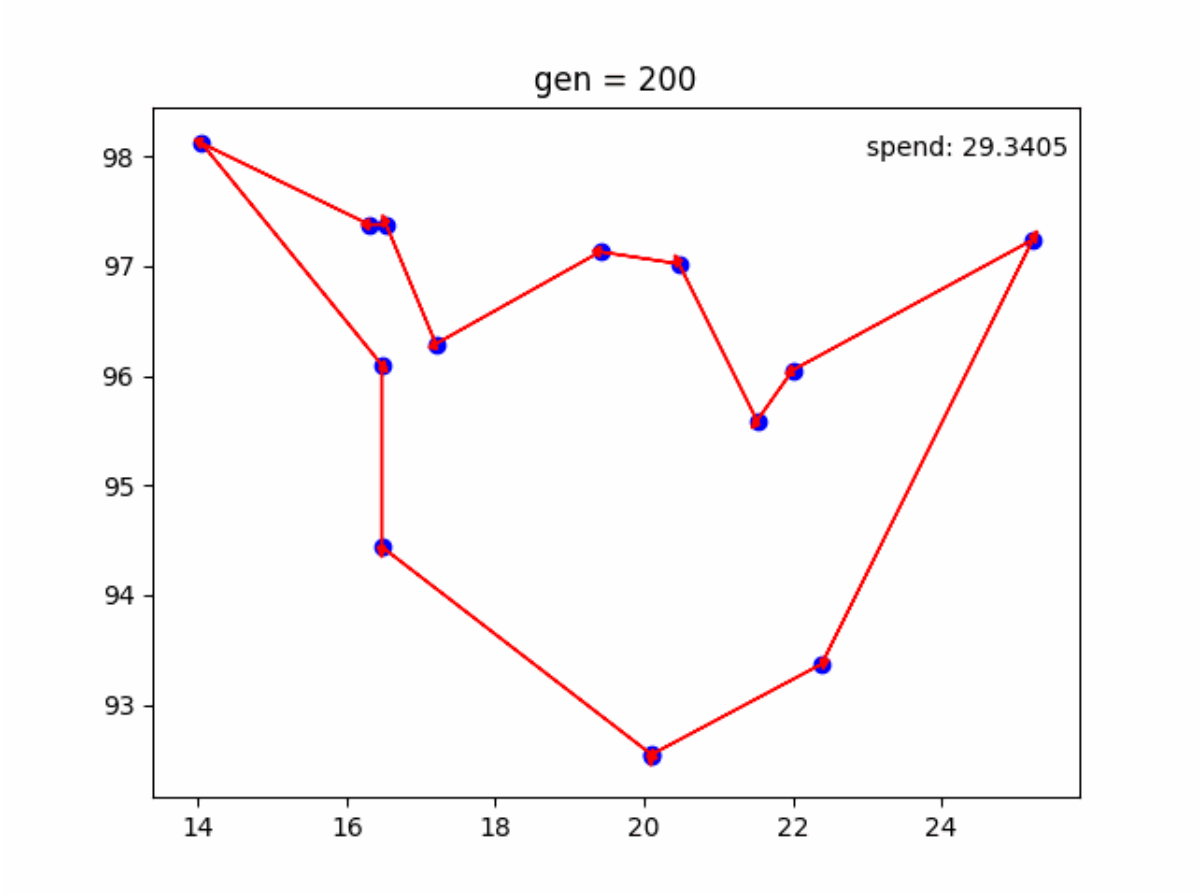


图 1: 进化过程

最终路径图如下：



3 分析遗传算法的特点以及不足之处

对于一个非凸问题，遗传算法可能会收敛到局部最优解；如果交叉、变异的概率过大可能会很难收敛（使得原本优秀的个体交叉变异为劣质的个体），出现震荡的情况，概率过小又有可能影响物种多样性，使得收敛速度过慢。

4 说一说遗传算法的主要应用场景

求解多变量问题的最大最小值的问题，可以避免蛮力搜索的大量 for 循环或是矩阵运算。

参考文献

[1] 史峰, 王辉, 郁磊, 胡斐, MATLAB 智能算法 30 个案例分析, 北京: 北京航空航天大学出版社,2011.7

附录

Listing 1: **TSP.py**

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation


class GeneticAlgorithm( object):
    def __init__(self) -> None:
        pass

    def initPop(self, NIND, N):
        """
        description: 初始化种群
        param: NIND 种群大小
        param: N 个体染色体长度(城市个数)
        Returns: Chrom 初始种群
        """

        Chrom = np.zeros(shape=(NIND, N), dtype= int)
        for i in  range(NIND):
            Chrom[i] = np.random.permutation(N)

        return Chrom

    def distance_matrix(self, position):
        """
        description: 计算距离矩阵
        param: position 每个城市的坐标
        Returns: Distance 距离矩阵
        """

        L =  len(position)
        Distance = np.zeros(shape=(L, L), dtype= float)
        for i in  range(L):
            for j in  range(L):
                Distance[i, j] = np.sqrt(np.power(
                    position[i, 0] - position[j, 0], 2) + np.power(position[i, 1] - position[j, 1],
                        2))

        return Distance

    def path_distance(self, Chrom, Distance):
        """
        description: 适应度函数
```

```
43          param: Chrom 种群
44          param: Distance 距离矩阵
45          Returns: distance 个体距离向量
46          """
47
48          NIND, N = Chrom.shape
49          distance = np.zeros(shape=(NIND, 1), dtype= float)
50          for i in  range(NIND):
51              for j in  range(N - 1):
52                  distance[i, 0] += Distance[Chrom[i, j], Chrom[i, j + 1]]
53              distance[i, 0] += Distance[Chrom[i, j + 1], Chrom[i, 0]]
54
55          return distance
56
57      def fitness(self, distance):
58          """
59          description: 适应度函数
60          param: distance 个体长度向量(TSP距离)
61          Returns: FitnV 个体适应度向量
62          """
63
64          return 1 / distance
65
66      def select(self, Chrom, FitnV, GGAP):
67          """
68          description: 选择操作
69          param: Chrom 种群
70          param: FitnV 适应度
71          param: GGAP 选择概率
72          Returns: SelCh 被选择的个体
73          """
74
75          def sus(Nsel, FitnV):
76              """
77              description:
78              param: FitnV 个体适应度向量
79              param: Nsel 被选择个体的数目
80              Returns: sel_index 选择到的个体索引
81              """
82
83              NIND = FitnV.shape[0]
84              sel_index = np.zeros(Nsel, dtype= int)          # 保存被选择的个体下标
85              cumfit = np.cumsum(FitnV)                       # 累计适应度
86              cumfit_one = cumfit / cumfit[-1]                # 映射到[0, 1]
87              for i in  range(Nsel):
88                  rand = np.random.rand()
89                  for j in  range(NIND):
```

```
90                      if cumfit_one[j] > rand:
91                          sel_index[i] = j
92                          break
93
94              return sel_index
95
96          NIND = Chrom.shape[0]                                   # 种群个数
97          # 被选择的个体数
98          Nsel =  int(np. max(np.array([np.ceil(NIND * GGAP), 2])))
99          sel_index = sus(Nsel=Nsel, FitnV=FitnV)
100         SelCh = Chrom[sel_index, :]
101
102         return SelCh
103
104     def recombin(self, SelCh, Pc):
105         """
106         description: 交叉操作
107         param: SelCh 被选中的个体
108         param: Pc 交叉概率
109         Returns: Selch 交叉后的个体
110         """
111
112         def intercross(A, B):
113             """
114             description: 两条染色体交叉
115             param: A 染色体A
116             param: B 染色体B
117             Returns: 交叉后的两条染色体A, B
118             """
119
120             L =  len(A)
121             # 随机生成两个交叉点
122             r1, r2 = 0, 0
123             while r1 == r2:
124                 r1 = np.random.randint(low=1, high=L - 1)
125                 r2 = np.random.randint(low=1, high=L - 1)
126             if r1 > r2:
127                 r1, r2 = r2, r1
128             # 交叉
129             for i in  range(r1, r2):
130                 A[i], B[i] = B[i], A[i]
131             # 解决A冲突
132             for i in  range(r1):
133                 j = r1
134                 while j < r2:
135                     if A[i] == A[j]:
136                         A[i] = B[j]
```

```
                        j = r1
                        continue
                    j += 1
            for i in  range(r2, L):
                j = r1
                while j < r2:
                    if A[i] == A[j]:
                        A[i] = B[j]
                        j = r1
                        continue
                    j += 1

            # 解决B冲突
            for i in  range(r1):
                j = r1
                while j < r2:
                    if B[i] == B[j]:
                        B[i] = A[j]
                        j = r1
                        continue
                    j += 1
            for i in  range(r2, L):
                j = r1
                while j < r2:
                    if B[i] == B[j]:
                        B[i] = A[j]
                        j = r1
                        continue
                    j += 1

            return A, B

        Nsel = SelCh.shape[0]
        for i in  range(Nsel - 1):
            rand = np.random.rand()
            if rand < Pc:
                SelCh[i, :], SelCh[i + 1,
                                   :] = intercross(SelCh[i, :], SelCh[i + 1, :])

        return SelCh

    def mutate(self, SelCh, Pm):
        """
        description: 变异操作
        param: SelCh 被选择的个体
        param: Pm 变异概率
        Returns: SelCh 变异后的个体
```

```
        """

        Nsel, L = SelCh.shape
        for i in  range(Nsel):
            rand = np.random.rand()
            if rand < Pm:
                r1, r2 = 0, 0
                while r1 == r2:
                    r1 = np.random.randint(low=0, high=L)
                    r2 = np.random.randint(low=0, high=L)
                if r1 > r2:
                    r1, r2 = r2, r1
                SelCh[i, r1], SelCh[i, r2] = SelCh[i, r2], SelCh[i, r1]

        return SelCh

    def reverse(self, SelCh, Distance):
        """
        description: 进化逆转操作
        param: SelCh 被选择的个体
        param: Distance 距离矩阵
        Returns: SelCh 进化逆转后的个体
        """

        row, L = SelCh.shape
        ObjV = self.path_distance(Chrom=SelCh, Distance=Distance)
        SelCh_1 = SelCh.copy()
        for i in  range(row):
            r1, r2 = 0, 0
            while r1 == r2:
                r1 = np.random.randint(low=0, high=L)
                r2 = np.random.randint(low=0, high=L)
            if r1 > r2:
                r1, r2 = r2, r1
            SelCh_1[i, r1], SelCh_1[i, r2] = SelCh_1[i, r2], SelCh_1[i, r1]
        ObjV_1 = self.path_distance(Chrom=SelCh_1, Distance=Distance)
        for i in  range(row):
            if ObjV[i] > ObjV_1[i]:
                SelCh[i, :] = SelCh_1[i, :]

        return SelCh

    def reins(self, Chrom, SelCh, ObjV, Distance, opt_path):
        """
        description: 父代精英重插入
        param: Chrom 父代种群
        param: SelCh 子代种群
```

```
231         param: ObjV 父代适应度
232         param: Distance 距离矩阵
233         param: opt_path 父代精英
234         Returns: SelCh 组合后得到的新种群
235         """
236 
237         NIND = Chrom.shape[0]
238         Nsel = SelCh.shape[0]
239         index = np.argsort(ObjV)
240         for i in  range(NIND - Nsel):
241             SelCh = np.vstack((SelCh, Chrom[index[i]]))
242         ObjV_s = self.path_distance(SelCh, Distance)
243         # 用父代精英替换最bad的个体
244         bad_index = np.argmax(ObjV_s)
245         SelCh[bad_index, :] = opt_path[:]
246         return SelCh
247 
248 
249 class TSP( object):
250     def __init__(self, pop_size, city_size, position, iters, Pc, Pm, Pa) -> None:
251         self.pop_size = pop_size                # 种群大小
252         self.city_size = city_size              # 城市个数
253         self.position = position                # 城市坐标
254         self.iters = iters                      # 迭代次数
255         self.Pc = Pc                            # 交叉概率
256         self.Pm = Pm                            # 变异概率
257         self.Pa = Pa                            # 选择概率
258         self.ga = GeneticAlgorithm()            # 遗传算法实例
259         self.opt_distance = np.zeros(
260             shape=(self.iters, 1), dtype= float)                # 记录每代最优个体值
261         self.opt_path = np.zeros(
262             shape=(1, self.city_size))                          # 父代精英
263         self.all_opt_path = np.zeros(
264             shape=(self.iters, self.city_size), dtype= int)     # 记录每代最优个体
265 
266     def compute_optimal_path(self):
267         # 初始化种群
268         self.Chrom = self.ga.initPop(self.pop_size, self.city_size)
269         # 计算距离矩阵
270         self.Distance = self.ga.distance_matrix(self.position)
271         # 初始的距离
272         self.ObjV = self.ga.path_distance(self.Chrom, self.Distance)
273         # 初始时最优个体值
274         self.preObjV = np. min(self.ObjV)
275         # 初始时最优个体
276         self.opt_path = self.Chrom[np.argmin(self.ObjV)]
277         # 开始进化
```

```
278         for gen in  range(self.iters):
279             # 计算适应度向量
280             self.ObjV = self.ga.path_distance(self.Chrom, self.Distance)
281             # 计算当前代适应度
282             self.FitnV = self.ga.fitness(self.ObjV)
283             # 选择
284             self.SelCh = self.ga.select(self.Chrom, self.FitnV, self.Pa)
285             # 交叉
286             self.SelCh = self.ga.recombin(self.SelCh, self.Pc)
287             # 变异
288             self.SelCh = self.ga.mutate(self.SelCh, self.Pm)
289             # 逆进化
290             self.SelCh = self.ga.reverse(self.SelCh, self.Distance)
291             # 找出最优个体(与父代精英比较)
292             if np. min(self.ObjV) < self.preObjV:
293                 self.preObjV = np. min(self.ObjV)
294                 self.opt_path = self.Chrom[np.argmin(self.ObjV)]
295             # 记录每代最优路径值
296             self.opt_distance[gen, 0] = self.preObjV
297             # 记录每代最优路径
298             self.all_opt_path[gen, :] = self.opt_path
299             # 重插入
300             self.Chrom = self.ga.reins(
301                 self.Chrom, self.SelCh, self.ObjV, self.Distance, self.opt_path)
302 
303     def draw_evolution(self):
304         fig = plt.figure()
305         ax = fig.add_subplot(111)
306         x = np.arange(self.iters)
307         ax.plot(x, self.opt_distance, label="optimal", color="orange")
308         ax.set_xlabel("The number of iterations")
309         ax.set_ylabel("The optimal distance of each generation")
310         ax.set_title("The process of evolution")
311         ax.legend()
312         plt.savefig("evolution.pdf", bbox_inches="tight")
313         plt.show()
314 
315     def draw_distance(self):
316         fig = plt.figure()
317         ax = fig.add_subplot(111)
318         def update(j):
319             path = self.all_opt_path[j, :]
320             plt.cla()
321             ax.scatter(self.position[:, 0],
322                        self.position[:, 1], marker="o", c="b")
323             ax.set_title("gen = %d" %(j + 1))
324             ax.text(23, 98, "spend: %.4f"%self.opt_distance[j, 0], fontsize = 10)
```

```
            for i in  range(self.city_size):
                x_1, y_1 = self.position[path[i % self.city_size]]
                x_2, y_2 = self.position[path[(i + 1) % self.city_size]]
                dx, dy = x_2 - x_1, y_2 - y_1
                ax.arrow(x_1, y_1, dx, dy, head_width=0.1,
                         head_length=0.1, fc="r", ec="r")
        gens = np.hstack((np.arange(0, iters, 5), np.array([iters - 1])))
        ani = FuncAnimation(fig, update, frames=gens, interval=150, blit=False, repeat=False)
        ani.save("tsp.gif", writer='imagemagick')
        plt.show()


if __name__ == "__main__":
    pop_size = 20
    city_size = 14
    position = np.array([[16.47, 96.10],
                         [16.47, 94.44],
                         [20.09, 92.54],
                         [22.39, 93.37],
                         [25.23, 97.24],
                         [22.00, 96.05],
                         [20.47, 97.02],
                         [17.20, 96.29],
                         [16.30, 97.38],
                         [14.05, 98.12],
                         [16.53, 97.38],
                         [21.52, 95.59],
                         [19.41, 97.13],
                         [20.09, 92.55]])
    iters = 200
    Pc = 0.7
    Pm = 0.05
    Pa = 0.8
    tsp = TSP(pop_size=pop_size, city_size=city_size,
              position=position, iters=iters, Pc=Pc, Pm=Pm, Pa=Pa)
    tsp.compute_optimal_path()
    print("最优路径: ", tsp.opt_path)
    print("花费代价: ", tsp.preObjV)
    tsp.draw_evolution()
    tsp.draw_distance()

# 最优路径:  [ 8  9  0  1 13  2  3  4  5 11  6 12  7 10]
# 花费代价:  29.340520066994223
```