1 15-Puzzle 问题

1.1 实验原理

分别利用 A* 搜索算法与 IDA* 算法进行本实验。

1.1.1 A* 算法原理

在 A* 算法中, 令 f(S) 为状态 S 的估值,g(S) 为从初始状态到达 S 所需步数,h(S) 为从状态 S 到达目标状态所需步数的估计值, 有

$$f(S) = g(S) + h(S)$$

每个状态的 F 值可以由优先队列进行维护, 搜索时每次从优先队列 (即 open 表) 中取出堆顶元素 (即队列中 F 值最小的元素), 并由此元素生成其下一状态加入至队列中, 最后将此元素放入 close 表 (在程序中表现为将其标记为已搜过)。

1.1.2 启发式函数的选取

对于启发式函数的选取, 结合实际问题的复杂性, 选择使用曼哈顿距离 + 线性惩罚函数:
曼哈顿距离:

$$dist(a, b) = |a_x - b_x| + |a_y - b_y|$$

线性惩罚则表现为:
若同行或同列的两个数字的相对位置相反, 则它们会产生线性冲突, 惩罚因子 +2。
因此, 启发式函数 h 为:

$$h(S) = \sum_{i=1}^{4}\sum_{j=1}^{4} dist(S_{ij}, Target(S_{ij})) + \text{惩罚因子}$$

此启发式函数优点如下:

- 可采纳性: $\forall S, h(S) \leq h^*(S)$, 能保证一定能搜索到最优解
- 一致性: 令 $S^{'}$ 为 S 的后继状态,$\forall S, h(S) \leq C(S, S^{'}) + h(S^{'})$, 此性质保证第一次搜到节点就能保证找到最优解。

- 性能高: 横向对比其它启发式函数, 在解决步数 45 步以上的 15puzzle 问题时, 程序运行时间极大缩短

1.1.3 有无解的判断

判断有无解利用以下性质:
空格左右移动,15 个数的逆序数不变
空格上下移动, 逆序数的奇偶性改变
因此一个 15puzzle 问题有解, 当且仅当: 初始状态逆序数 + 空格所在行 = 目标状态逆序数 + 空格所在行

1.1.4 状态的存储

50 步以上的样例对程序的性能要求极高，单纯的元组/列表存储状态会导致内存过载，影响程序性能，因此采用位压缩的方式进行存储:
每个格子存储一个数字 (0-15), 可以用 4 位二进制压缩存储，则 4*4 的格子刚好可以用 64bit 的整数来存储，在此按行优先顺序存储每个格子的值:

```
(1,1) (1,2) (1,3) (1,4)
(2,1) (2,2) (2,3) (2,4)
(3,1) (3,2) (3,3) (3,4)
(4,1) (4,2) (4,3) (4,4)
```

同时，预先定义移位表，便于快速判断第 (i,j) 的数字所在的移位量:

```
shift_map = [
    [0,0,0,0,0]
    [0,60,56,52,48],  # (1,1) (1,2) (1,3) (1,4)
    [0,44,40,36,32],   # (2,1) (2,2) (2,3) (2,4)
    [0,28,24,20,16],   # (3,1) (3,2) (3,3) (3,4)
    [0,12,8,4,0]     # (4,1) (4,2) (4,3) (4,4)
]
```

解码时, 令状态为 S, 对于坐标 (i,j), 移位量 shift=shift_map[i,j], 从而按照以下方式解码:

$$\text{Value} = (S \gg \text{shift}) \wedge 0xF$$

1.2 关键代码展示

首先是启发式函数的实现:

```
def h(num):
    ans = 0
    for i in range(1,5):
        for j in range(1,5):
            shift = shift_map[i][j]
            val = (num >> shift) & 0xF
            if val == 0:
                continue
            tx, ty = tar[val]
            ans += abs(i - tx) + abs(j - ty)
            # Check row conflicts
            if i == tx:
                for jj in range(j + 1, 5):
                    shift_jj = shift_map[i][jj]
                    val_jj = (num >> shift_jj) & 0xF
                    if val_jj==0:
                        continue
                    tx_jj, ty_jj = tar[val_jj]
                    if tx_jj == i:
                        if (ty < ty_jj) != (j < jj):
                            ans += 2
            if j == ty:
                for ii in range(i + 1, 5):
                    shift_ii = shift_map[ii][j]
                    val_ii = (num >> shift_ii) & 0xF
                    if val_ii==0:
                        continue
                    tx_ii, ty_ii = tar[val_ii]
                    if ty_ii == j:
                        if (tx < tx_ii) != (i < ii):
                            ans += 2
    return ans
```

接着是有无解的判断: 逆序数 + 空格所在行数与目标的奇偶性是否相同,(目标状态为奇数)

```
def calc(a):
    d = []
    row = 0
    for i in range(1, 5):
        for j in range(1, 5):
            if a[i][j] != 0:
                d.append(a[i][j])
            else:
                row = 5 - i
    ans = 0
    for i in range(len(d)):
        for j in range(i):
            if d[j] > d[i]:
                ans += 1
    return ans + row
```

优先队列存储 (f 值,g 值, 当前状态 S),fa 用于最后进行回溯得出解决方案

```
start = array_to_int(a)
start_f = 0 + h(start)
q = []
heapq.heappush(q, (start_f, 0, start, sx, sy))
fa[start] = None
global success
success = False
start_time = time.time()
node_count = 0
```

搜索过程:

```
while q and not success:
    cur_f, u_g, u_s, u_x, u_y = heapq.heappop(q)
    if u_s in st:
        continue
    node_count += 1
```

```
    st.add(u_s)
    if h(u_s) == 0:
        success = True
        print(u_g)
        path = []
        def backtrace(state):
            if fa.get(state) is None:
                x, y = (u_x, u_y) if state == start else int_to_pos
                    (state)
                path.append(f"({x},{y})")
                return
            backtrace(fa[state])
            x, y = int_to_pos(state)
            path.append(f"({x},{y})")
        backtrace(u_s)
        print("->".join(path))
        break
    x, y = u_x, u_y
    for i in range(4):
        nx, ny = x + dx[i], y + dy[i]
        if 1 <= nx <= 4 and 1 <= ny <= 4:
            shift1 = shift_map[x][y]
            shift2 = shift_map[nx][ny]
            val2 = (u_s >> shift2) & 0xF  #要交换的值
            mask = (0xF << shift1) | (0xF << shift2)  #掩码:把val2
                原本的位置变为空格
            new_s = u_s & ~mask
            new_s |= (val2 << shift1)  #把val2放在原本空格的位置
            new_g = u_g + 1
            if new_s not in st and new_s not in fa:
                fa[new_s] = u_s
                new_f = new_g + h(new_s)
                heapq.heappush(q, (new_f, new_g, new_s, nx, ny))
```

而对于迭代加深 IDA* 算法，在每次搜索中限制深度即可

```
maxdep = 0
while success == 0:
    dfs(maxdep)
    end = time.time()
    print("step:%d  Running time:%fs" % (maxdep, end - begin))
    maxdep += 3
```

1.3 创新点

- 运用位压缩存储状态，并通过位运算进行编码解码，带来了性能的极大提高，在 62 步的样例中仅花费 538 秒
- 加入了有无解的判断, 所有 15puzzle 问题并不是一定有解的, 需要通过逆序数 + 空格所在行数进行判断
- 选取了兼具可采纳性与一致性的启发式函数, 简化了程序的实现与搜索步骤, 只需用一个 set 记录 close 表判重即可。

1.4 结果展示

由于 IDA* 算法效率较低，仅展示 A* 算法所得结果。以下是各个样例的最短步数, 以及解法, 以及程序运行时间，探索的节点数:

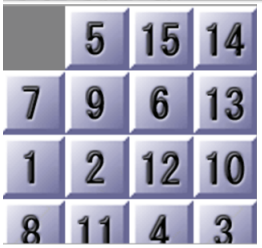

```
PS D:\college\AI实验\homework\week5> & D:/Python/python.exe d:/college/AI实验/homework/week5/15puzzleDS.py
1 2 4 8
5 7 11 10
13 15 0 3
14 6 9 12
22
(4,4)->(3,2)->(4,2)->(4,3)->(3,3)->(2,3)->(2,4)->(3,4)->(3,3)->(2,3)->(2,4)->(1,4)->(1,3)->(2,3)->(2,2)->(3,2)->(4,2)->(4,1)->(3
4)->(4,4)
Total running time: 0.018223285675048828s
total explore nodes: 157
```


```
PS D:\college\AI实验\homework\week5> & D:/Python/python.exe d:/college/AI实验/homework/week5/15puzzleDS.py
14 10 6 0
4 9 1 8
2 3 5 11
12 13 7 15
49
(4,4)->(1,3)->(1,2)->(2,2)->(2,1)->(1,1)->(1,2)->(2,2)->(2,3)->(1,3)->(1,2)->(2,2)->(3,2)->(3,1)->(2,1)->(1,1)->(1,2)->(2,2)->(
1)->(3,2)->(3,3)->(3,4)->(2,4)->(1,4)->(1,3)->(1,2)->(2,2)->(3,2)->(4,2)->(4,3)->(3,3)->(3,2)->(4,2)->(4,1)->(3,1)->(2,1)->(2,
>(3,3)->(3,2)->(2,2)->(2,3)->(3,3)->(4,3)->(4,4)
Total running time: 8.17817497253418s
total explore nodes: 446383
```


```
PS D:\college\AI实验\homework\week5> & D:/Python/python.exe d:/college/AI实验/homework/week5/15puzzleDS.py
5 1 3 4
2 7 8 12
9 6 11 15
0 13 10 14
15
(4,4)->(4,2)->(4,3)->(4,4)->(3,4)->(2,4)->(2,3)->(2,2)->(2,1)->(1,1)->(1,2)->(2,2)->(3,2)->(4,2)->(4,3)->(4,4)
Total running time: 0.0s
total explore nodes: 20
```


```
PS D:\college\AI实验\homework\week5> & D:/Python/python.exe d:/college/AI实验/homework/week5/15puzzleDS.py
6 10 3 15
14 8 7 11
5 1 0 2
13 12 9 4
48
(4,4)->(4,3)->(4,2)->(4,1)->(3,1)->(3,2)->(3,3)->(2,3)->(2,4)->(3,4)->(4,4)->(4,3)->(4,2)->(3,2)->(3,3)->(2,3)->(2,4)->(1,4)->(1
4)->(3,3)->(2,3)->(2,2)->(2,1)->(3,1)->(4,1)->(4,2)->(4,3)->(3,3)->(3,2)->(2,2)->(1,2)->(1,1)->(2,1)->(3,1)->(4,1)->(4,2)->(3,2
>(1,4)->(2,4)->(2,3)->(3,3)->(3,4)->(4,4)
Total running time: 17.78942346572876s
total explore nodes: 964991
```


```
56
(4,4)->(3,4)->(3,3)->(2,3)->(2,2)->(3,2)->(4,2)->(4,3)->(4,4)->(3,4)->(3,3)->(3,2)->(3,1)->(4,1)->(4,2)->(4,3)->(3,3)->(3,2)->(
2)->(2,2)->(2,1)->(1,1)->(1,2)->(1,3)->(2,3)->(2,2)->(2,1)->(1,1)->(1,2)->(1,3)->(2,3)->(2,4)->(3,4)->(4,4)->(4,3)->(3,3)->(3,4
>(2,3)->(2,2)->(2,1)->(3,1)->(3,2)->(3,3)->(2,3)->(2,2)->(1,2)->(1,3)->(2,3)->(2,4)->(3,4)->(4,4)
Total running time: 245.12845826148987s
total explore nodes: 10609356
```




```
62
(4,4)->(2,1)->(2,2)->(3,2)->(3,1)->(2,1)->(2,2)->(1,2)->(1,1)->(2,1)->(2,2)->(3,2)->(4,2)->(4,1)->(3,1)->(2,1)->(2,2)->(2,3)->(
3)->(4,2)->(3,2)->(3,3)->(2,3)->(2,4)->(3,4)->(4,4)->(4,3)->(3,3)->(2,3)->(1,3)->(1,4)->(2,4)->(3,4)->(4,4)->(4,3)->(3,3)->(2,
>(2,1)->(3,1)->(3,2)->(4,2)->(4,3)->(3,3)->(2,3)->(1,3)->(1,4)->(2,4)->(3,4)->(4,4)->(4,3)->(3,3)->(3,2)->(3,1)->(4,1)->(4,2)->
Total running time: 538.7775294780731s
total explore nodes: 25745173
```

2 利用遗传算法求解 TSP 问题

2.1 实验原理

遗传算法包括基因的定义, 适应度的计算、自然选择、变异、基因重组的过程, 在旅行商问题中, 各个要素的定义与实现如下

- 基因的定义: 在 TSP 问题中, 基因定义为 1-n 的排列, 其中 n 为问题规模 (即 city 数量), 其意义是访问城市的顺序
- 适应度的计算: 对于一个基因, 求出按此基因顺序访问城市的总路程长度 y, 则适应度为

fit_value=-y, 其意义是总路程越短越好

- 自然选择: 采用锦标赛的方式, 每次从种群中随机选择 siz(通常为 3-5 左右) 个个体, 并从中选取适应度最高的一个; 重复 population_size(种群规模) 次, 即完成自然选择。
- 变异: 变异是基于一定概率的变异, 采取反转子段的方式进行变异。同时还有变异概率的衰减系数 γ , 体现在变异概率在每一轮后乘以 γ, 其意义是前期广泛探索以免陷入局部最优, 后期则减少变异保留最优个体
- 基因重组: 随机选取双亲同一子段, 进行排列交换, 以下为一个例子:
  双亲一: 1 4 2 3 5
  双亲二: 2 5 4 1 3
  选取的子段:[3,4] (下标从 1 开始)
  产生的子代一:3 2 4 1 5(双亲一第三四个变成了双亲二对应的 4,1)
  产生的子代二: 4 5 2 3 1(双亲二第三四个变成了双亲一对应的 2,3)

2.2 关键代码展示

以类的形式创建一个遗传算法的框架:

```
class GA_TSP:
    def __init__(self,F,n_dim,population_size,max_iter,
        mutation_prob,gamma=0.99):
        self.F=F    #函数
        self.n_dim=n_dim    #维度
        self.population_size=population_size
        self.max_iter=max_iter
        self.mutation_prob=mutation_prob
        self.X=None    #population_size*n_dim
        self.Y=None    #population_size,1
        self.fit_value=None #population_size,1
        self.gamma=gamma
        self.generation_best_X = []
        self.generation_best_Y = []

        self.best_x, self.best_y = None, None
```

```
        self.create()
```

种群的初始化, 创造一个全排列

```
def create(self):  #创造初始种群
    '''创建一个全排列'''
    x=np.random.rand(self.population_size,self.n_dim)
    self.X=x.argsort(axis=1)
```

适应度的计算:

```
def rank(self): #计算适应度
    self.fit_value=-self.Y
```

变异, 经检测, 反转子段的效果优于交换两点:

```
def mutation(self): #变异
    '''交换两个点:效果很差'''
    # for i in range(self.population_size):
    #     for j in range(self.n_dim):
    #         if np.random.rand()<self.mutation_prob:
    #             k=np.random.randint(0,self.n_dim,1)
    #             self.X[i,j],self.X[i,k]=self.X[i,k],self.X[i,j]
    '''反转子段:效果很好'''
    for i in range(self.population_size):
        if np.random.rand() < self.mutation_prob:
            j, k = np.sort(np.random.choice(self.n_dim, 2, replace=
                False))
            self.X[i, j:k+1] = self.X[i, j:k+1][::-1]
```

自然选择, 采取锦标赛的方式

```
def select(self,siz=3): #选择
    '''采用锦标赛方式:每次随机挑选siz个个体为候选者,从中选一个最优
       的,重复population_size次'''
    candidate_idx=np.random.randint(self.population_size,size=(self
       .population_size,siz))
    candidate_value=self.fit_value[candidate_idx]
```

```
    winner=candidate_value.argmax(axis=1)
    select_idx=[candidate_idx[i,j] for i,j in enumerate(winner)]
    self.X=self.X[select_idx,:]
```

基因重组, 产生后代:

```
def cross(self,cross_prob=0.8): #交叉,生成后代
    '''排列交叉,维护基因唯一性'''
    for i in range(0,self.population_size,2):
        if np.random.rand()<cross_prob:
            p,k=np.random.randint(0,self.n_dim,2)
            if p>k:
                p,k=k,p
            mp1={value: idx for idx,value in enumerate(self.X[i])}
            mp2={value: idx for idx,value in enumerate(self.X[i+1])
               }
            for j in range(p,k):
                val1,val2=self.X[i,j],self.X[i+1,j]
                pos1,pos2=mp1[val2],mp2[val1]
                self.X[i,j],self.X[i,pos1]=self.X[i,pos1],self.X[i,
                   j]
                self.X[i+1,j],self.X[i+1,pos2]=self.X[i+1,pos2],
                   self.X[i+1,j]
                mp1[val1],mp1[val2]=pos1,j
                mp2[val1],mp2[val2]=j,pos2
```

迭代运行阶段:

```
def run(self):
    for i in range(self.max_iter):
        OLD_X=self.X.copy()  #原始种群需要保留
        self.Y=self.F(self.X)
        self.rank()
        self.select()
        self.cross()
        self.mutation()
```

```
            self.X=np.concatenate([OLD_X,self.X],axis=0) #新产生的种群
                和原有种群一起参与自然选择,保留原有种群的优秀个体
            self.Y=self.F(self.X)
            self.rank()
            select_idx=np.argsort(self.Y)[:self.population_size]
            self.X=self.X[select_idx,:]
            self.Y = self.Y[select_idx]
            self.rank()

            '''记录最优'''
            best_index=self.fit_value.argmax()
            self.generation_best_X.append(self.X[best_index,:].copy())
            self.generation_best_Y.append(self.Y[best_index])
            self.mutation_prob=self.mutation_prob*self.gamma

            best_index=np.array(self.generation_best_Y).argmin()
            best_x=self.generation_best_X[best_index]
            best_y = self.F(np.array([best_x]))[0]
            print(f'iteration:{i} best_y:{best_y}')
        global_best_index=np.array(self.generation_best_Y).argmin()
        self.best_x=self.generation_best_X[global_best_index]
        self.best_y = self.F(np.array([self.best_x]))[0]
        return self.best_x,self.best_y
```

2.3 实验结果展示

碍于机器性能限制, 本实验中采用了卡塔尔 194 个城市的数据集, 本数据集中已求得最优解为 **9352**, 可作为评估算法精确性的依据。

受限于问题规模以及运行时间, 本实验一致采取迭代 1000 轮得到的最优解评估算法性能, 同时根据 194 的城市规模, 设定合适的种群大小为 500, 以下是针对变异概率的不同参数的运行结果:

首先是变异概率动态变化, 即 $\gamma<1$:

参数如下

```
ga_tsp=GA_TSP(F=cal_total_distance,n_dim=num_points,population_size=500,max_iter=1000,mutation_prob=1,gamma=0.99)
```

```
[ 12  24  32  17  28  27  68  74  76  78  83  55  43  52  51  53  50  39
  33  30  49  65  60  72  69  75  25  23  20  21   6   3   4   2   1   0
   5  64  85   7  15  19  35  62  61  58  84  97  93 103 100 110 105  91
  87  70  81  88  89  98 129 136 125 126 124 131 137 143 140 130 135 120
 119 127 122 116 115 157 167 172 163 156 174 177 179 187 183 189 191 190
 192 188 180 176 184 170 165 169 166 161 150 151 154 164 149 138 153 168
 178 182 186 171 173 185 160 162 175 193 181 152 145 144 139 155 141 133
 148  79  77  16  36  26  41  54  48  47  45  37  38  46  66  44  67  59
  71  73 104  96 107 117  95  94  86  90  92 102 106 114 111 109 112 101
 108 113 118 121 147 159 132 146 158 134 123 142 128  99  80  82  63  56
  34  42  57  40  14  29  31  18   9  11   8  10  13  22] 20005.073476919228
```

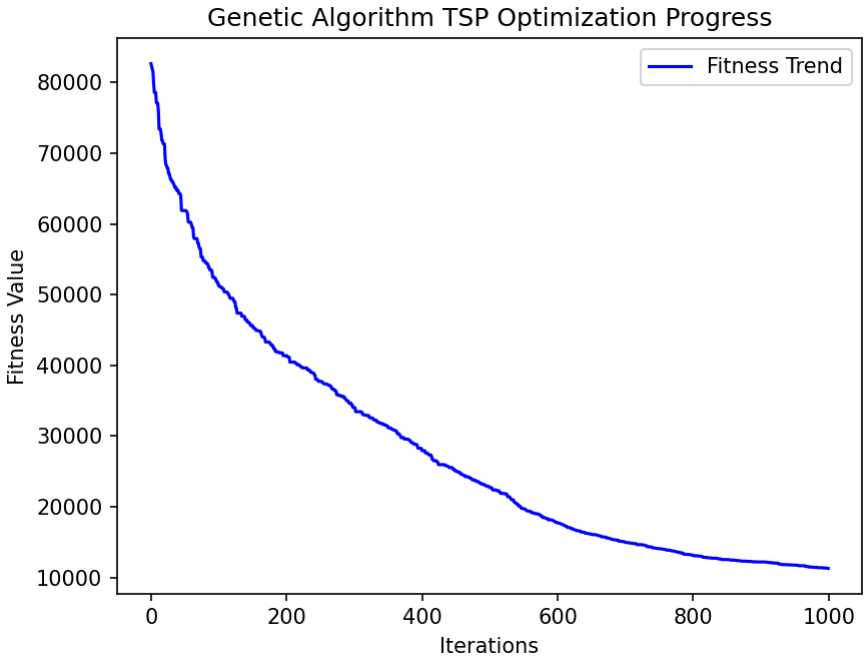
结果并不理想

究其原因,$0.99^{300} = 0.049$, 导致在中期变异概率已经接近为 0, 缺乏变异带来的增益

于是将 γ 调整为 0.999, 结果如下

```
ga_tsp=GA_TSP(F=cal_total_distance,n_dim=num_points,population_size=500,max_iter=1000,mutation_prob=1,gamma=0.999)
```

```
[149 153 156 146 150 158 157 166 165 159 154 147 142 135 130 134 132 123
 122 127 128 117 120 119 116 115 114 109 111 107 106 104 105  95  87  82
  76  67  72  66  60  65  38  26  21  28  44  27  32  17  20  23  25  68
  59  56  63  36  37  40  47  52  45  53  51  55  57  39  30  31  33  46
  50  42  43  48  54  49  41  34  29  18  14  11   9   8   4   2   6   3
   1   0   5   7  15  12  13  16  10  24  22  70  81  61  58  35  62  19
  64  84  85  97  89  88  93 100  98 103 110 126 113 118 112 108  79  75
  86  90  74  77  71  73  69  80  78  83  99  96  91  94  92 102 101 121
 137 140 143 138 145 141 139 136 148 144 133 131 124 125 129 155 163 162
 160 168 175 181 193 185 186 178 171 173 172 174 183 182 189 188 191 190
 192 187 180 176 177 179 184 170 169 167 164 161 151 152] 11280.142252342102
```



经过了参数调整, 极大的改善了结果, 非常接近最优解 9352

接着就是变异概率不变的情况, 分别测试了概率为 0.1 和 0.2 的情况, 结果如下:

```
ga_tsp=GA_TSP(F=cal_total_distance,n_dim=num_points,population_size=500,max_iter=1000,mutation_prob=0.1,gamma=1)
```

```
[ 41  49  34  31  29  18  14   4   2   8   9  11  30  37  40  33  36  63
  56  44  28  26  21  27  32  23  25  13  10  16  17  20  59  68  77  90
  86 101 102  91  87  69  73  71  74  75  70  22  24  79  81  89  97  85
  84  64  19   0   1   3   6   5   7  15  12  35  62  61  58  88  93  98
 100 103 110 118 125 136 148 144 141 145 139 133 131 129 126 124 113 112
 108 121 130 146 151 149 143 140 138 137 153 156 152 167 166 161 157 158
 164 176 180 183 174 172 173 182 185 186 178 171 163 160 155 162 168 175
 181 193 189 191 190 188 187 192 177 179 169 184 170 165 159 150 154 147
 142 135 132 134 128 119 122 123 127 120 117 116 115 111 114 109  99 106
 107 104 105  96  95  92  94  82  83  80  78  76  65  67  66  72  50  60
  57  52  51  45  47  53  55  42  46  38  39  43  48  54] 12268.162169068819
```

```
ga_tsp=GA_TSP(F=cal_total_distance,n_dim=num_points,population_size=500,max_iter=1000,mutation_prob=0.2,gamma=1)
```

```
[175 171 181 193 189 188 187 190 191 186 185 172 173 174 180 176 177 179
 169 167 164 146 151 149 152 156 153 145 141 148 144 155 139 133 131 129
 126 124 125 136 137 138 143 140 121 118 113 112 108 101  86  79  81  88
 100 110 103  98  93  89  85  97  84  64  19  62  35   5   0   1   2   6
   3   7  15  58  61  75  70  24  22  12  10  13  16  25  23  20  17  21
  26  11   9   8   4  14  18  29  31  30  41  49  54  48  53  47  51  52
  55  57  46  50  45  43  34  40  37  42  39  33  38  36  28  44  27  32
  56  59  68  71  73  74  77  90  82  80  78  76  69  63  67  65  60  66
  72  83  91  87  92  94  95  96  99 109 111 114 107 106 104 105 102 117
 116 115 120 119 122 123 127 132 134 128 130 135 142 147 154 159 165 161
 157 150 158 166 170 184 192 183 182 178 163 162 160 168] 11237.150782690704
```

可见在概率为 0.2 的情况下结果更优, 说明算法更依赖于变异带来的增益。

2.4 创新点

- 对比了不同的基因重组方法, 以及不同的自然选择方式 (锦标赛和轮盘赌), 选择了针对 TSP 问题中的更优方式
- 针对特定的 TSP 问题, 在基因重组中选择了独特的排列交换方法
- 对遗传算法的各参数进行了消融研究, 得出了较优的结果

3 参考文献

https://blog.csdn.net/wuyzh39/article/details/130072858