计算智能大作业 郑宇鹏

旅行商问题

10 个以上城市的旅行商问题，城市位置可以自己设定或参考现有旅行商问题，求最短路径。

1. 问题描述

给定 N 个城市，这 N 个城市记为$C = \{c_1, \dots, c_n\}$，他们之间的距离可表示为 N * N 的矩阵 D，形如$D = \begin{bmatrix} d_{11} & \cdots & d_{1N} \\ \vdots & \ddots & \vdots \\ d_{N1} & \cdots & d_{NN} \end{bmatrix}$，求把每一个城市都走一遍并回到原点的最短路径长度$d$。路径的限制是每个城市只能拜访一次，而且最后要回到原来出发的城市。路径的选择目标是要求所选的路径路程为所有路径中的最小值。

从图论的角度可以认为是给定一个有 N 个点的带权完全图 G，我们需要找到它权值和最小的哈密顿回路。

2. 遗传算法求解

(1) 算法设计

a) 初始化城市：首先都需要初始化城市图及计算城市距离矩阵。即用 N * 2 的数组存储 N 个城市坐标，N * N 的数组存储任意两城市间距

```
def init_graph(args):
    cities = np.random.randint(0 , args.n, size=(args.n, 2))
    dist = np.zeros([args.n, args.n])

    for i in range(args.n):
        for j in range(args.n):
            if i != j:
                dist[i,j] = get_dist(cities, i, j)
    return cities, dist
```

b) 初始化相应数量的个体，并初始化随机基因序列

```
def init_pop(args, dist):
    pop = []
    for _ in range(args.pn):
        ind = {}
        gene = [i for i in range(args.n)]
        random.shuffle(gene)
        ind['gene'] = gene
        ind['fit'] = compute_fitness(args, gene, dist)
        pop.append(ind)
    return pop
```

c) 交叉

为了得到父代随机交换产生子代的效果，首先对种群打乱排序，然后将种群中相邻的个体两辆配对（如 i 和 i + 1, i + 2 和 i + 3）作为父代。随后，随机生成父代间交换位置进行交换，每一对父代组合得到两个子代个体。并计算每个子代个体的适应度。

```
# random cross
random.shuffle(self.pop)
for i in range(0, self.args.pn - 1, 2):

    gene1 = self.pop[i]['gene'].copy()
    gene2 = self.pop[i + 1]['gene'].copy()

    begin_idx = random.randint(0, self.args.n - 2)
    end_idx = random.randint(begin_idx, self.args.n - 1)

    pos1_r = {value: idx for idx, value in enumerate(gene1)}
    pos2_r = {value: idx for idx, value in enumerate(gene2)}


    if random.random() < self.args.cross_prob:
        for j in range(begin_idx, end_idx):
            pos1, pos2 = pos1_r[gene2[j]], pos2_r[gene1[j]]
            gene1[j], gene1[pos1] = gene1[pos1], gene1[j]
            gene2[j], gene2[pos2] = gene2[pos2], gene2[j]

            pos1_r[gene1[j]], pos1_r[gene2[j]] = pos1, j
            pos2_r[gene1[j]], pos2_r[gene2[j]] = j, pos2

    fit1 = compute_fitness(self.args, gene1, self.dist)
    fit2 = compute_fitness(self.args, gene2, self.dist)

    new_gene1 = {}
    new_gene1['fit'] = fit1
    new_gene1['gene'] = gene1

    new_gene2 = {}
    new_gene2['fit'] = fit2
    new_gene2['gene'] = gene2

    self.new_ind.append(new_gene1)
    self.new_ind.append(new_gene2)
```

d) 变异

对得到的新个体进行变异操作，首先根据参数判断是否随机变异，随后选择变异序列，对该序列的基因进行翻转（.inverse()）操作。

```
# random variation
for ind in self.new_ind:
    if random.random() < self.args.variation_prob:
        old_gene = ind['gene'].copy()

        begin_idx = random.randint(0, self.args.n - 2)
        end_idx = random.randint(begin_idx, self.args.n - 1)

        gene_variation = old_gene[begin_idx: end_idx]
        gene_variation.reverse()

        ind["gene"] = old_gene[:begin_idx] + gene_variation + old_gene[end_idx:]

self.pop += self.new_ind
```

e) 选择

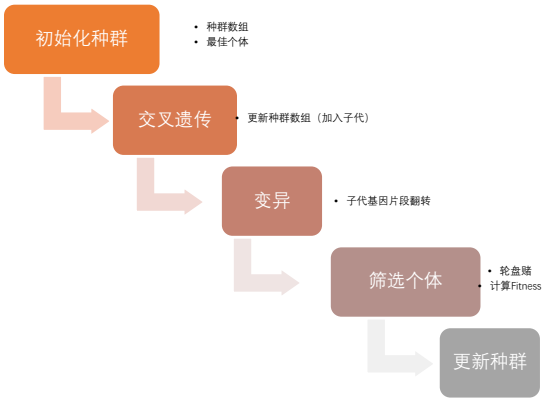
在竞争选择保留下来的子代个体时，选用了锦标赛和轮盘赌两种算法。选择每代保留下的个体。轮盘赌法将每个个体的适应度的倒数（因为适应度是距离之和，越小越好，因此选用倒数）作为每个个体被选择的概率。锦标赛法随机从群体中选出 gn 个个体，然后挑出其中的最佳个体 winner。

```
# select
if self.args.choice == "championship":
    group_winner = self.args.pn // self.args.gn
    winners = []
    for i in range(self.args.gn):
        group = []
        for j in range(self.args.gn):
            player = random.choice(self.pop)
            group.append(player)
        group = get_sort(group)
        winners += group[:group_winner]
    self.pop = winners

elif self.args.choice == 'roulette':
    self.cities_prob = [(1 / self.pop[i]["fit"]) for i in range(len(self.pop))]

    total_prob = sum(self.cities_prob)
    roulette = []
    new_pop = []

    while len(new_pop) < 30:
        temp_prob = random.uniform(0.0, total_prob)
        for j in range(len(self.pop)):
            temp_prob -= self.cities_prob[j]
            if temp_prob < 0 and j not in roulette:
                roulette.append(j)
                new_pop.append(self.pop[j])
                break

    self.pop = new_pop
```

f) 更新下一代

更新种群数组

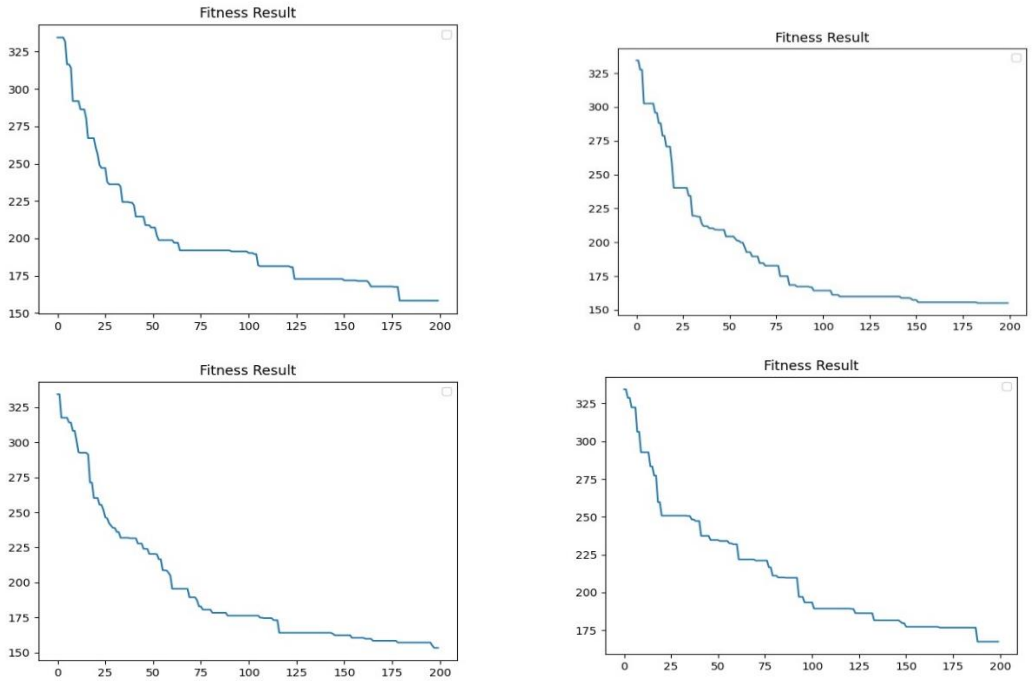
整个过程流程图如下所示，因此，只需要维持种群数组这一个全局变量，并在各个步骤实现相应的更新操作即可。



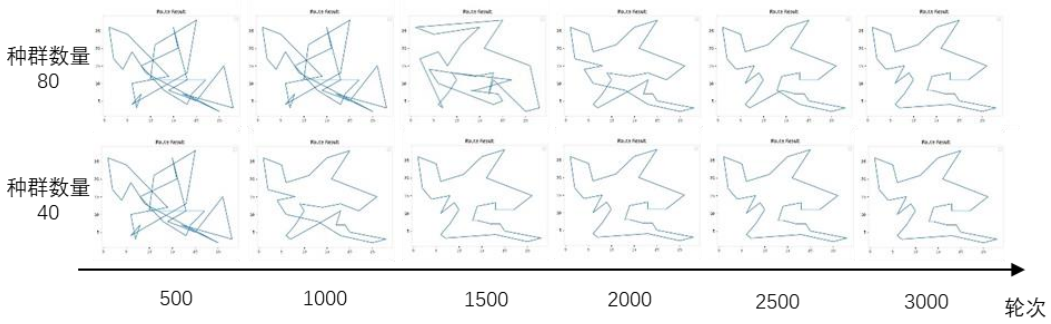
(2) 实验分析

a) 种群数量

固定城市数量 30，设置种群数量为 30，40，50，60。如下图所示，随着种群数量增加，Fitness 下降越发缓慢，运行效率降低。

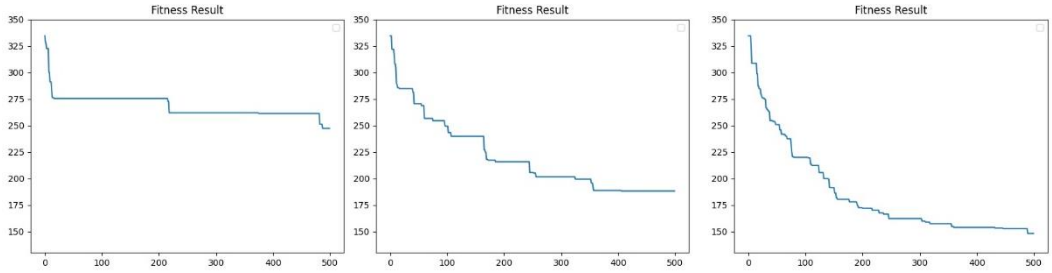


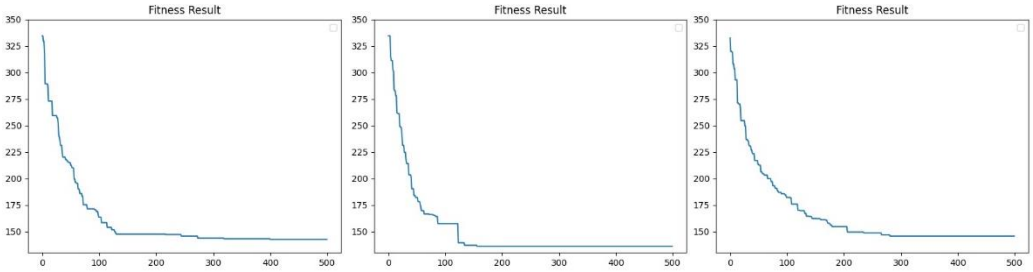
种群数量为 80 和 40 时，每 1000 迭代轮次的路线更新结果如下图所示。其中上面的序列是种群数量为 80 的路线更新，下图为 40 的路线更新。可以看出前者效率明显低于后者。



b) 变异概率

固定城市 30 个，个体数量 40 个，将变异概率分别设置为 0.001，0.01，0.1，0.3，0.5，0.7





由上图可见，概率过小时，新个体产生速度慢，Fitness 下降慢，优化进程缓慢。随着概率增加，新个体产生速度增加，Fitness 下降加快。当概率增大至 0.5 时，Fitness 下降最快。随后，再增大变异概率，Fitness 下降变慢，因为优良个体被破坏。

3. Hopfield 神经网络求解

(1) Hopfield 理论

Hopfield 神经网络的基础构架是网络中每一个神经元的输出乘以某些权值后送到所有神经元的输入端，从而形成一个全反馈结构。Hopfield 网络的一大特点就是可以用电路来表示。Hopfield 有如下特点：

1) 每个神经元既是输入也是输出，构成单层全连接递归网络；
2) 网络的权重是在搭建网络时按照约束规则计算出来的，且在迭代过程中不再改变；
3) 网络的状态是随时间的变化而变化的；
4) 用能量函数来判断网络迭代的稳定性，当能量函数达到极小值时网络收敛；
5) 网络的解是网络收敛时，各个神经元的状态集合。

Hopfield 可分为连续型（运行在连续时间域，简称 HM）和离散型（运行在连续时间域，

简称离散 HNN）。连续型适合解决组合优化问题，离散型适合处理联想记忆问题。因此，在 TSP 问题中，采用连续型 HNN（HM）来求解。在 TSP 问题中，有如下约束规则：

1) 每个城市必须去一次；

2) 每个城市只能去一次；

3) 一次只能访问一个城市。

将 N 个城市记为$C=\{c_1,\dots,c_n\}$，他们之间的距离矩阵 $D=\begin{bmatrix} d_{11} & \cdots & d_{1N} \\ \vdots & \ddots & \vdots \\ d_{N1} & \cdots & d_{NN} \end{bmatrix}$，构造一个 N×N 的矩阵$\boldsymbol{X}=[\boldsymbol{x_{ij}}]$

$$x_{ij}=\begin{cases}1; & \text{如果旅行的第 j 个城市是 } C_i \\ 0; & \text{如果旅行的第 j 个城市不是 } C_i\end{cases}$$

则可以把上述 3 条约束规则表示为：

$$\sum_{i=1}^{N}\sum_{j=1}^{N}x_{ij}=N$$

$$\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k\neq j}x_{ij}x_{ik}=0$$

$$\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k\neq i}x_{ij}x_{kj}=0$$

由此，可以得到能量函数为：

$$E=\frac{\rho_1}{2}\left(\sum_{i=1}^{N}\sum_{j=1}^{N}x_{ij}-N\right)^2+\rho_2\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k\neq j}x_{ij}x_{ik}+\rho_3\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k\neq i}x_{ij}x_{kj}+\rho_4\sum_{i}^{N}\sum_{j\neq i}^{N}\sum_{k=1}^{N}d_{ij}x_{ik}x_{j,k+1}+\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_{ij}\int_0^{x_{ij}}f^{-1}(v)dv$$

其中每一项分别表示：每个城市必须去一次（当且仅当所有的 N 个城市一共被访问 N 次时才取得最小值 0）、每个城市只去一次（当且仅当所每个城市仅被访问 1 次时才取得最小值 0）、每一时刻只经过一个城市（当且仅当所每次仅访问一个城市时才取得最小值 0）、路径总开销（表示按照当前的访问路线的安排，所需要走的路径的总长度）、HM 本身要求。

对能量函数求导得到：

$$\frac{\partial E}{\partial x_{ij}}=\rho_1\left(\sum_{i=1}^{N}\sum_{j=1}^{N}x_{ij}-N\right)+\rho_2\sum_{k\neq j}x_{ik}+\rho_3\sum_{k\neq i}x_{kj}+\rho_4\sum_{k\neq i}d_{ik}x_{k,j+1}+\alpha_{ij}f^{-1}(x_{ij})$$

要保持稳定性，则应有：

$$\frac{du_{ij}}{dt}=-\frac{\partial E}{\partial x_{ij}}$$

于是 Hopfield 的标准网络形式为：

$$\begin{cases} \dfrac{du_{ij}}{dt} = -\alpha_{ij}u_{ij} + \sum_{k=1}^{N}\sum_{l=1}^{N} w_{ij,kl}x_{kl} + \theta \\ x_{ij} = f(u_{ij}) \end{cases}$$

在实际应用中，Hopfield 存在神经网络优化计算的普遍的问题，比如

1) 解的不稳定性，输入的微弱的改动会导致结果差异很大；

2) 参数难以确定，需要经过严格的参数调整以达到较好的性能；

3) 难以保证最优解，因为能量函数可能存在多个局部最优解；

4) 只能解决一类特殊形式的优化问题。

(2) Hopfield 的实现：

计算输入状态的增量 du

```
def get_du(self, v):
    a = np.sum(v, axis=0) - 1
    b = np.sum(v, axis=1) - 1
    t1 = np.zeros((self.args.n, self.args.n))
    t2 = np.zeros((self.args.n, self.args.n))
    for i in range(self.args.n):
        for j in range(self.args.n):
            t1[i, j] = a[j]
    for i in range(self.args.n):
        for j in range(self.args.n):
            t2[j, i] = b[j]

    tmp_v = v.copy()
    for i in range(len(tmp_v) - 1):
        tmp_v[:, i] = tmp_v[: ,i + 1]
    tmp_v[:, -1] = v[:, 0]
    c = np.dot(self.dist, tmp_v)

    return -self.args.n * self.args.n * (t1 + t2) - (self.args.n / 2) * c
```

按照能量函数计算能量值

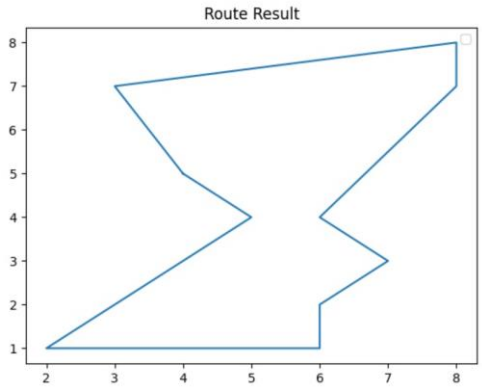
```
def get_energy(self, v):
    item1 = np.square(np.sum(v, axis=0) - 1)
    item2 = np.square(np.sum(v, axis=1) - 1)
    tmp_v = v.copy()
    for i in range(len(tmp_v) - 1):
        tmp_v[:, i] = tmp_v[: ,i + 1]
    tmp_v[:, -1] = v[:, 0]
    item3 = v * self.dist * tmp_v

    t1 = np.sum(item1)
    t2 = np.sum(item2)
    t3 = np.sum(item3)
```

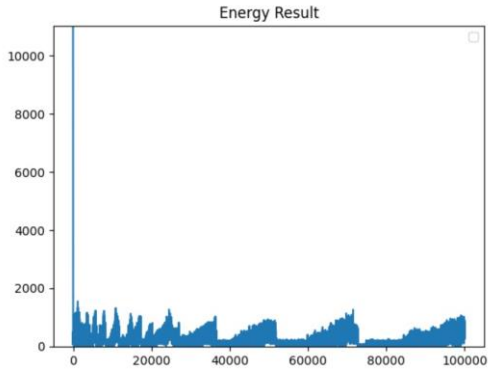
由输入状态增量更新输入状态和由 sigmoid 函数更新 HNN 下个时刻的输出状态

```python
def update_u(self, u, du):
    return u + du * self.eta

def update_v(self, u):
    return 1 / 2 * (1 + np.tanh(u / self.u0))
```

(3) 实验结果：

给定城市数量为 10，Hopfield 算法可以在 80000 迭代轮次找到最优解





具体的代码请见：https://github.com/ucaszyp/hw-GA