用遗传算法求解中国省会城市 TSP 问题

SA19023005　景军元

摘要：

TSP 问题是一个著名的组合优化问题。该问题具有 NPC 计算复杂性。迄今为止，这类问题中没有一个找到有效算法。倾向于接受 NP 完全问题（NP-Complete 或 NPC）和 NP 难题（NP-Hard 或 NPH）不存在有效算法这一猜想，认为这类问题的大型实例不能用精确算法求解，必须寻求这类问题的有效的近似算法。

遗传算法（Genetic Algorithm，GA）最早是由美国的 John holland 于 20 世纪 70 年代提出，该算法是根据大自然中生物体进化规律而设计提出的。是模拟达尔文生物进化论的自然选择和遗传学机理的生物进化过程的计算模型，是一种通过模拟自然进化过程搜索最优解的方法。该方法能够较为有效的求解 TSP 问题的近似最优解。

本文首先介绍了遗传算法的基本原理，之后采用遗传算法求解中国 34 个省会城市的 TSP 问题，最后将最终的结果可视化。实验结果表明，该算法能够较为高效的处理 TSP 问题，得到 TSP 问题的近似最优解。

1 引言：

旅行商问题，即 TSP 问题（Traveling Salesman Problem）又译为旅行推销员问题、货郎担问题，是数学领域中著名问题之一。假设有一个旅行商人要拜访 n 个城市，他必须选择所要走的路径，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得的路径路程为所有路径之中的最小值。

TSP 问题是一个组合优化问题。该问题可以被证明具有 NPC 计算复杂性。Edmonds，Cook 和 Karp 等人发现，这批难题有一个值得注意的性质，对其中一个问题存在有效算法时，每个问题都会有有效算法。迄今为止，这类问题中没有一个找到有效算法。倾向于接受 NP 完全问题（NP-Complete 或 NPC）和 NP 难题（NP-Hard 或 NPH）不存在有效算法这一猜想，认为这类问题的大型实例不能用精确算法求解，必须寻求这类问题的有效的近似算法。

遗传算法（Genetic Algorithm，GA）最早是由美国的 John holland 于 20 世纪 70 年代提出,该算法是根据大自然中生物体进化规律而设计提出的。是模拟达尔文生物进化论的自然选择和遗传学机理的生物进化过程的计算模型，是一种通过模拟自然进化过程搜索最优解的方法。该算法通过数学的方式,利用计算机仿真运算,将问题的求解过程转换成类似生物进化中的染色体基因的交叉、变异等过程。在求解较为复杂的组合优化问题时,相对一些常规的优化算法,通常能够较快地获得较好的优化结果。遗传算法已被人们广泛地应用于组合优化、机器学习、信号处理、自适应控制和人工生命等领域。

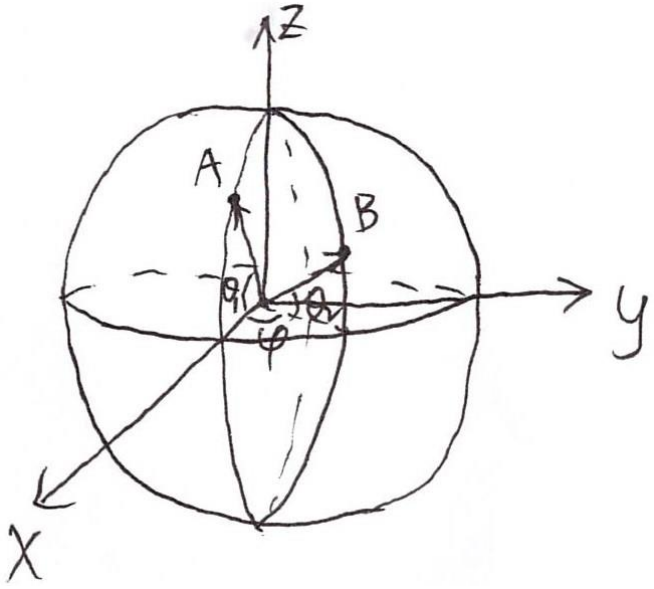
2 基本原理

2.1 球面距离

如下图所示，A、B 为球面上两点，θ_1表示 A 点的纬度，θ_2表示 B 点的纬度，φ表示 A、B 两点的经度差（均以弧度表示）。则建立如图所示的直角坐标系，A、B 两点可分别用坐标表示为：

$$\vec{A} = (R\cos\theta_1, 0, R\sin\theta_1)$$

$$\vec{B} = (R\cos\theta_2\cos\varphi, R\cos\theta_2\sin\varphi, R\sin\theta_2)$$



则$\vec{A}$和$\vec{B}$之间的夹角可表示为：

$$\langle\vec{A},\vec{B}\rangle = \cos^{-1}\frac{\vec{A}\cdot\vec{B}}{|\vec{A}||\vec{B}|}$$

根据$\vec{A}$和$\vec{B}$的夹角就可以得到 A、B 两点之间的球面距离为：

$$dist = R * \langle\vec{A},\vec{B}\rangle$$

假设地球为一个标准的球形，查阅相关资料得到地球的平均半径为 6371km，则根据上述公式，在已知经纬度的情况下可以求出任意两地之间的距离。

2.2 遗传算法

2.2.1 运算过程

遗传算法的基本运算过程如下：

（1）初始化：设置进化代数计数器 t=0，设置最大进化代数 T，随机生成

M 个个体作为初始群体 P(0)。

（2）个体评价：计算群体 P(t)中各个个体的适应度。

（3）选择运算：将选择算子作用于群体。选择的目的是把优化的个体直接遗传到下一代或通过配对交叉产生新的个体再遗传到下一代。选择操作是建立在群体中个体的适应度评估基础上的。

（4）交叉运算：将交叉算子作用于群体。遗传算法中起核心作用的就是交叉算子。

（5）变异运算：将变异算子作用于群体。即是对群体中的个体串的某些基因座上的基因值作变动。群体 P(t)经过选择、交叉、变异运算之后得到下一代群体 P(t+1)。

（6）终止条件判断：若 t=T,则以进化过程中所得到的具有最大适应度个体作为最优解输出，终止计算。

遗传操作包括以下三个基本遗传算子(genetic operator)：选择(selection)；交叉(crossover)；变异(mutation)。

2.2.2 编码

由于遗传算法不能直接处理问题空间的参数,因此必须通过编码将要求解的问题表示成遗传空间的染色体或者个体。这一转换操作就叫做编码，也可以称作（问题的）表示（representation)。

评估编码策略常采用以下 3 个规范：

a)完备性(completeness):问题空间中的所有点(候选解)都能作为 GA 空间中的点（染色体）表现。

b)健全性(soundness): GA 空间中的染色体能对应所有问题空间中的候选解。

c)非冗余性(nonredundancy):染色体和候选解一一对应。

2.2.3 适应度函数

进化论中的适应度，是表示某一个体对环境的适应能力，也表示该个体繁殖后代的能力。遗传算法的适应度函数也叫评价函数，是用来判断群体中的个体的优劣程度的指标，它是根据所求问题的目标函数来进行评估的。

遗传算法在搜索进化过程中一般不需要其他外部信息，仅用评估函数来评估个体或解的优劣，并作为以后遗传操作的依据。由于遗传算法中，适应度函数要比较排序并在此基础上计算选择概率，所以适应度函数的值要取正值。由此可见，在不少场合，将目标函数映射成求最大值形式且函数值非负的适应度函数是必要的。

适应度函数的设计主要满足以下条件：

a）单值、连续、非负、最大化

b) 合理、一致性

c）计算量小

d）通用性强。

在具体应用中，适应度函数的设计要结合求解问题本身的要求而定。适应度函数设计直接影响到遗传算法的性能。

2.2.4 初始群体选取

遗传算法中初始群体中的个体是随机产生的。一般来讲，初始群体的设定可采取如下的策略:

a)根据问题固有知识，设法把握最优解所占空间在整个问题空间中的分布范围，然后，在此分布范围内设定初始群体。

b)先随机生成一定数目的个体，然后从中挑出最好的个体加到初始群体中。这种过程不断迭代，直到初始群体中个体数达到了预先确定的规模。

2.2.5 选择

从群体中选择优胜的个体，淘汰劣质个体的操作叫选择。选择算子有时又称为再生算子(reproduction operator)。选择的目的是把优化的个体(或解)直接遗传到下一代或通过配对交叉产生新的个体再遗传到下一代。选择操作是建立在群体中个体的适应度评估基础上的，常用的选择算子有以下几种：适应度比例方法、随机遍历抽样法、局部选择法。

2.2.6 交叉

在自然界生物进化过程中起核心作用的是生物遗传基因的重组(加上变异)。同样，遗传算法中起核心作用的是遗传操作的交叉算子。所谓交叉是指把两个父代个体的部分结构加以替换重组而生成新个体的操作。通过交叉，遗传算法的搜索能力得以飞跃提高。

2.2.7 变异

变异算子的基本内容是对群体中的个体串的某些基因座上的基因值作变动。依据个体编码表示方法的不同，可以有以下的算法：

a)实值变异。

b)二进制变异。

一般来说，变异算子操作的基本步骤如下：

a)对群中所有个体以事先设定的变异概率判断是否进行变异

b)对进行变异的个体随机选择变异位进行变异。

遗传算法引入变异的目的有两个:一是使遗传算法具有局部的随机搜索能力。当遗传算法通过交叉算子已接近最优解邻域时，利用变异算子的这种局部随机搜索能力可以加速向最优解收敛。显然，此种情况下的变异概率应取较小值，否则接近最优解的积木块会因变异而遭到破坏。二是使遗传算法可维持群体多样性，以防止出现未成熟收敛现象。此时收敛概率应取较大值。

2.2.8 终止条件

当最优个体的适应度达到给定的阈值，或者最优个体的适应度和群体适应度不再上升时，或者迭代次数达到预设的代数时，算法终止。预设的代数一般设置为100-500代。

3 实验方法

3.1 问题描述

已知中国34个省会城市的位置，现有一人从某个城市出发，不重不漏地遍

历 34 个城市，最终返回出发城市，求如何安排访问次序，使得总路程最短。

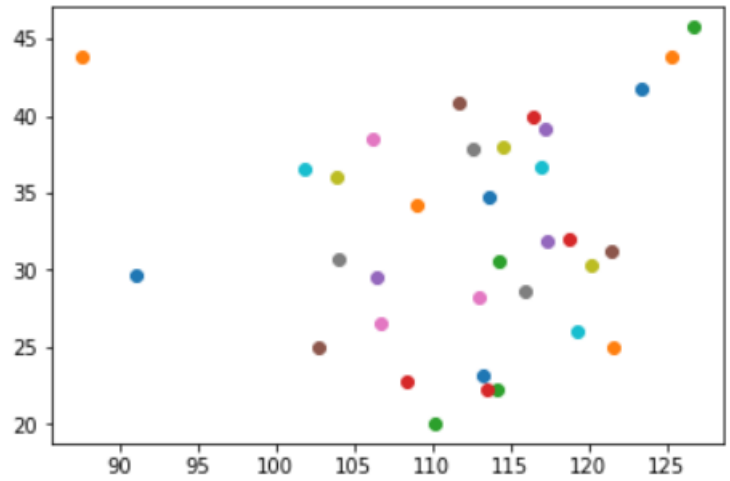
3.2 数据处理

参考 https://blog.csdn.net/myspace_word/article/details/88367134，得到每个城市的经纬度如下图所示：

```
all_cities = {'Shenyang': [123.429092, 41.796768],
              'Changchun': [125.324501, 43.886841],
              'Harbin': [126.642464, 45.756966],
              'Beijing': [116.405289, 39.904987],
              'Tianjin': [117.190186, 39.125595],
              'Hohhot': [111.751990, 40.841490],
              'Yinchuan': [106.232480, 38.486440],
              'Taiyuan': [112.549248, 37.857014],
              'Shijiazhuang': [114.502464, 38.045475],
              'Jinan': [117.000923, 36.675808],
              'Zhengzhou': [113.665413, 34.757977],
              'Xian': [108.948021, 34.263161],
              'Wuhan': [114.298569, 30.584354],
              'Nanjing': [118.76741, 32.041546],
              'Hefei': [117.283043, 31.861191],
              'Shanghai': [121.472641, 31.231707],
              'Changsha': [112.982277, 28.19409],
              'Nanchang': [115.892151, 28.676493],
              'Hangzhou': [120.15358, 30.287458],
              'Fuzhou': [119.306236, 26.075302],
              'Guangzhou': [113.28064, 23.125177],
              'Taipei': [121.5200760, 25.0307240],
              'Haikou': [110.199890, 20.044220],
              'Nanning': [108.320007, 22.82402],
              'Chongqing': [106.504959, 29.533155],
              'Kunming': [102.71225, 25.040609],
              'Guizhou': [106.713478, 26.578342],
              'Chengdu': [104.065735, 30.659462],
              'Lanzhou': [103.834170, 36.061380],
              'Xining': [101.777820, 36.617290],
              'Lhasa': [91.11450, 29.644150],
              'Urumqi': [87.616880, 43.826630],
              'Hongkong': [114.165460, 22.275340],
              'Macao': [113.549130, 22.198750]}
```

其中字典索引为城市名，列表的第一个元素为经度（东经），第二个元素为纬度（北纬）。

将上述坐标可视化之后，得到下图的省会城市分布：



接下来根据之前所述的求球面距离的方法得到城市间的距离（这里为了简化问题，将城市间的距离按直线距离计算，实际路程会受到地形、公路等的限制）：

```
dist = {}
for city_1 in all_cities:
    for city_2 in all_cities:
        if city_1 == city_2:
            continue
        longi_1 = all_cities[city_1][0]*np.pi/180
        lati_1 = all_cities[city_1][1]*np.pi/180
        longi_2 = all_cities[city_2][0]*np.pi/180
        lati_2 = all_cities[city_2][1]*np.pi/180
        a1 = np.array([np.cos(lati_1),0,np.sin(lati_1)])
        a2 = np.array([np.cos(lati_2)*np.cos(abs(longi_1-longi_2)),np.cos(lati_2)*np.sin(abs(longi_1-longi_2)),np.sin(lati_2)])
        dist[(city_1,city_2)] = np.arccos(np.dot(a1,a2))*6371
```

3.3 算法步骤

（1）生成初始种群

```
def generate_random_agent():
    """
        生成单个随机路线
    """
    new_random_agent = list(all_cities.keys())
    random.shuffle(new_random_agent)
    return tuple(new_random_agent)

def generate_random_population(pop_size):
    """
        生成pop_size个随机路线
    """

    random_population = []
    for agent in range(pop_size):
        random_population.append(generate_random_agent())
    return random_population
```

（2）计算个体适应度（总路程越短，适应度越高）

```python
def compute_fitness(solution):
    """
        计算个体适应度
        返回当前路线的总距离
        遗传算法会更倾向于选择距离较短的路线
    """

    solution_fitness = 0.0

    for index in range(len(solution)):
        city_1 = solution[index - 1]
        city_2 = solution[index]
        solution_fitness += dist[(city_1, city_2)]

    return solution_fitness
```

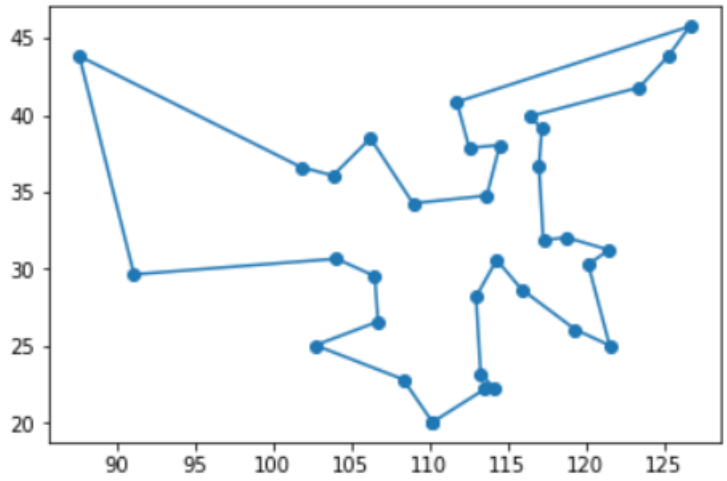
（3）产生后代

```python
def mutate_agent(agent_genome, max_mutations=3):
    """
        对路线产生1至max_mutations个点的突变
    """

    agent_genome = list(agent_genome)
    num_mutations = random.randint(1, max_mutations)

    for mutation in range(num_mutations):
        swap_index1 = random.randint(0, len(agent_genome) - 1)
        swap_index2 = swap_index1

        while swap_index1 == swap_index2:
            swap_index2 = random.randint(0, len(agent_genome) - 1)

        agent_genome[swap_index1], agent_genome[swap_index2] = agent_genome[swap_index2], agent_genome[swap_index1]

    return tuple(agent_genome)

def shuffle_mutation(agent_genome):
    """
        对路线进行一次打乱操作
    """

    agent_genome = list(agent_genome)

    start_index = random.randint(0, len(agent_genome) - 1)
    length = random.randint(2, 20)

    genome_subset = agent_genome[start_index:start_index + length]
    agent_genome = agent_genome[:start_index] + agent_genome[start_index + length:]

    insert_index = random.randint(0, len(agent_genome) + len(genome_subset) - 1)
    agent_genome = agent_genome[:insert_index] + genome_subset + agent_genome[insert_index:]

    return tuple(agent_genome)
```

4 实验结果

设置迭代次数为 5000 次，种群个数为 100，最终得到的最优路线为：('Haikou', 'Macao', 'Hongkong', 'Guangzhou', 'Changsha', 'Wuhan', 'Nanchang', 'Fuzhou', 'Taipei', 'Hangzhou', 'Shanghai', 'Nanjing', 'Hefei', 'Jinan', 'Tianjin', 'Beijing', 'Shenyang', 'Changchun', 'Harbin', 'Hohhot', 'Taiyuan', 'Shijiazhuang', 'Zhengzhou', 'Xian', 'Yinchuan', 'Lanzhou', 'Xining', 'Urumqi', 'Lhasa', 'Chengdu', 'Chongqing', 'Guizhou', 'Kunming', 'Nanning')

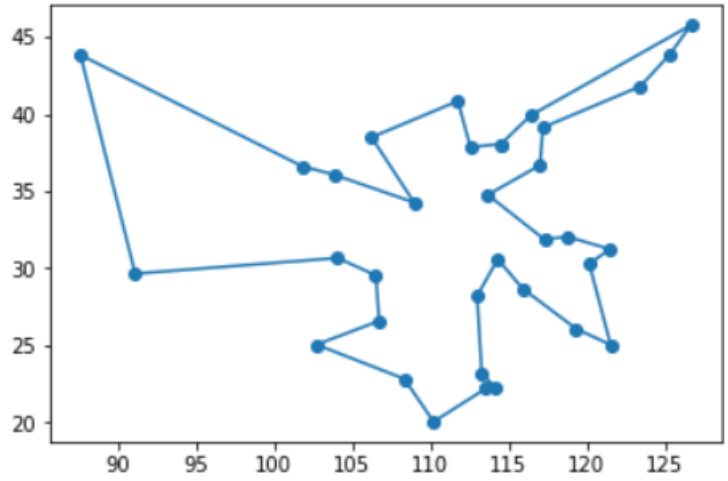
如下图所示：



总距离为 15689km。

设置迭代次数为 5000 次，种群个数为 200，最终得到的最优路线为：
('Shijiazhuang', 'Beijing', 'Harbin', 'Changchun', 'Shenyang', 'Tianjin', 'Jinan', 'Zhengzhou', 'Hefei', 'Nanjing', 'Shanghai', 'Hangzhou', 'Taipei', 'Fuzhou', 'Nanchang', 'Wuhan', 'Changsha', 'Guangzhou', 'Hongkong', 'Macao', 'Haikou', 'Nanning', 'Kunming', 'Guizhou', 'Chongqing', 'Chengdu', 'Lhasa', 'Urumqi', 'Xining', 'Lanzhou', 'Xian', 'Yinchuan', 'Hohhot', 'Taiyuan')

如下图所示：



总距离为 15751km。

5 总结和反思

多次改变实验参数，可以发现，尽管每次得到的解相对来说已经较好，但并不总是最优解，甚至不知道最优解到底是哪一个，这也是 NPC 问题难以解决的地方。但是，遗传算法能够在短时间内得到一个相对较优的解，这也正是遗传算法在处理优化问题上优越之处。