智能优化技术结课报告

基于智能优化算法解决离散优化问题

院（系）：计算机学院

专　　业：计算机科学与技术

姓　　名：常文瀚

班级学号：191181

指导教师：龚文引

2021 年 5 月 20

目录

第一章 智能优化算法解决离散问题 1

1.1 问题描述 1

1.2 算法描述 2

1.2.1 遗传算法 2

1.2.2 算法流程 2

1.2.3 算法参数 2

1.3 实验 3

1.3.1 实验结果 3

1.3.2 结果分析 7

第二章 总结与参考 8

2.1 总结 8

2.2 参考与引用 8

附录一 核心代码 9

基于智能优化算法解决离散优化问题

常文瀚

（中国地质大学，武汉）

摘要 优化问题是指在满足一定条件下，在众多方案或参数值中寻找最优方案或参数值，以使得某个或多个功能指标达到最优，或使系统的某些性能指标达到最大值或最小值。优化问题广泛地存在于信号处理、图像处理、生产调度、任务分配、模式识别、自动控制和机械设计等众多领域。优化方法是一种以数学为基础，用于求解各种优化问题的应用技术。各种优化方法在上述领域得到了广泛应用，并且已经产生了巨大的经济效益和社会效益。实践证明，通过优化方法，能够提高系统效率，降低能耗，合理地利用资源，并且随着处理对象规模的增加，这种效果也会更加明显。本文通过研究 TSP 问题和求函数最小值问题。对模拟退火算法和遗传算法进行了性能比较。

关键词 TSP 问题，遗传算法

第一章 智能优化算法解决连续问题

1.1 问题描述

TSP 问题的模型是简单而易于描述的。实际应用中，像印刷电路板工艺这样的应用可能是和模型比较接近的但是模型中的城市之间的距离代表的是某个解的耗费，实际中不一定是欧氏距离,有可能点与点之间的耗费需要另外用权值给出。而且在实际中，两个城市之间的消耗不一定是对称的，例如乘船到另一城市和返回的费用可能是不同的。

这里对 TSP 问题模型进行简化，在平面内随机生成了 50 个点，用它们来代表城市。假设每个城市和其它任意一个城市之间都直接相连。

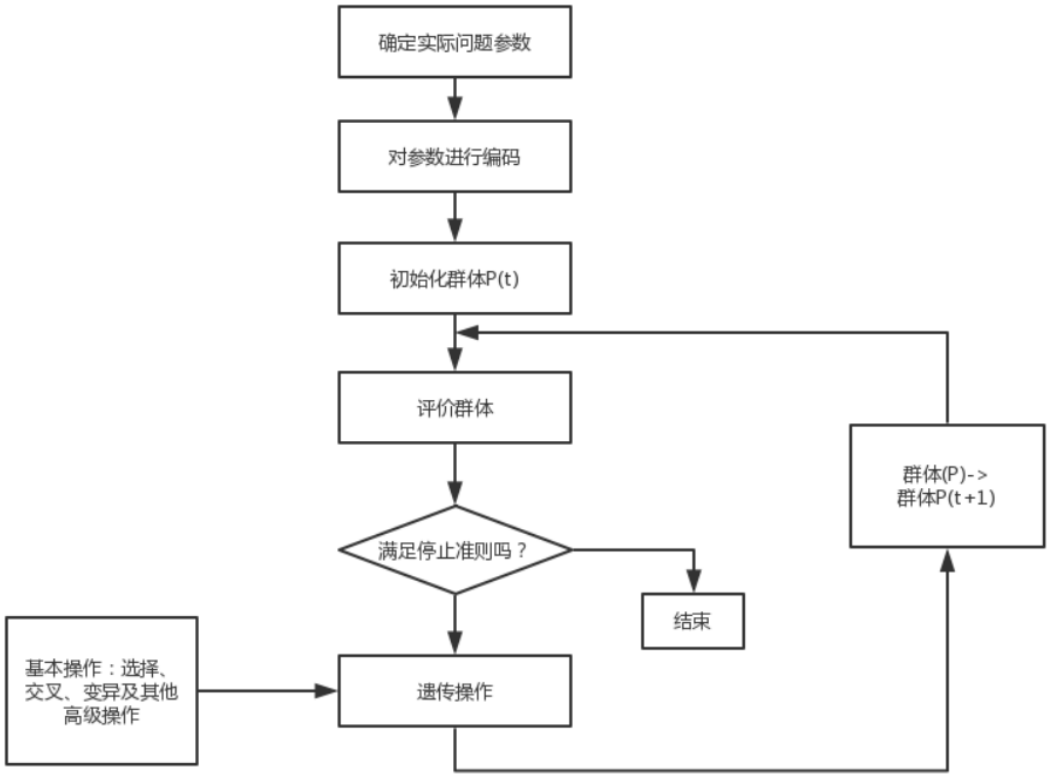
1.2 算法描述

1.2.1 遗传算法

用路径顺序进行编码，比如 5 个城市路径：[4,2,3,0,1]。

1.2.2 算法流程

算法流程与前一问题中描述的流程基本相同。但对于交叉操作，我们尝试了部分映射交叉和单位置次序交叉两种方法，实验发现单位制次序交叉方法的性能更优。变异操作中，我们对染色体中的每个基因(某城市)按变异概率随机与路径中另-城市交换次序。与函数优化问题相同，我们在该问题中同样采用保优操作。



图（1）遗传算法流程

1.2.3 算法参数

表（1）遗传算法实验参数

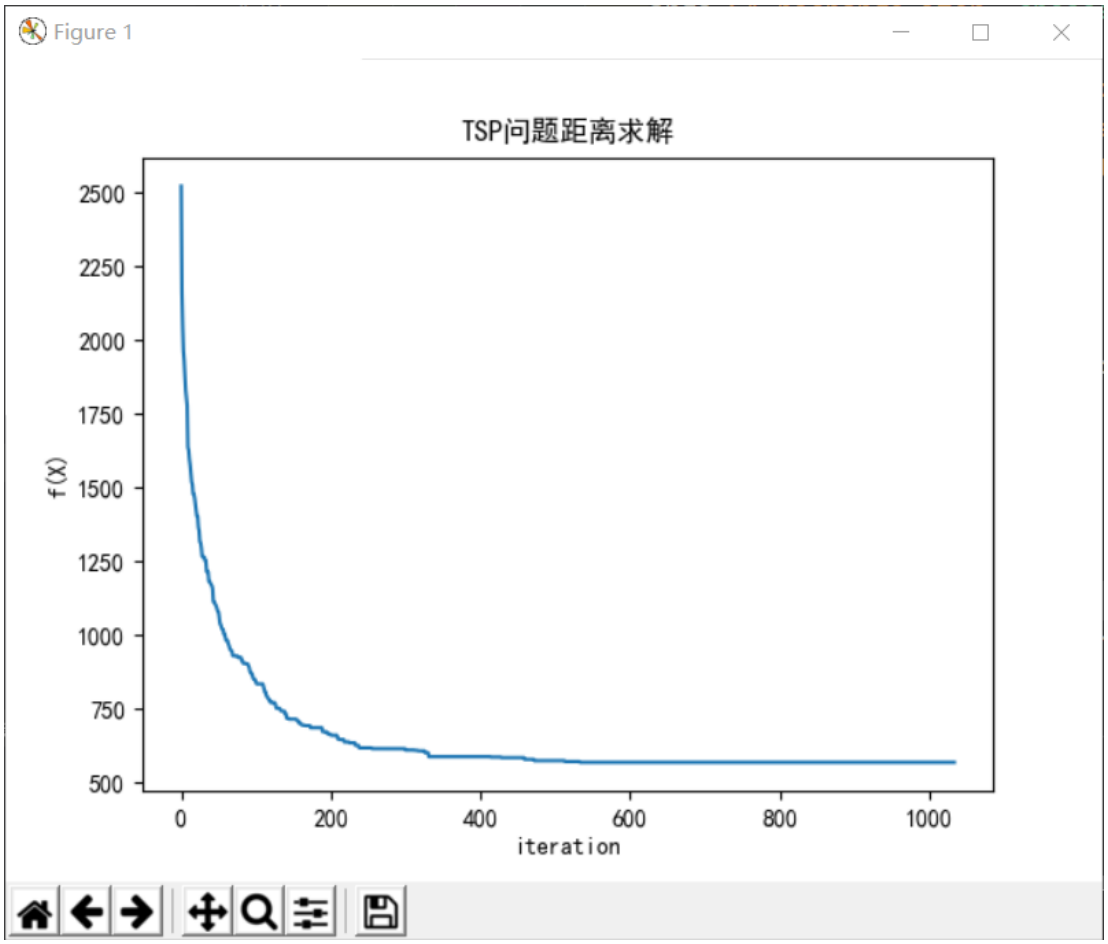
参数名称	参数含义	参数值
/	最大迭代次数	2000
GA_N	种群规模	60
GA_C	交叉概率	0.95
GA_M	变异概率	0.2

GA_nochange_iter	保优值保持不变提前结束迭代的回合数	500
GA_last_gl	产生新种群时先从上一代种群选出一部分较优个体	0.2

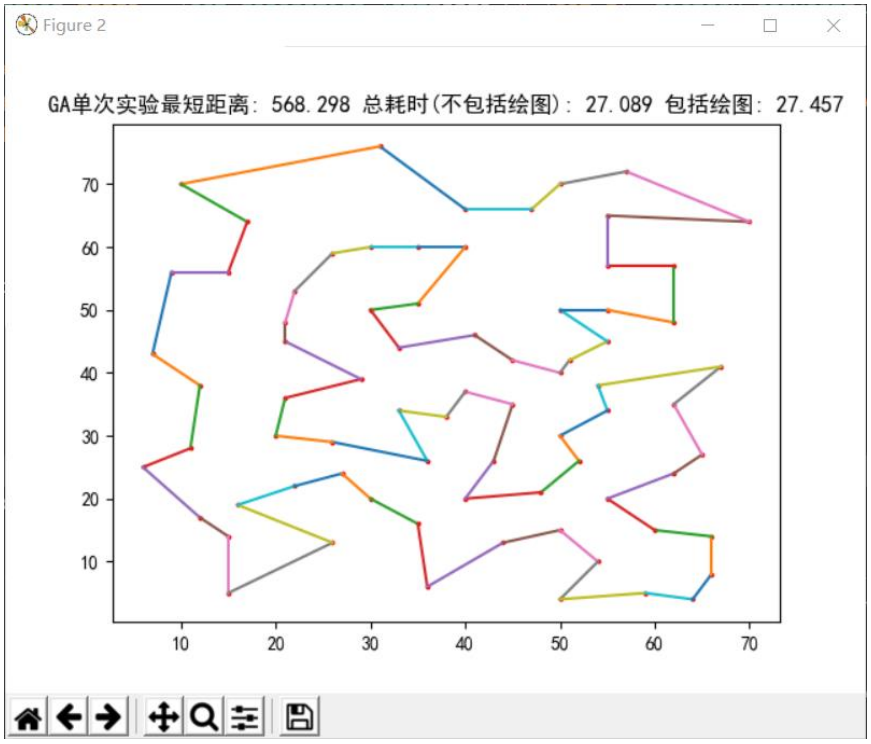
1.3 实验

1.3.1 实验结果

（1） 单次运行结果

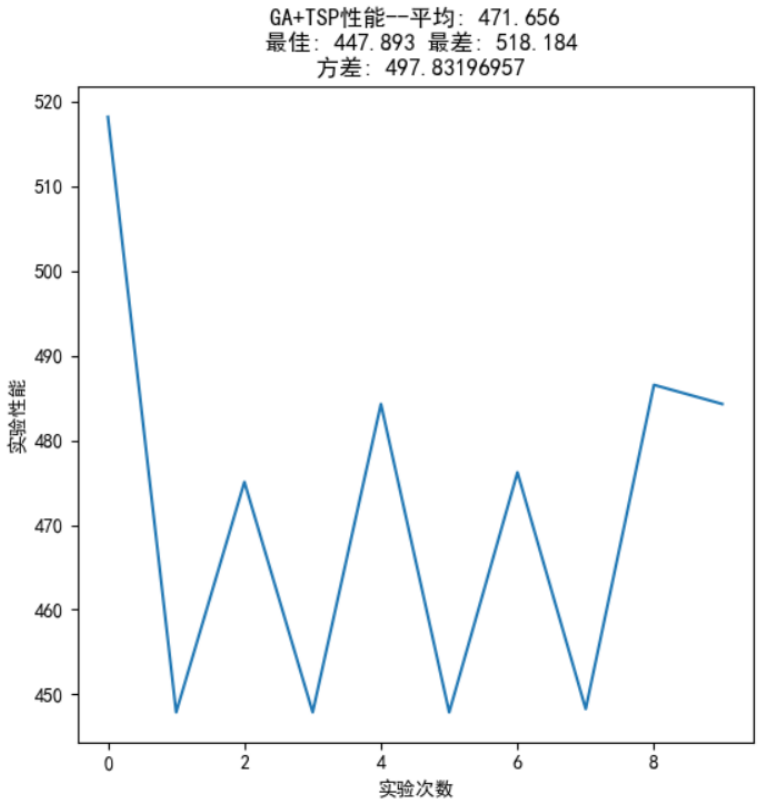


图（2）遗传算法解决 TSP 问题

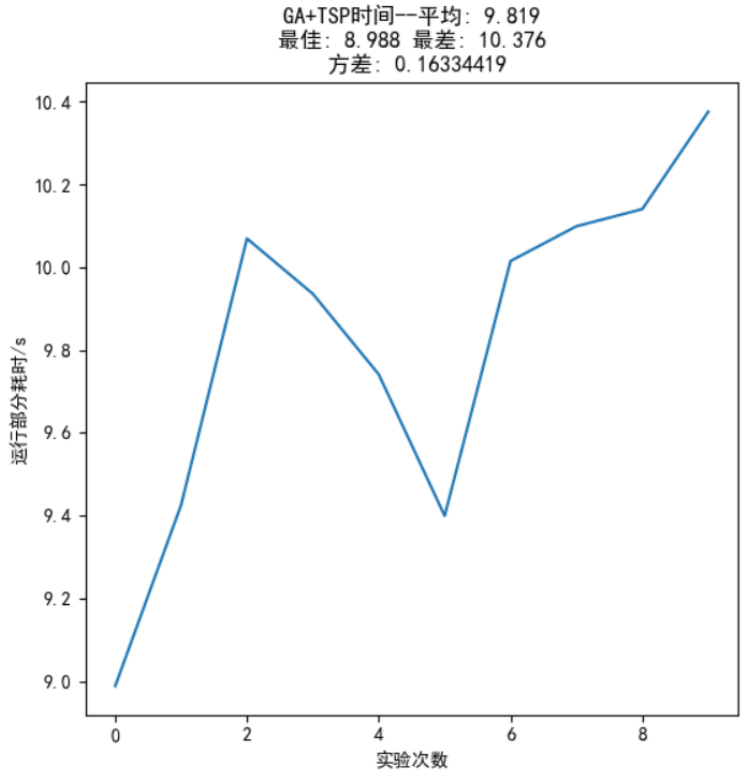


图（3）结果地图

（2） 多次运行结果（随机生成地图）

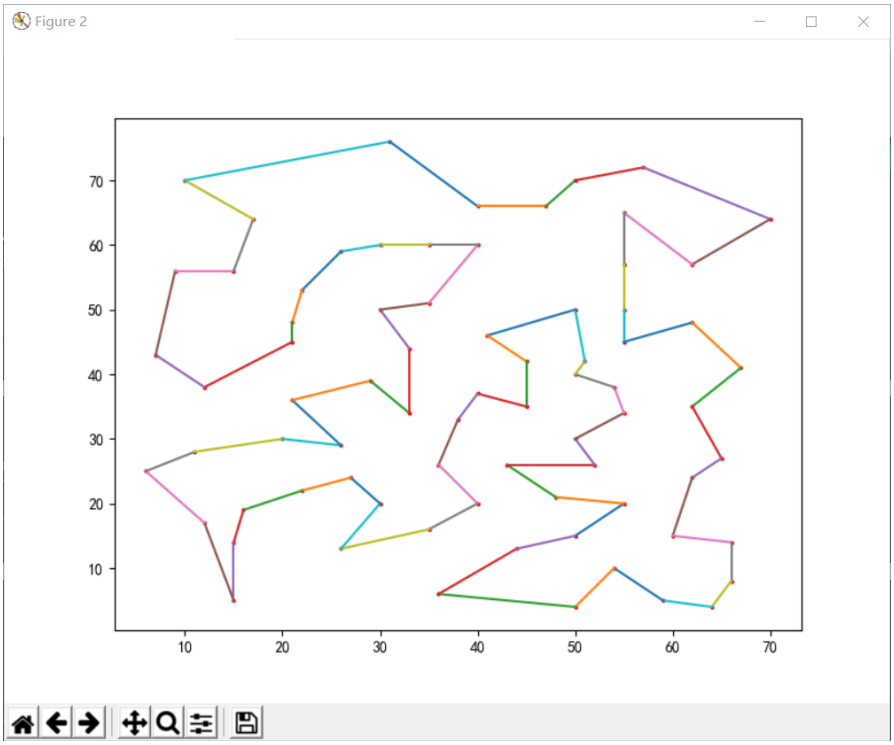


图（4）10 次遗传算法实验得到的解

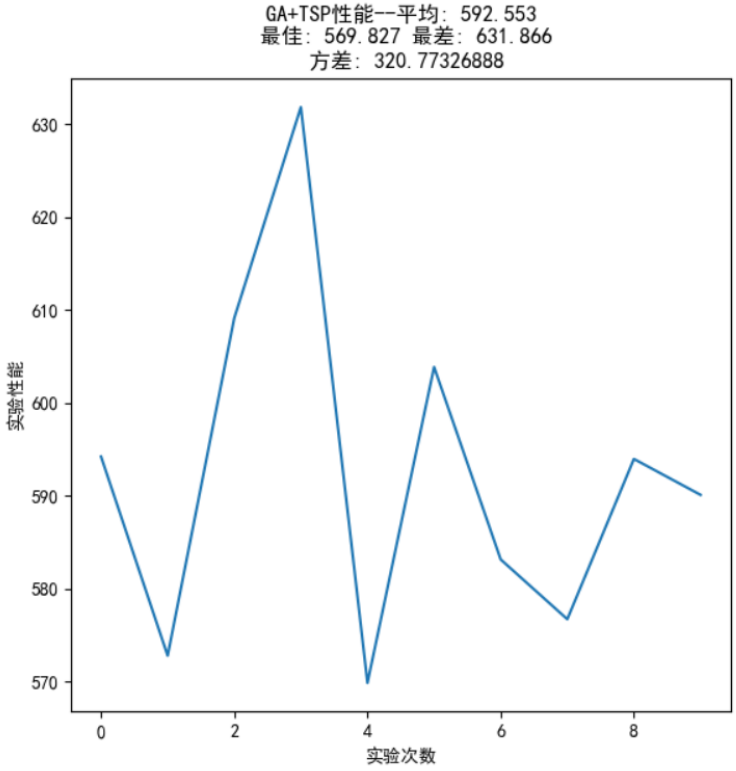


图（5）10 次遗传算法实验分别花费的时间

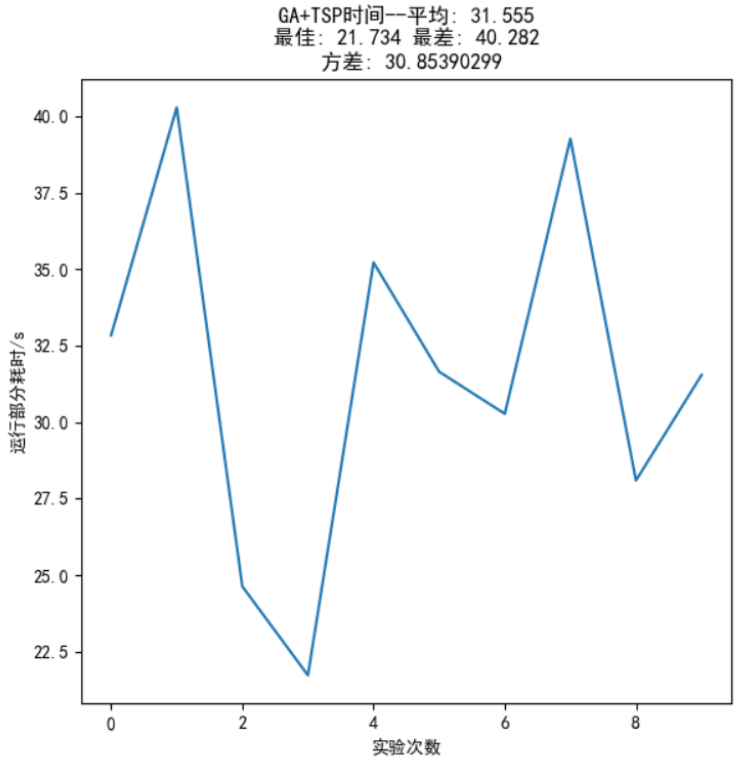
（3） 多次运行结果（固定地图）



图（6）得到的结果图



图（7）10 次遗传算法实验得到的解



图（8）10 次遗传算法实验分别花费的时间

1.3.2 结果分析

表（2）遗传算法对于随机地图和固定地图两种条件下的的解

	最优解	最差解	平均值
随机地图	447.893	518.184	471.656
固定地图	569.827	631.866	592.553

表（3）遗传算法对于随机地图和固定地图两种条件下的的求解耗时

	最快	最慢	平均值
随机地图	8.988	10.376	9.819
固定地图	21.734	40.282	31.555

通过遗传算法解决 TSP 问题得到解的精度能把误差控制在 15%以内，如果是解决随机地图的问题算法的速度大概在 10s 左右，对于一个具有五十个结点的固定地图时间在 30s 左右。

遗传算法具有良好的全局搜索能力，可以快速地将解空间中的全体解搜索出，而不会陷入局部最优解的快速下降陷阱，但是遗传算法的局部搜索能力较差，导致单纯的遗传算法比较费时，在进化后期搜索效率较低，在上面对连续问题的求解中可以看到求解速度稍慢。当我们使用的变异率为 0.95 时，可以获得上表中的结果，如果将变异率调低，那么求解速度会更慢。

利用遗传算法的内在并行性，可以方便地进行分布式计算，加快求解速度。在实际应用中，遗传算法容易产生早熟收敛的问题。采用何种选择方法既要使优良个体得以保留，又要维持群体的多样性，一直是遗传算法中较难解决的问题。

遗传算法的优点可以总结为以下几点：与问题领域无关切快速随机的搜索能力。搜索从群体出发，具有潜在的并行性，可以进行多个个体的同时比较。搜索使用评价函数启发，过程简单。使用概率机制进行迭代，具有随机性。具有可扩展性，容易与其他算法结合。

第二章 总结与参考

2.1 总结

智能优化技术是计算机研究领域的一个重要分支，它已经渗透到生活中的各个领域，计算机技术的高速发展为各行业的生命注入了新的血液，给我们的生活带来了极大的便利，这同时对各行业的发展也是一个考验，人们将更加离不开智能优化技术，而计算机也将更好地服务于人类，使人们的生活更加丰富。未来智能优化技术将更加适应人们的生活。

当前，智能计算正在蓬勃发展，研究智能计算的领域十分活跃。虽然智能算法研究水平暂时还很难使“智能机器”真正具备人类的智能，但智能计算将在 21 世纪蓬勃发展，人工智能将不仅是模仿生物脑的功能，而且两者具有相同的特性，这两者的结合将使人工智能的研究向着更广和更深的方向发展，将开辟一个全新的领域，开辟很多新的研究方向。智能计算将探索智能的新概念、新理论、新方法和新技术,而这些研究将在以后的发展中取得重大的成就。

2.2 参考与引用

[1] 高经纬, 张煦, 李峰, 等. 求解 TSP 问题的遗传算法实现[J]. 计算机时代, 2004, 2: 19-21.

[2] 吴春梅.现代智能优化算法的研究综述[J].科技信息,2012(08):31+33.

[3] 沈小伟,万桂所,王一云.现代智能优化算法研究综述[J].山西建筑,2009,35(35):30-31.

[4] 李岩,袁弘宇,于佳乔,张更伟,刘克平.遗传算法在优化问题中的应用综述[J].山东工业技术,2019(12):242-243+180.

[5] 刘锦. 混合遗传算法和模拟退火算法在 TSP 中的应用研究[D].华南理工大学,2014.

附录一 核心代码

GA.algorithm.py

```
import numpy as np
import copy

class GA_optimizer():
    def __init__(self, class_individual, N, C, M, nochange_iter, choose_mode='range', last_generation_left=0.2, history_convert = lambda x:x):
        # class_individual: 个体的 class，可调用产生新个体
        # N: 种群规模
        # C: 交叉概率
        # M: 变异概率
        # nochange_iter: 性能最好的个体保持不变 nochange_iter 回合后，优化结束
        # history_convert: 在记录 history 时将 fitness 转化为实际效用
        # last_generation_left: 保留上一代的比例
        # choose_mode: range（按排名算概率）/ fitness（按 fitness 算概率）

        # assert N%2==0 # 默认种群规模为偶数
        self.class_individual = class_individual
        self.N = N
        self.C = C
        self.M = M
        self.nochange_iter = nochange_iter
        self.history_convert = history_convert
        self.last_generation_left=last_generation_left
        self.choose_mode = choose_mode

    # 由旧种群产生新种群，包含交叉变异操作
```

```
    def selection(self, population, fitnesses):
        # 按排名分配被选择的概率
        population_fit_sorted = sorted(zip(fitnesses, population), key=lambda x:x[0])
# 按 fitnesses 从小到大排序
        population_sorted = list(zip(*population_fit_sorted))[1]
        population_sorted_left = population_sorted[::-
1][:int(self.last_generation_left*len(population_sorted))]
        population_sorted_left = population_sorted_left[::-1]
        if self.choose_mode=='range':
            choose_probability = list(range(1, len(population_sorted_left)+1))
            choose_probability =
np.array(choose_probability)/np.sum(choose_probability)
        elif self.choose_mode=='fitness':
            fitness_sorted = list(zip(*population_fit_sorted))[0]
            choose_probability = (fitness_sorted[::-
1][:len(population_sorted_left)])[::-1]
            choose_probability =
np.array(choose_probability)/np.sum(choose_probability)

        new_population = [population_sorted_left[-1], population_sorted_left[-2]] #
先将当前种群效用最佳的两个个体继承到子代种群
        # new_population = []
        while len(new_population)<self.N:
            # 按适配值大小随机选择两个个体
            p1 = np.random.choice(population_sorted_left, p=choose_probability)
            p2 = np.random.choice(population_sorted_left, p=choose_probability)

            # 按概率随机选择是否进行交叉
            if np.random.rand()<self.C:
                p1_chromosome_new, p2_chromosome_new = p1.crossover(p2)
```

```
                p1_new = self.class_individual(p1_chromosome_new)
                p2_new = self.class_individual(p2_chromosome_new)
            else:
                p1_new = copy.deepcopy(p1)
                p2_new = copy.deepcopy(p2)

            # 进行随机变异
            p1_new.mutation(self.M)
            p2_new.mutation(self.M)

            if p1_new.fitness()>p2_new.fitness():
                new_population.append(p1_new)
            else:
                new_population.append(p2_new)

        return new_population


def optimize(self, max_iteration, verbose=True):
        # max_iteration:  最大迭代次数
        # verbose:  是否有 print

        population = []
        for i in range(self.N):
            a = self.class_individual()
            a.randomize()
            population.append(a)

        best_individual = population[0] #  保优操作，保存性能最好的个体
        nochange_iter_running = self.nochange_iter
```

```
        fitness_history = []

        for i in range(max_iteration):
            fitness_history.append(self.history_convert(best_individual.fitness()))
            if nochange_iter_running < 0:
                break

            fitnesses = [a.fitness() for a in population]

            # 找到最优的个体，保优
            best_index = np.argmax(fitnesses)
            if fitnesses[best_index] > best_individual.fitness():
                nochange_iter_running = self.nochange_iter
                best_individual = copy.deepcopy(population[best_index])

            population = self.selection(population, fitnesses)

            nochange_iter_running = nochange_iter_running - 1
            if verbose:

class GA_Individual():
    def __init__(self, chromosome=None):
        self.num_of_points = TSP_map.num_of_points
        if type(chromosome) == list:
            self.chromosome = chromosome    # x0 和 x1 的二进制代码
        else:
            self.chromosome = list(range(self.num_of_points))

    def randomize(self):
        np.random.shuffle(self.chromosome)
```

```
    # 单位置次序交叉
    def crossover_order(self, p2):
        # p2: 用于交叉的另一个个体
        p1_chromosome = self.chromosome
        p2_chromosome = p2.chromosome
        # 随机选择一个节点进行交叉
        joint = np.random.choice(range(1, self.num_of_points - 1))

        # 从 chromosome_added 中去掉 chromosome_part 的节点，然后将结果与 chromosome_part 拼接起来
        def complete_chromosome(chromosome_part, chromosome_added):
            # chromosome_part: 待补充的染色体
            # chromosome_added: 用于补充的染色体
            left_points = list(set(chromosome_added) - set(chromosome_part))
            left_points_sorted = sorted(left_points, key=chromosome_added.index)
            return chromosome_part + left_points_sorted

        p1_chromosome_new_part = p1_chromosome[:joint]
        p1_chromosome_new_complete = complete_chromosome(p1_chromosome_new_part, p2_chromosome)
        p2_chromosome_new_part = p2_chromosome[:joint]
        p2_chromosome_new_complete = complete_chromosome(p2_chromosome_new_part, p1_chromosome)
        return p1_chromosome_new_complete, p2_chromosome_new_complete

    # 部分映射交叉
    def crossover_partially(self, p2):
        # p2: 用于交叉的另一个个体
        p1_chromosome = self.chromosome
```

```
            p2_chromosome = p2.chromosome
            # 随机选择两个节点进行交叉
            joint_left = np.random.choice(range(self.num_of_points - 1))
            joint_right = np.random.choice(range(joint_left + 1, self.num_of_points))

            def conflict_fill_in(chromosome_base, chromosome_added):
                # chromosome_base: 待填入的染色体
                # chromosome_added: 用于填入的染色体
                result = []
                for i in range(len(chromosome_base)):
                    if i >= joint_left and i < joint_right:
                        result.append(chromosome_added[i - joint_left])
                    else:
                        point_base = chromosome_base[i]
                        while True:
                            if point_base in chromosome_added:
                                index_in_added = chromosome_added.index(point_base)
                                point_base = chromosome_base[index_in_added + joint_left]
                            else:
                                break
                        result.append(point_base)
                return result

            p1_chromosome_new = conflict_fill_in(p1_chromosome, p2_chromosome[joint_left:joint_right])
            p2_chromosome_new = conflict_fill_in(p2_chromosome, p1_chromosome[joint_left:joint_right])
            return p1_chromosome_new, p2_chromosome_new
```

```
def crossover(self, p2):
    return self.crossover_partially(p2)
    # return self.crossover_order(p2)

def mutation(self, p):
    if np.random.rand() > p:
        return
    index_list = list(range(len(self.chromosome)))
    sam = list(random.sample(index_list, 2))
    start, end = min(sam), max(sam)
    tmp = self.chromosome[start:end]
    # np.random.shuffle(tmp)
    tmp = tmp[::-1]
    self.chromosome[start:end] = tmp

def fitness(self):
    distance_sum = TSP_map.route_distance(self.chromosome)
    return -distance_sum

def __repr__(self):
    return str(self.chromosome)
```