智能优化技术结课报告

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

基于智能优化算法解决连续问题和离散问题

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，求函数最小值，模拟退火算法，遗传算法

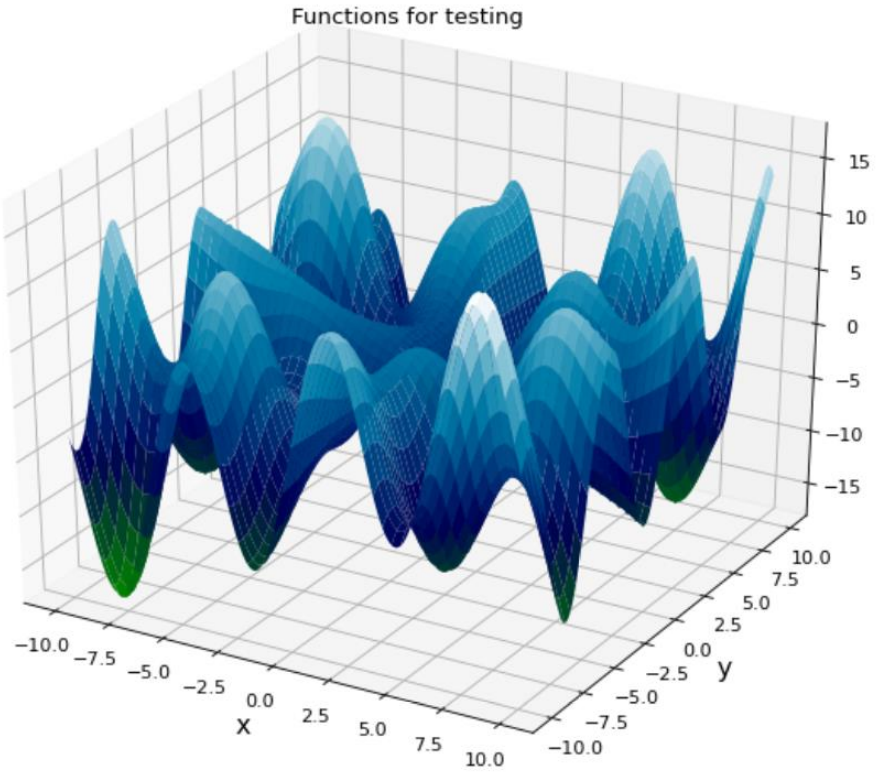
第一章 智能优化算法解决连续问题

1.1 问题描述

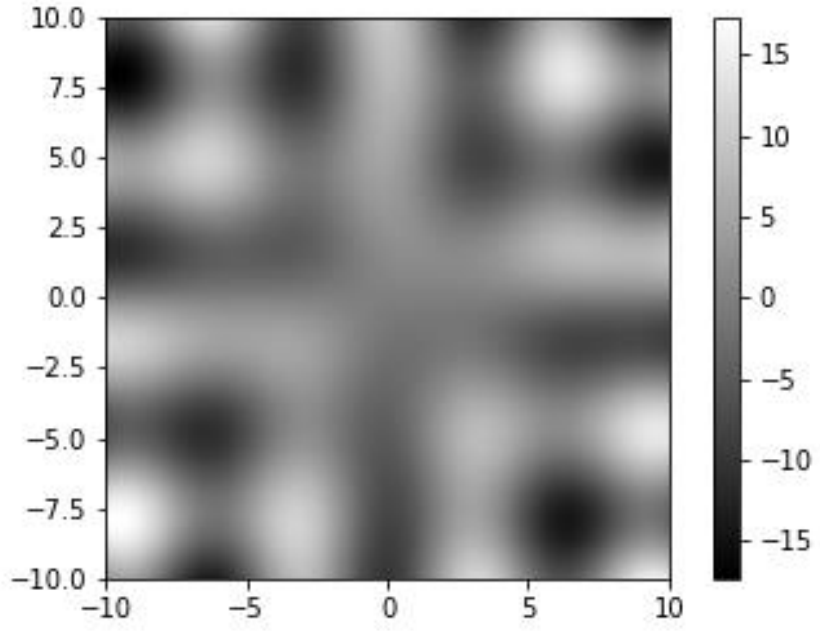
通过智能优化算法求解连续问题的典型案例就是求解多极值函数最小（大）值。问题中，我们需要求出函数的最值，以下列二元函数作为案例进行分析：

$$f(x,y) = -y \times \sin x - x \times \cos y \quad x,y \in (-10,10)$$

通过对该函数的图像分析，该函数存在多个极小值和极大值，我们可以使用模拟退火算法或者遗传算法对其最小值进行求解，函数图像见图（1）,通过对该函数的函数值分析可以得到热力图，并初步判断最小值位置。



图（1）案例函数在指定区间内的图像



图（2）案例函数根据具体函数值得到的热力图

通过对区间(−10, 10)内的函数进行分析可以得到一个热力图，其中颜色越深的地方代表函数值越小，反之颜色越浅表示函数值越大，通过图像初步判断该函数最小值坐标 x 位于(5,10)区间，坐标 y 位于(-7.5,-10)区间。

1.2 算法描述

1.2.1 模拟退火算法

模拟退火算法是一种通用的优化算法，是局部搜索算法的扩展。它与局部搜索算法的不同之处，是以一定的概率选择邻域中目标值大的状态。从理论上来说，它是一种全局最优算法。模拟退火算法具有十分强大的全局搜索性能，这是因为它采用了许多独特的方法和技术：基本不用搜索空间的知识或者其他的辅助信息，而只是定义邻域结构，在邻域结构内选取相邻解，再利用目标函数进行评估；采用概率的变迁来指导它的搜索方向，它所采用的概率仅仅是作为一种工具来引导其搜索过程朝着更优化解的区域移动。因此，虽然看起来它是一种盲目的搜索方法，但实际上有着明确的搜索方向。

（1） 流程

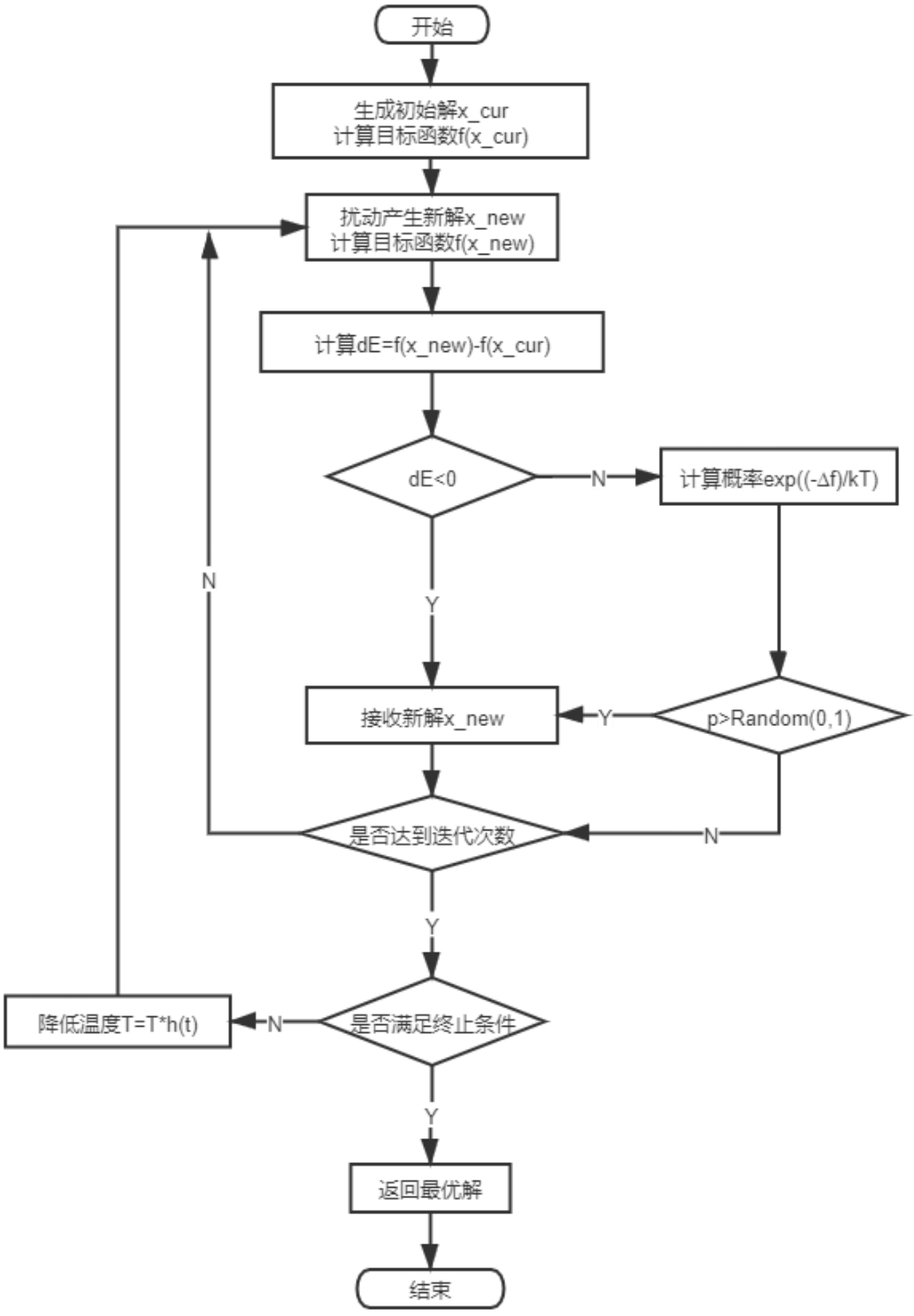
步骤 1：产生初始温度和初始解。初始温度可以有两种生成方式：按照经验设定固定初温或随机选择指定数量的状态计算目标值并把方差作为初温；初始解在连续函数中，为函数值。

步骤 2：判断是否满足收敛准则(外循环)，若满足则输出结果，结束，不满足则转步骤 3。在外循环中，温度随着迭代而发生变化，收敛准则分为基于时间的收敛（指定迭代次数，需要指定迭代次数）和基于性能的收敛（连续若干步的目标值之差小于阈值，需要指定阈值和迭代步数）。

步骤 3：判断是否满足抽样稳定准则(内循环)，若满足则退温(外循环)，并跳转到步骤 2；不满足则转到步骤 4。抽样稳定准则是在内循环中用来决定同一个温度下产生解的个数，温度不随着内循环迭代而发生改变。收敛准则分为基于时间的收敛（指定迭代次数，需要指定迭代次数）和基于性能的收敛（连续若干步的目标值之差小于阈值，需要指定阈值和迭代步数）。

步骤 4：由当前状态产生新状态，得到新解。

步骤 5：判断新状态和新解是否可被接受，若能被接受则用新状态替换当前状态；若不被接受则保持当前状态不变，回到步骤 3。



图（3）模拟退火算法流程图

（2） 说明

抽样稳定准则是在内循环中用来决定同一个温度下产生解的个数，温度不随着内循环迭代而发生改变。收敛准则分为基于时间的收敛（指定迭代次数，需要指定迭代次数）和基于性能的收敛（连续若干步的目标值之差小于阈值，需要指定阈值和迭代步数）。

1.2.2 遗传算法

遗传算法是通过模仿自然界生物进化机制而发展起来的随机全局搜索和优化方法。它借鉴了达尔文的进化论和孟德尔的遗传学说，本质上是一种并行、高效、全局搜索的方法，它能在搜索过程中自动获取和积累有关搜索空间的知识，并自适应地控制搜索过程以求得最优解。遗传算法操作：使用“适者生存”的原则，在潜在的解决方案种群中逐次产生一个近似最优的方案。在每一代中，根据个体在问题域中的适应度值和从自然遗传学中借鉴来的再造方法进行个体选择，产生一个新的近似解。这个过程导致种群中个体的进化，得到的新个体比原个体更能适应环境。

将遗传算法与求解多极值函数最小值问题相结合需要注意以下几点：

（1）编码

遗传算法需要对问题的解进行编码，对此，我们使用 32 位二进制码表示问题的解，前 16 位表示 Lo,后 16 位表示 r1。x0，L1 的取值范围都为[-10,10],使用 16 位二进制码表示可以使解精确到小数点后 3 位。二进制码$[S]_2$与实数 x 的关系为:

$$x = -10 + \frac{[s]_2}{2^{16} - 1} \times (10 + 10)$$

（2）流程

步骤 1：令 k=0，随机产生 N 个初始个体构成初始种群 P(0)。

步骤 2：评价 P(k)中各个体的适配值(fitness value)。

步骤 3:判断算法收敛准则是否满足。若满足则输出搜索结果;否则执行以下步骤。

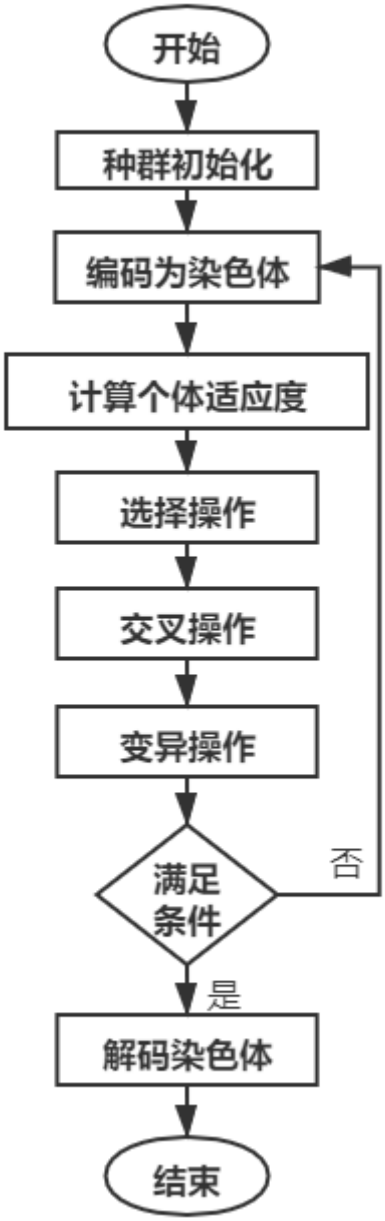
步骤 4：令 m= 0。

步骤 5：根据适配值大小以一定方式执行复制操作来从 P(k)中选取两个个体。

步骤 6：若交叉概率 $P_c > \xi \in [0, 1]$，则对选中个体执行交叉操作来产生两个临时个体;否则将所选中父代个体作为临时个体。

步骤 7：按变异概率 Pm 对临时个体执行变异操作产生两个新个体并放入 P(k+1)，并令 m=m+2。

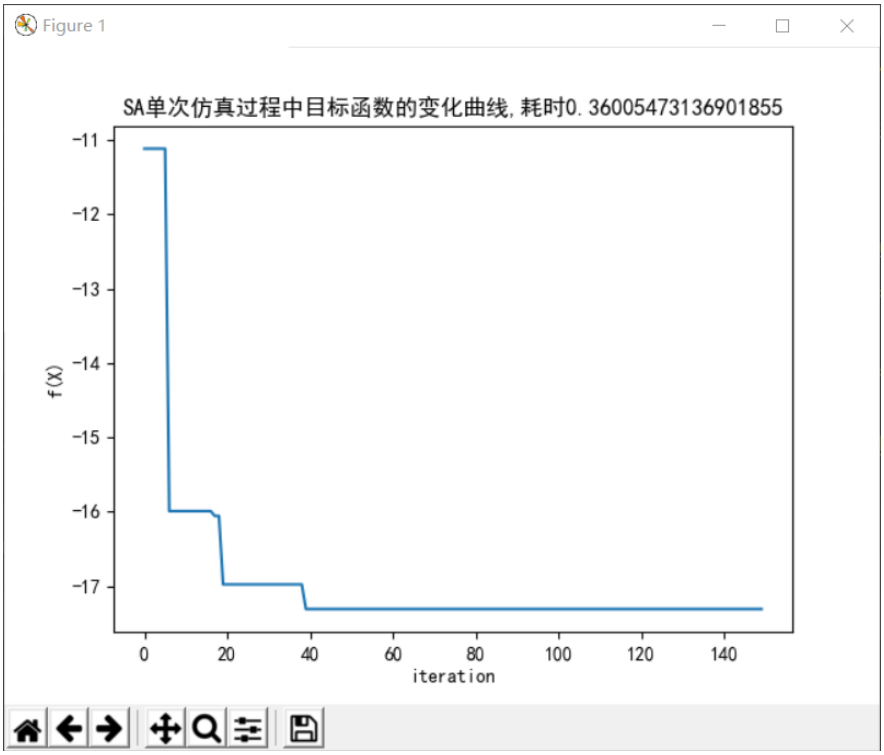
步骤 8：若 m<N，则返回步骤 5；否则令 k=k+l 并返回步骤 2。



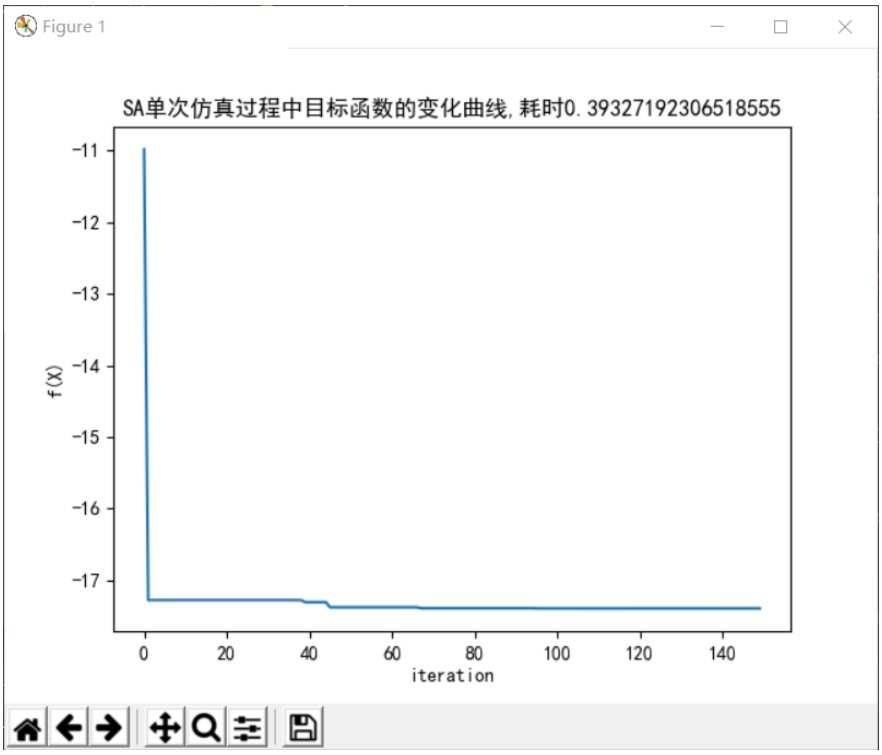
图（4）遗传算法流程图

1.3 实验

1.3.1 模拟退火算法实验结果

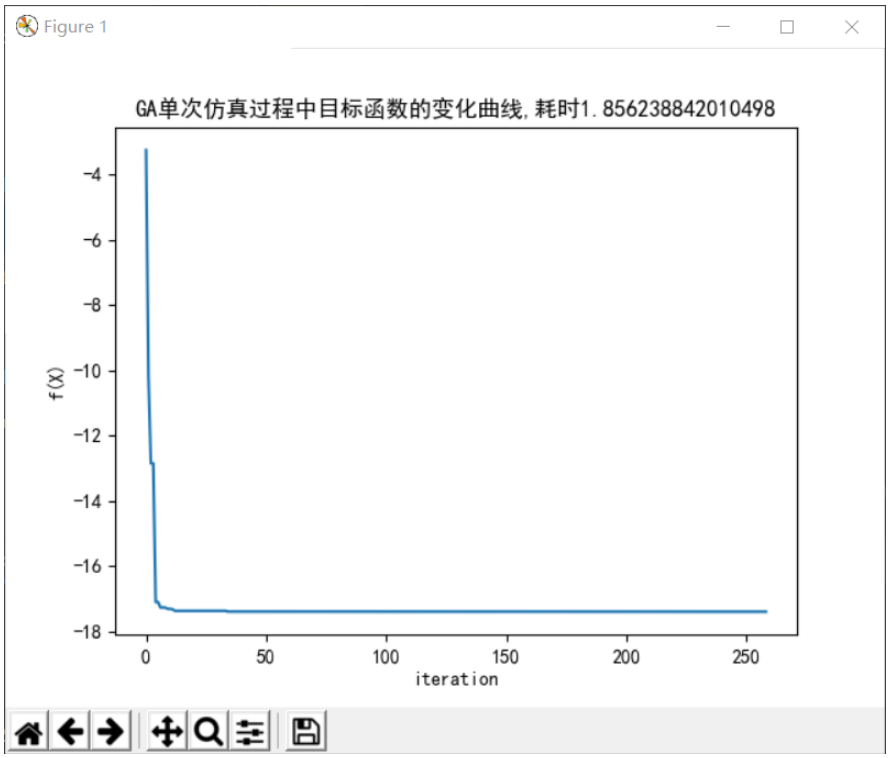


图（5）模拟退火算法实验结果 1

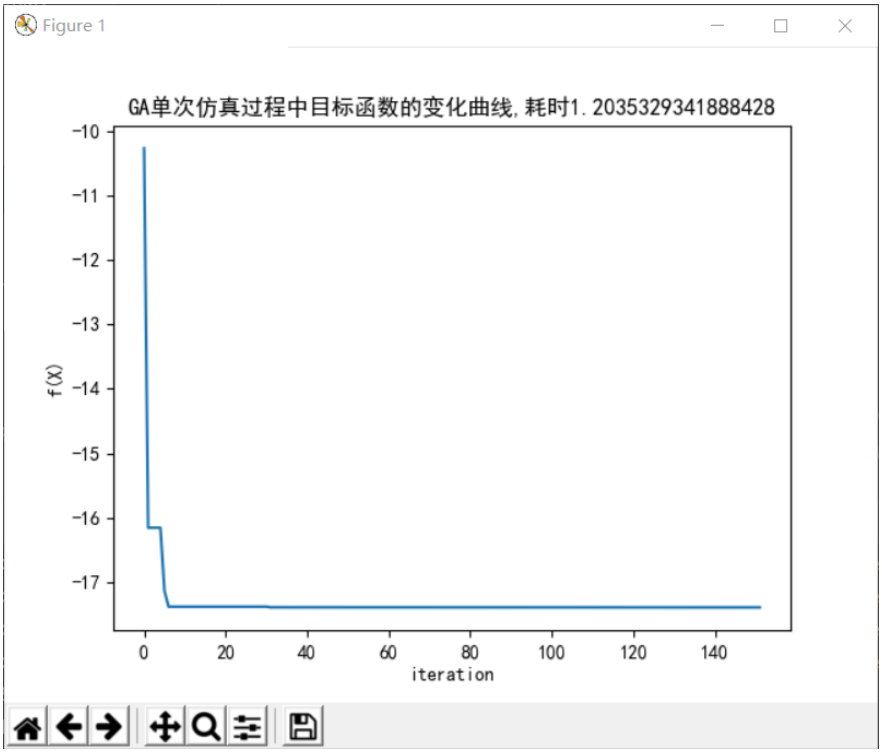


图（6）模拟退火算法实验结果 2

1.3.2 遗传算法实验结果



图（7）遗传算法实验结果 1



图（8）遗传算法实验结果 2

1.3.3 对比与分析

（1） 模拟退火算法实验参数

表（1）模拟退火算法实验参数

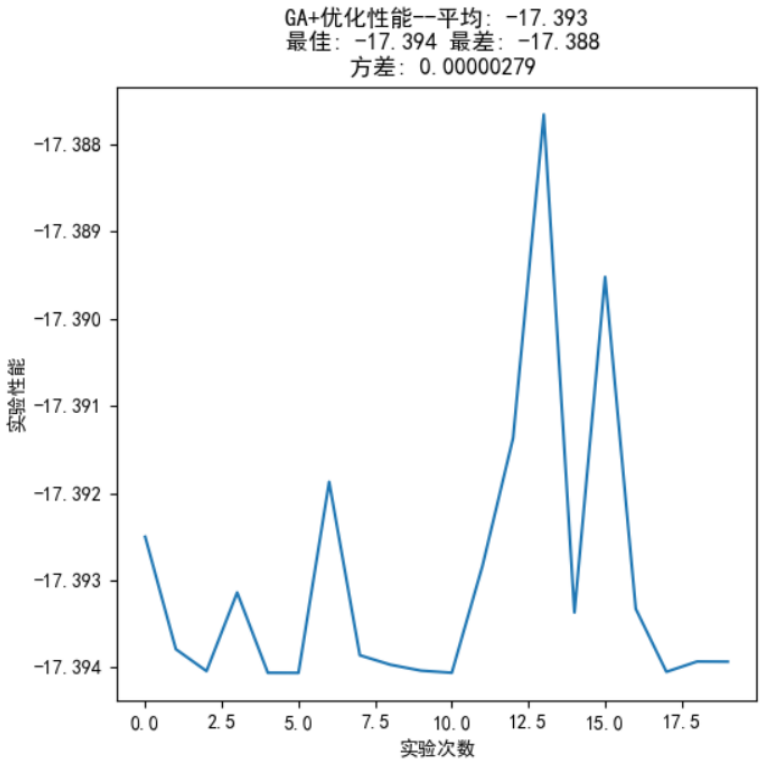
参数名称	参数含义	参数值
T0_mode	初温生成模式	experience
T_annealing_mode	退火模式(ordinary 常用指数退温,log 即温度与退温不输的对数成反比)	ordinary
T_Lambda	当退火模式为指数退温时的温度衰减系数	0.9
scale_min	自变量范围下限	-10
scale_max	自变量范围上限	10
x_eta	自变量更新时的系数 eta	2
x_mode	更新自变量/状态时使用的模式（高斯分布 Gauss/柯西分布 Cauchy）	Gauss

（2） 遗传算法实验参数

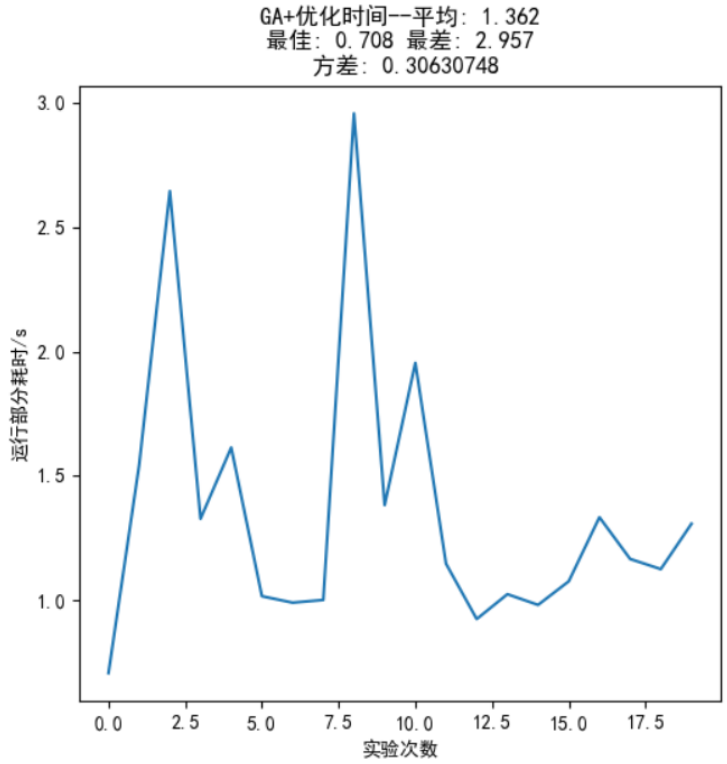
表（2）遗传算法实验参数

参数名称	参数含义	参数值
N	种群规模	30
C	交叉概率	0.95
M	变异概率	0.2
nochange_iter	性能最好的个体保持不变 nochange_iter 回合后，优化结束	100
last_generation_left	保留上一代的比例	0.2
choose_mode	计算概率的模式	按排名算概率
assert	种群规模	偶数

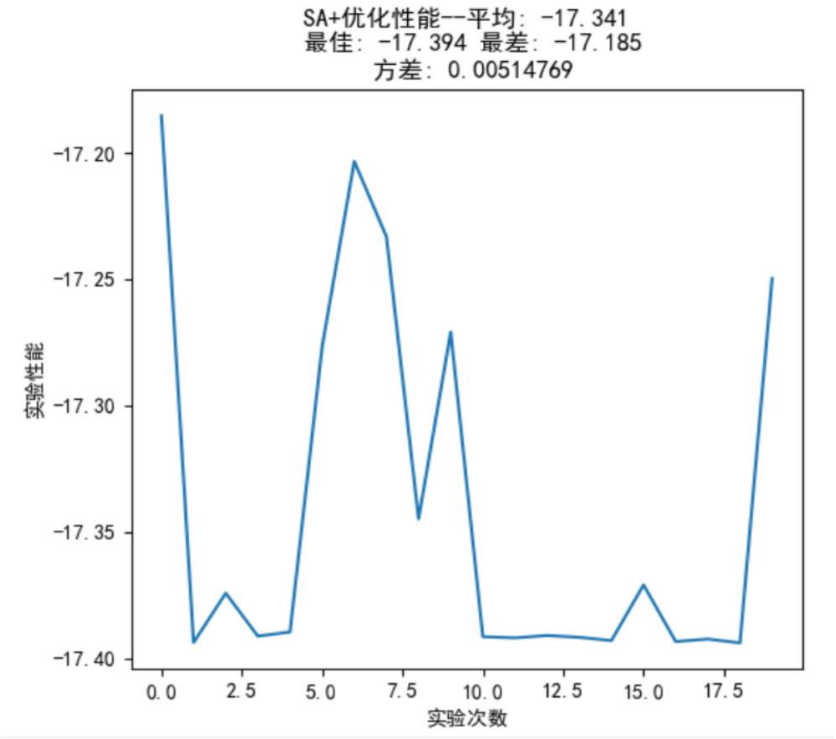
（3） 结果



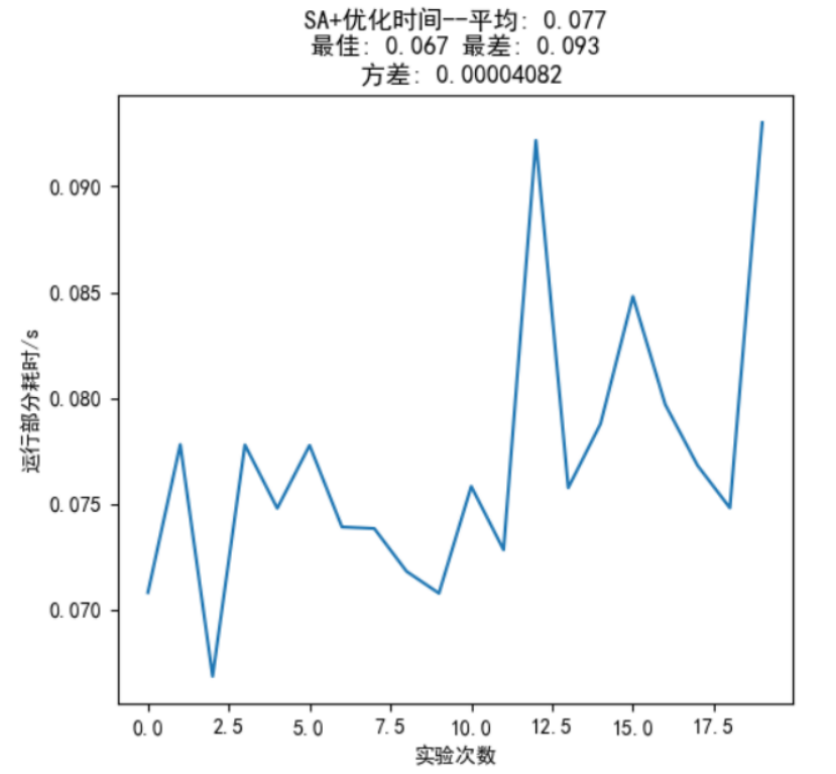
图（9）20 次遗传算法实验得到的解



图（10）20 次遗传算法实验分别花费的时间



图（11）20 次模拟退火算法实验得到的解



图（12）20 次模拟退火算法实验分别花费的时间

（4） 结果分析

通过对遗传算法（GA）和模拟退火算法（SA）在连续优化问题上连续二十轮的实验进行分析，可以看出 GA 算法和 SA 算法都可以解出该函数的最小值，但是 GA 算法可以保证结果持续稳定在-17.38 以下，而 SA 算法虽然得到了最优解的次数更多，但从图（9）可以看出其结果并不稳定。

通过对于两种算法的运行时间分析，可以看到 GA 算法求得最优解的最快时间（0.708s）是 SA 算法求得最优解的最快时间（0.067s）的十倍以上，而 GA 算法最差时间（2.957s）是 SA 算法最差时间（0.093s）的 31.8 倍。

表（3）两种算法的最佳和最差求解时间

	最佳时间/s	最差时间/s
遗传算法	0.708	2.957
模拟退火算法	0.067	0.093

造成该现象的原因在于遗传算法的编程实现比较复杂，首先需要对问题进行编码，找到最优解之后还需要对问题进行解码。另外三个算子的实现也有许多参数，如交叉率和变异率，并且这些参数的选择严重影响解的品质，而目前这些参数的选择大部分是依靠经验。遗传算法本质上是一种随机搜索优化算法。当问题规模较大或问题较复杂时（即多参数寻优问题），往往造成搜索空间非常的庞大，从而导致遗传算法的收敛速度很慢，加之遗传算法本身存在着群体分散性和收敛性之间的矛盾，这给遗传算法的实时应用带来了极大的不便。

第二章 智能优化算法解决离散问题

2.1 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。

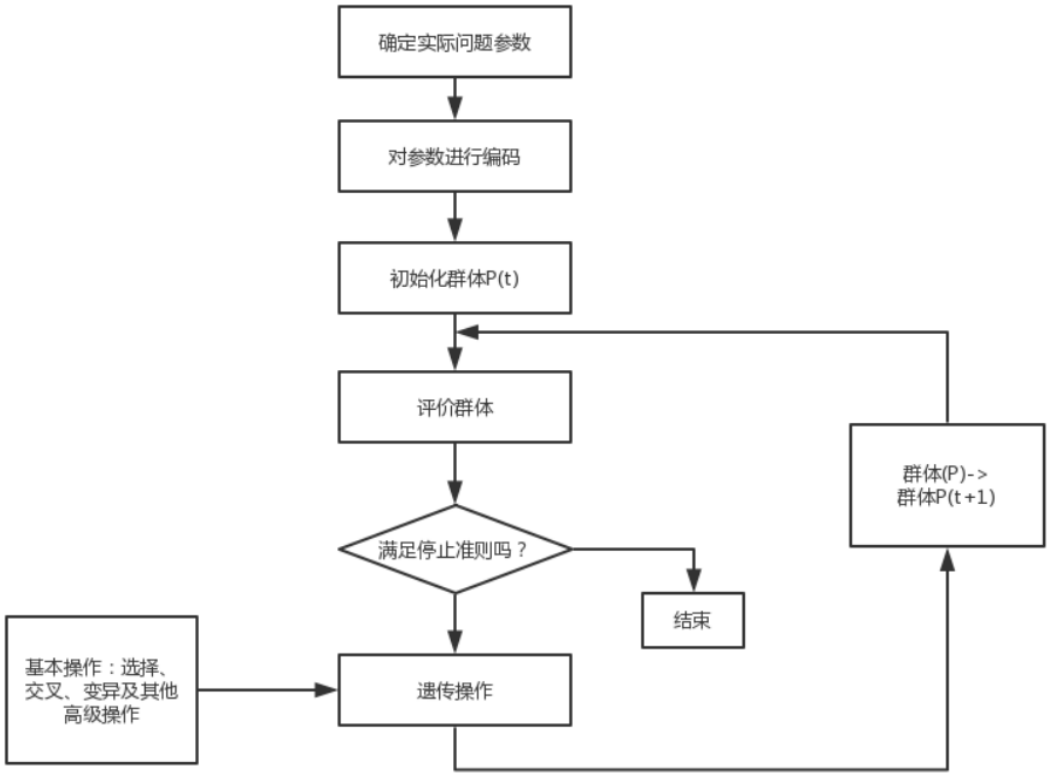
2.2 算法描述

2.2.1 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]。

2.2.2 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（13）遗传算法流程

2.2.3　算法参数

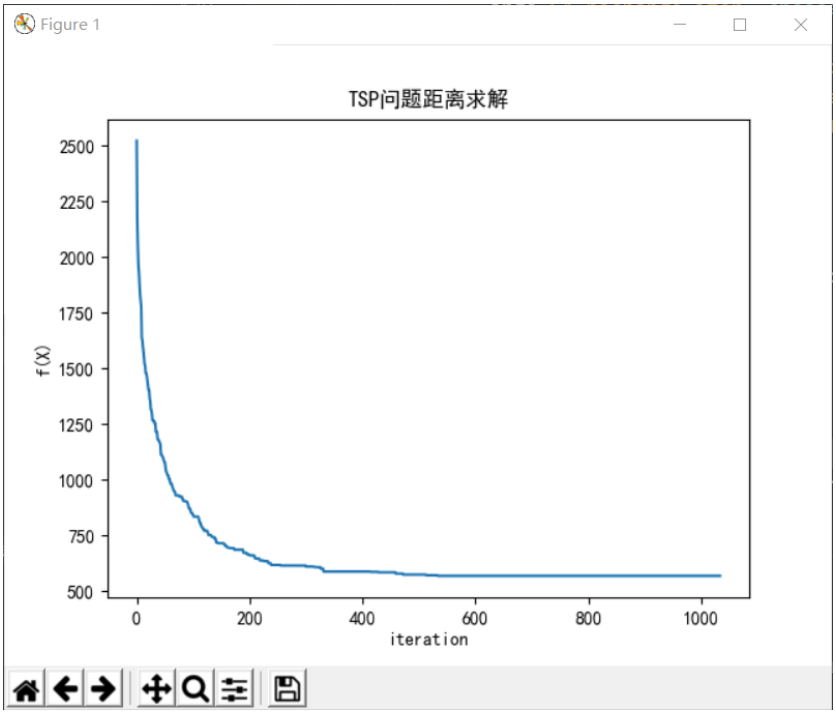
表（4）遗传算法实验参数

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

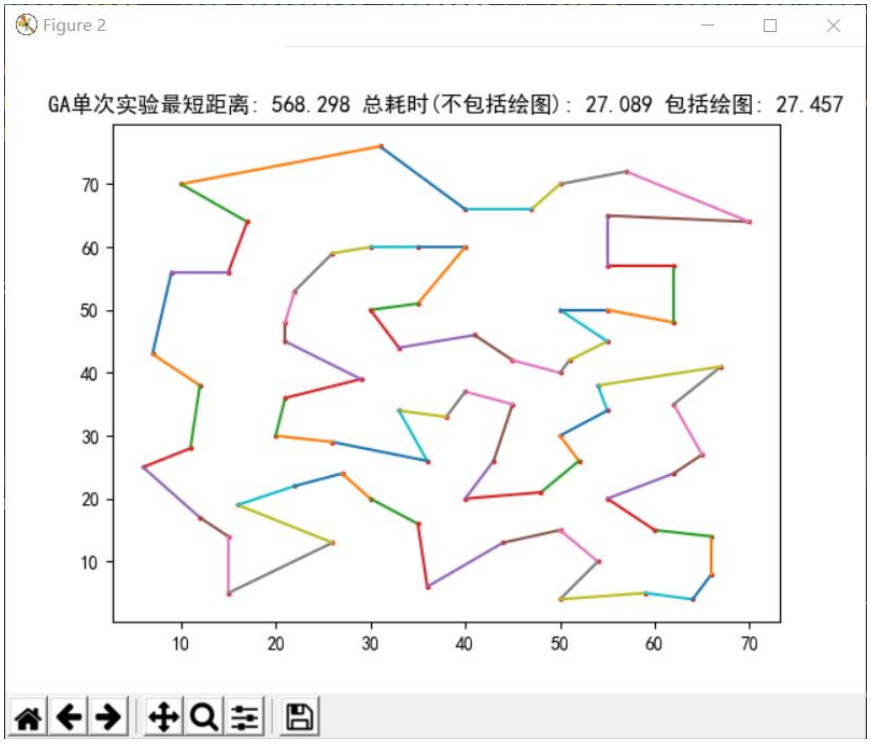
2.3 实验

2.3.1 实验结果

（1） 单次运行结果

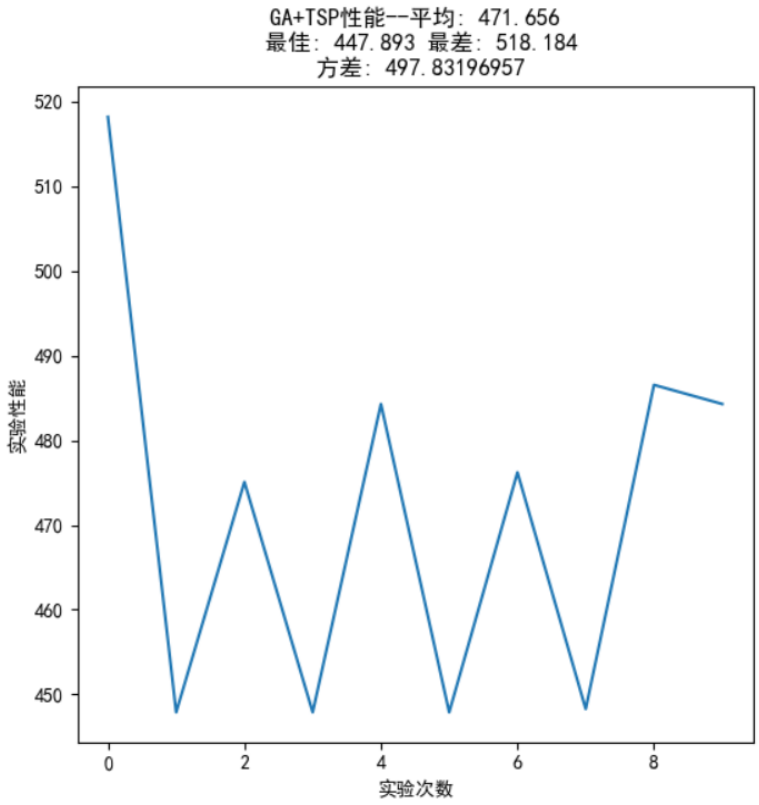


图（14）遗传算法解决 TSP 问题

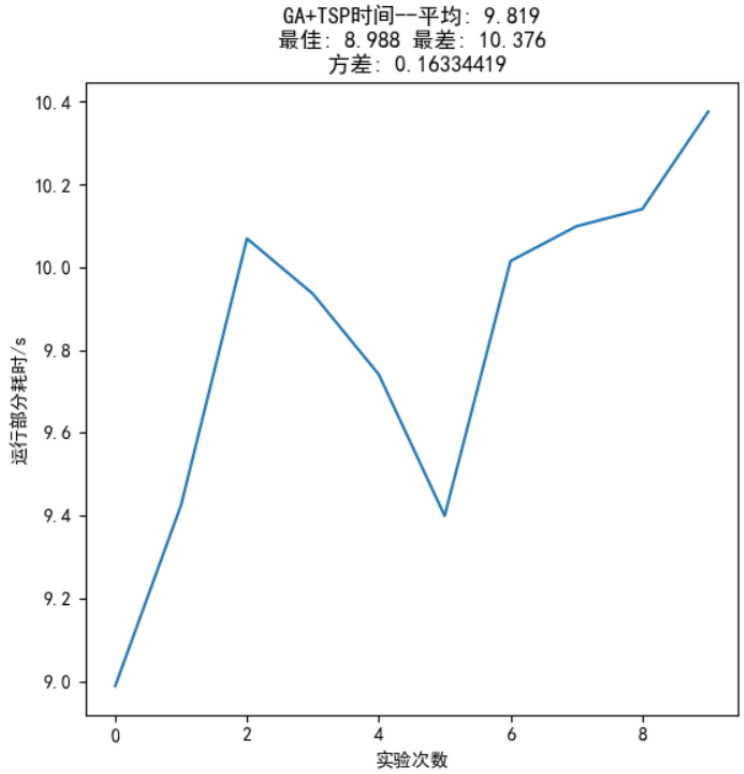


图（15）结果地图

（2） 多次运行结果（随机生成地图）

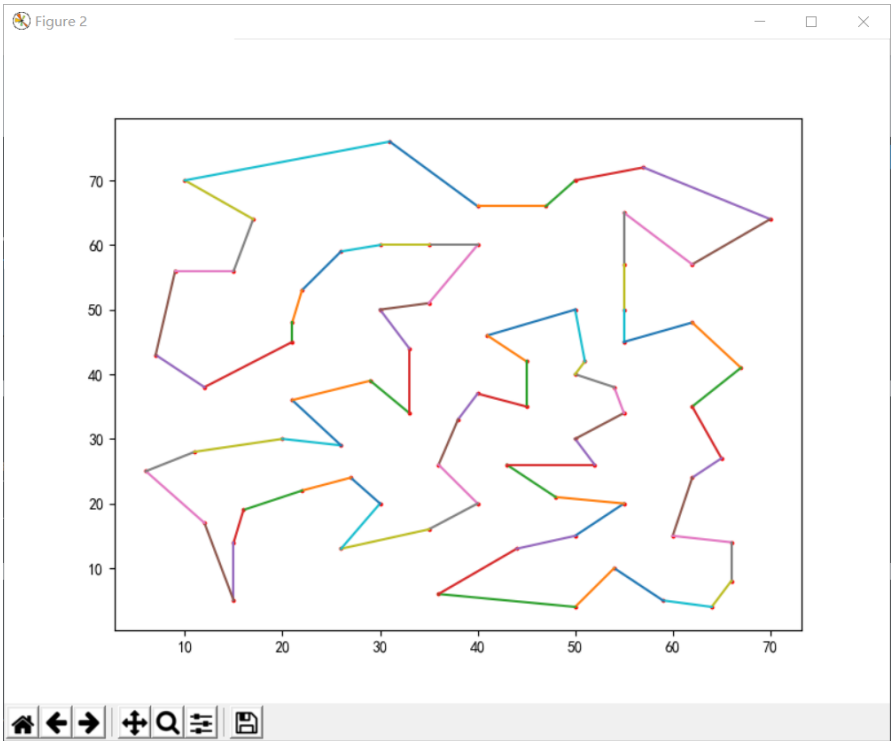


图（16）10 次遗传算法实验得到的解

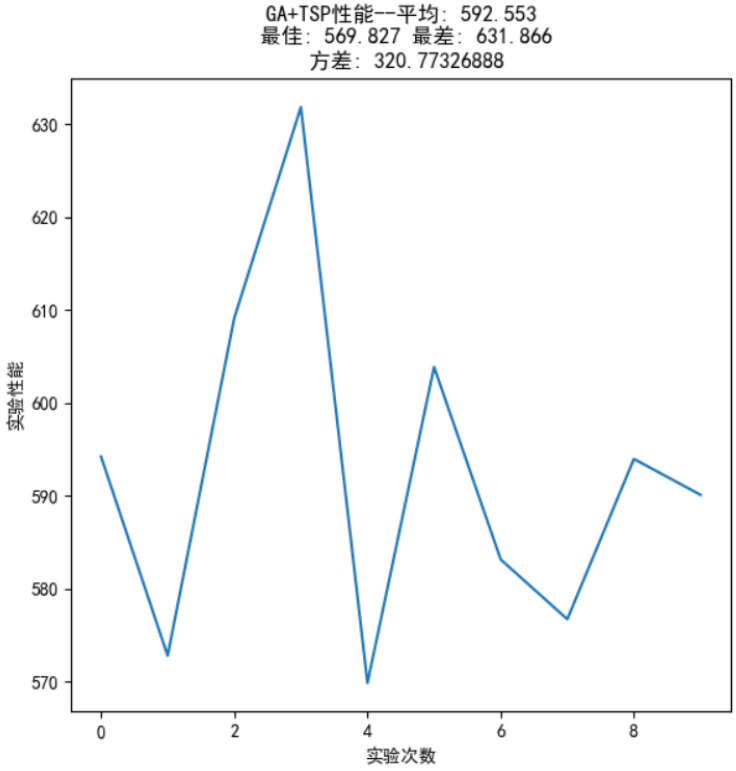


图（17）10 次遗传算法实验分别花费的时间

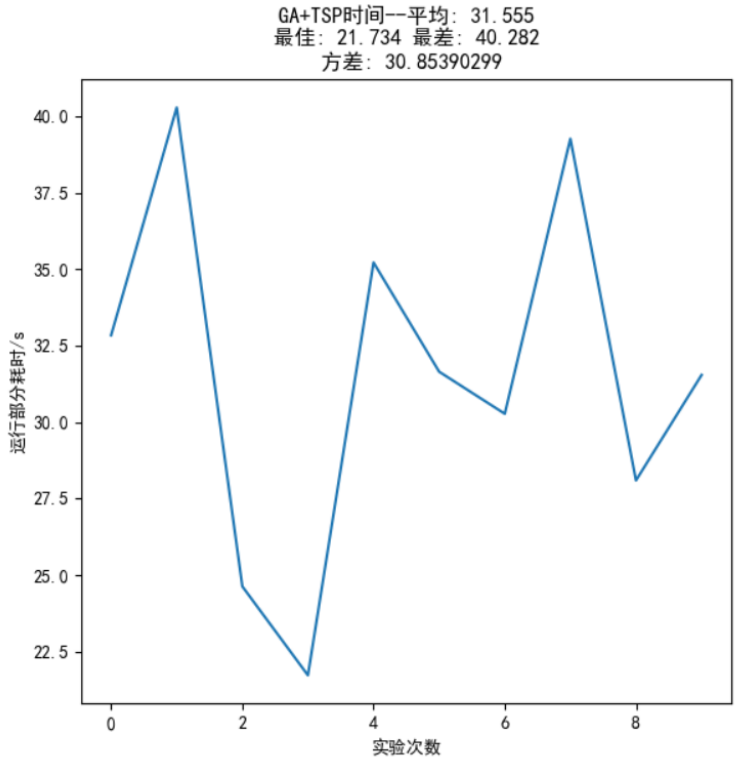
（3） 多次运行结果（固定地图）



图（18）得到的结果图



图（19）10 次遗传算法实验得到的解



图（20）10 次遗传算法实验分别花费的时间

2.3.2 结果分析

表（5）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（6）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。

第三章 总结与参考

3.1 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。

3.2 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:
                print('iteration: %d\t best_individual: %s\t best_fitness: %.3f\t nochange_iter:%d'%(i, best_individual.__repr__(), self.history_convert(best_individual.fitness()), nochange_iter_running))

        return best_individual, fitness_history
```