本科实验报告

课程名称：	人工智能
实验名称：	高级搜索
专业名称：	保密管理
学生姓名：	武自厚
学生学号：	20336014
实验地点：	东校园实验楼 D502
实验成绩：	
报告时间：	2022 年 5 月 5 日

一、 实验要求

1. 局部搜索

实现多种邻域操作的局部搜索算法, 并解决结点数量在 100 以内的旅行商问题.

2. 模拟退火算法

实现模拟退火算法, 并解决结点数量在 100 以上的旅行商问题, 与局部搜索算法比较效果. 提供可视化结果展示.

3. 遗传算法

设计比较高级的局部操作, 并解决结点数量在 100 以上的旅行商问题, 与模拟退火算法比较效果. 得出设计高效遗传算法的一些经验, 并比较单点搜索和多点搜索的优缺点.

二、 实验过程

1. 算法原理

(1) 直接登山法

登山法是一种完全的贪心算法, 它单纯地依靠启发式信息向当前的相邻状态前进, 并且由于只存储当前状态, 所以它并没有回溯功能, 容易陷入局部极值的干扰. 其中直接登山法是一种最为简单直接的实现, 它的主要步骤即是选择相邻状态的最值来前进, 直到当前状态不存在更大/小的相邻状态为止.

(2) 变邻域登山法

单一的邻域实现可能会让搜索陷入局部极大/小值, 但是如果在找到极值时更换一种邻域生成方法, 就有可能跳出局部极值. 在旅行商问题中可以采用三种邻域: 两个城市交换、两个城市之间路径倒转以及一个城市插入另一个位置. 只有当这三种邻域都达到极值时才返回结果.

(3) 模拟退火算法

这是一种模拟物理能量耗散的算法, 该算法中有一种称为 “温度” 的特殊变量, 它会决定算法接受相对较差解的概率. 在算法开始时接受差解的概率很高, 而随着算法的进行, 温度会下降, 算法更倾向于接受好的解. 这个算法初期鲁棒性高, 容易跳出局部最优解.

(4) 遗传算法

这是一种模拟自然生物界种群演化的算法. 首先将一系列解编码为线性表存储的 “染色体”, 其次再模拟自然选择、交叉互换以及基因突变的过程从而提高染色体对问题的适应度.

2. 伪代码展示

Algorithm 1: 直接登山法

输入: 初始状态 s, 评估函数 f
输出: 最优状态 s^*

```
loop
    U := s 的邻域
    s* := arg min f(s)
    if s* = s then
        return s*
    s := s*
```

Algorithm 2: 多邻域登山法

输入: 初始状态 s, 评估函数 f, 邻域生成函数 $\vec{F} = \{F_0, \cdots, F_{n-1}\}$
输出: 最优状态 s^*

```
counter := 0
F* := F_0
loop
    U := s 的 F* 邻域
    s* := arg min f(s)
    if s* = s then
        if counter = n then
            return s*
        else
            counter := 0
        counter = counter + 1
        F* := 下一个F
    s := s*
```

Algorithm 3: 模拟退火算法

输入: 初始状态 s, 评估函数 E, 邻域生成函数 $\vec{F} = \{F_0, \cdots, F_{n-1}\}$
输出: 最优状态 s^*

$T :=$ 随机抽取一定状态的评估函数值极差
$T_e := 0.1$
$\alpha := 98\%$
$l := 1000$
while $T > T_e$ **do**
    $s' :=$ 随机邻域F中随机生成的状态
    for $i := 1$ **to** l **do**
        if $E(s') < E(s)$ **then**
            $s := s'$
        else if 随机生成 $(0, 1]$ 中的数 $< \exp(\frac{E(s')-E(s)}{T})$ **then**
            $s := s'$
    $T := \alpha \cdot T$

Algorithm 4: 遗传算法

输入: 初始种群 S, 适应性函数 f, 基因突变率 $p_m = 0.01$, 交叉互换率 $p_c = 0.7$, 迭代数量 $l = 10000$
输出: 最优状态 s^*

for $i := 1$ **to** l **do**
    `select()`
    `crossover()`
    `mutate()`
    $s^* := \arg\max_{s \in S} f(s)$
return s^*

3. 关键代码

(1) 直接登山法和多邻域登山法

```
def climb(state, graph: NodeGraph):
    count = 0
    counts = []
    weights = []
    while True:
        # 选择邻域中最优的状态
        neighbor = min(next_states_swap(state), key=graph.weight)
        if graph.weight(neighbor) >= graph.weight(state):
```

```
            vsl.figure(counts, weights, '迭代次数', '总路径权值', '直接登山法优化效
            ↪ 果')
            return state
        else:
            # 如果没有最好的状态直接返回
            counts.append(count)
            weights.append(graph.weight(neighbor))
            state = neighbor
            count += 1

def climb_var(state, graph: NodeGraph):
    count = 0
    counts = []
    weights = []
    swap_which = 1
    min_count = 0

    next_states = {
        0: next_states_swap,
        1: next_states_reverse,
        2: next_states_insert,
    }

    while True:
        # 选择邻域中最优的状态
        neighbor = min(next_states[swap_which](state), key=graph.weight)
        if graph.weight(neighbor) >= graph.weight(state):
            if min_count == 2:
                # 所有邻域都是最优，可以返回
                vsl.figure(counts, weights, '迭代次数', '总路径权值', '多邻域登山法优
                ↪ 化效果')
                return state
            else:
                # 如果没有最好的状态，开始检测是否是所有邻域都最优. 更换邻域
                min_count += 1
                swap_which = (swap_which + 1) % 3
        else:
            min_count = 0
            counts.append(count)
            weights.append(graph.weight(neighbor))
            state = neighbor
            count += 1
```

(2) 模拟退火算法

```python
def simulate_annealing(state, graph: NodeGraph):
    """
    模拟退火算法
    :param state: 初始状态
    :param graph: 图
    :return: 最优状态
    """
    maximum, minimum = random_maxmin(graph)
    temperature = maximum - minimum # 初温取极差
    end_temperature = 1 # 终温
    alpha = 0.98 # 降温系数
    balance_len = 1000 # Markov 链长度
    counts = []
    weights = []

    neighbor = {
        0: neighbor_swap,
        1: neighbor_reverse,
        2: neighbor_insert,
    }

    while temperature > end_temperature:

        for i in range(balance_len):
            new_state = neighbor[random.randrange(0, 3)](state) # 随机生成下一个状态
            ↪ (使用随机邻域)
            receive = False
            delta = graph.weight(new_state) - graph.weight(state)
            if delta < 0:
                # 永远接受更好的状态
                receive = True
            else:
                # 概率接受更差的状态
                prob = math.exp(-delta / temperature)
                if random.random() < prob:
                    receive = True

            if receive:
                state = new_state
                counts.append(len(counts) + 1)
                weights.append(graph.weight(state))
```

```
        temperature *= alpha
    vsl.figure(counts, weights, '迭代次数', '总路径权值', '模拟退火算法优化效果')
    return state
```

(3) 遗传算法

```
    def crossover(self):
        """
        交叉互换算法
        """
        for i in range(self.size // 5, self.size):
            if random.random() < self.crossover_p:
                j = np.random.randint(0, self.size // 5)
                # 子代取代基因较差的亲代
                self.members[i] = position_based_crossover(self.members[i],
                ↪  self.members[j])

    def select(self):
        """
        基因筛选算法
        """
        selected = np.zeros(self.members.shape, dtype=int)

        # 对种群按照适应度进行降序排序
        indices = self.fitness().argsort()
        self.members = self.members[indices][::-1]

        # 按照顺序分配几何分布概率
        p = 0.7
        prob = np.hstack([(1 - p) ** k * p for k in range(self.size - 1)])
        prob = np.hstack([prob, 1 - prob.sum()])

        # 轮盘赌
        for i in range(self.size):
            choice = np.random.choice(range(self.size), p=prob)
            selected[i] = self.members[choice]

        # 更新种群
        self.members = selected

    def mutate(self):
```

```
        """
        基因突变算法
        """
        # 生成随机数列代表是否基因突变
        p_rand = np.random.rand(self.size)
        for i in range(self.size):
            if p_rand[i] < self.mutation_p:
                # 随机抽取两个位置进行区间翻转
                self.members[i] = mutate_reverse(self.members[i])
```

```
def position_based_crossover(parent_1, parent_2):
    """
    基于位置的交叉互换算法
    :param parent_1: 一个亲本
    :param parent_2: 一个亲本
    :return: 两个亲本交叉互换生成的子代
    """
    length = len(parent_1)
    child = np.zeros(length, dtype=int)
    num = random.randint(1, length)
    pos, val = set(), set()
    work = 0

    # 随机选择区间进行交叉
    for i in range(num):
        while True:
            j = random.randint(0, length - 1)
            if j not in pos:
                break
        child[j] = parent_1[j]
        pos.add(j)
        val.add(parent_1[j])

    # 将剩余的位置按照原有顺序放上去
    for i in range(length):
        if i not in pos:
            while parent_2[work] in val:
                work += 1
            child[i] = parent_2[work]
            work += 1

    return child
```

三、 实验结果

采用了超算习堂提供的网站中的 kroB150 问题, 也就是说城市数量为 150.
随机产生初始状态, 如下图:

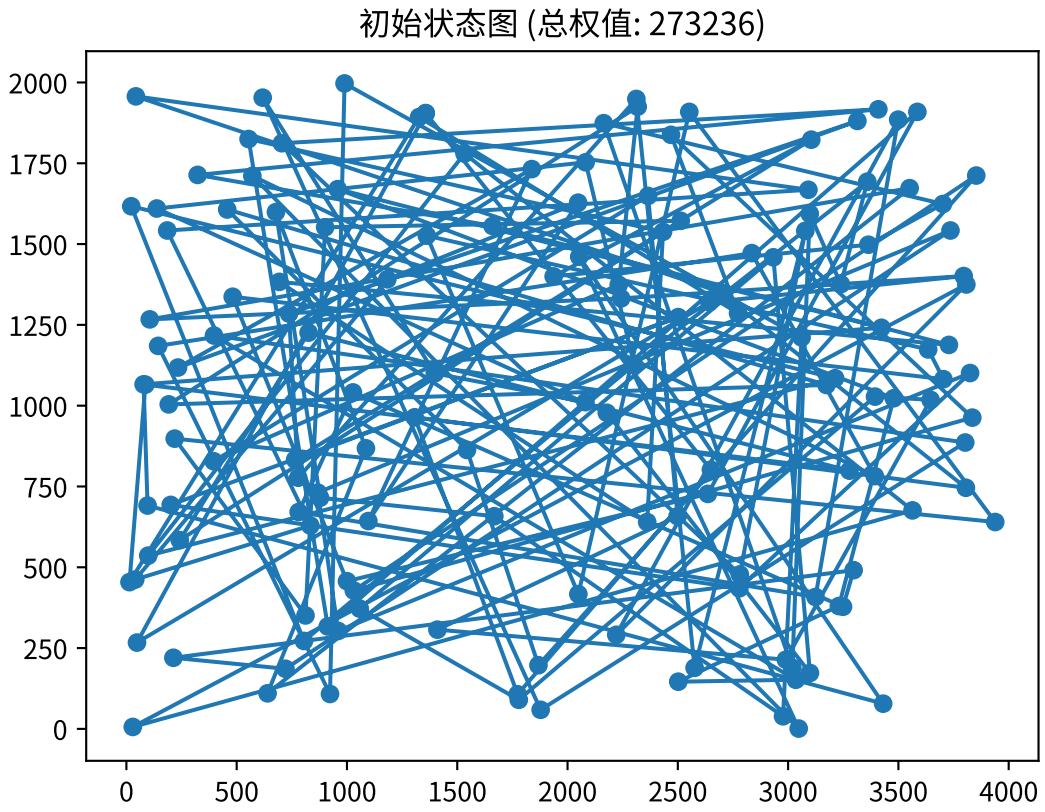
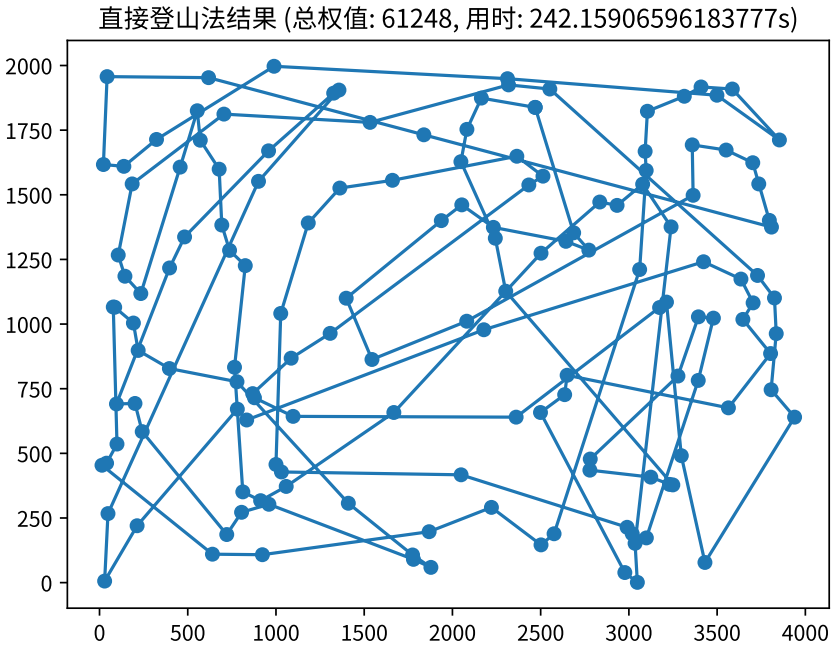
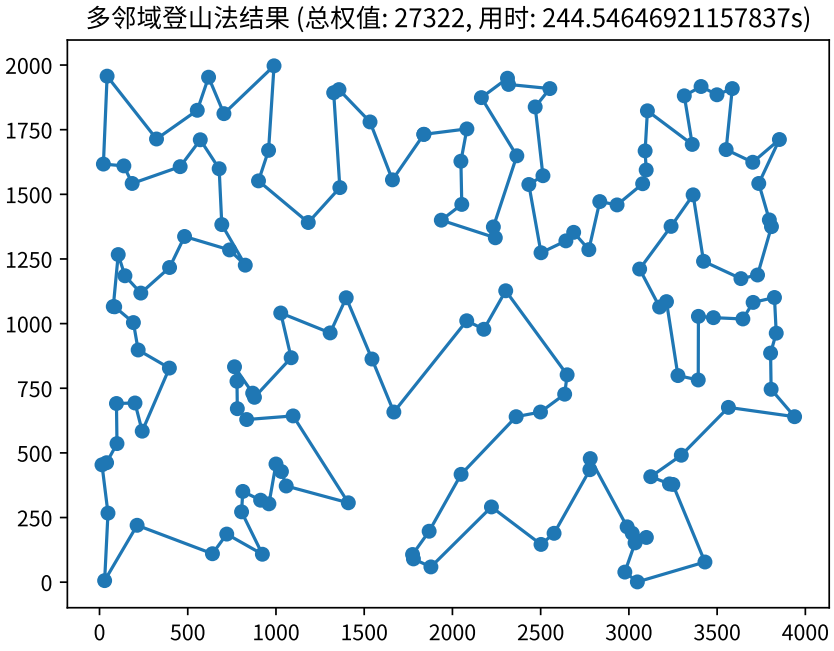


图 1: kroB150 问题的初始状态

可以看到这个路径存在非常多交叉, 显然不是最优解, 让我们用算法对它进行优化.
两种登山法的优化结果如下:



(a) 直接登山法



(b) 多邻域登山法

图 2: 登山法的优化结果

显然直接登山法的结果还存在很多交叉, 陷入了局部最优解; 而多邻域登山法不存在这个问题.

还可以观察到最优解权值随算法运行的趋势:

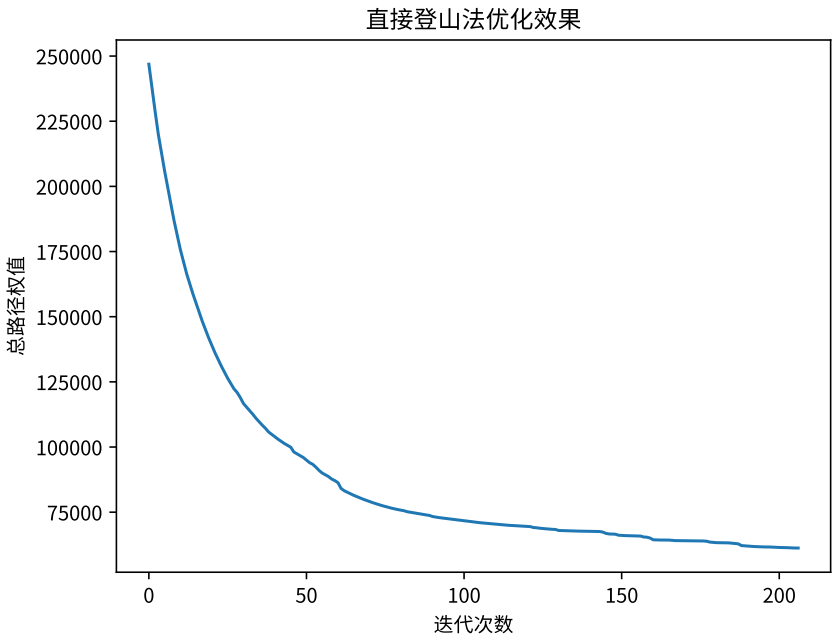


图 3: 直接登山法的趋势

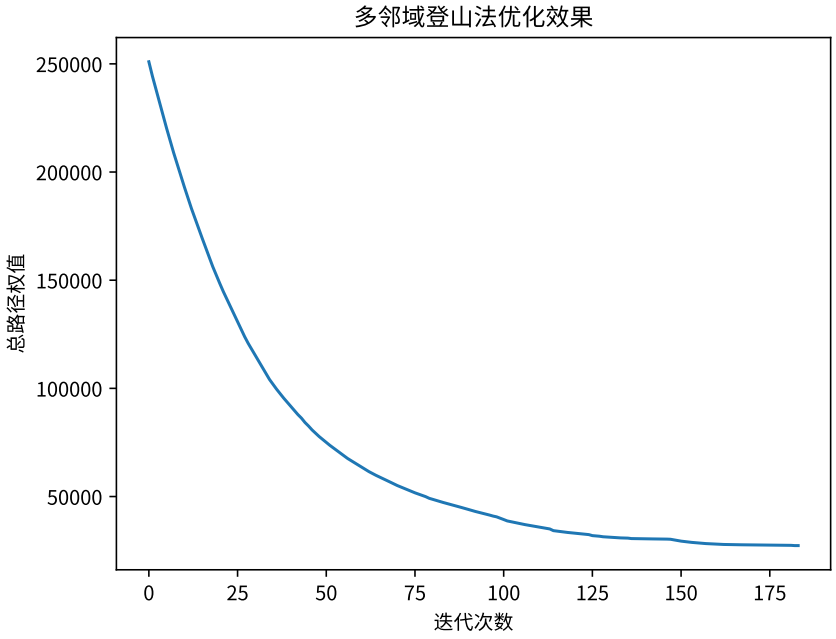


图 4: 多邻域登山法的趋势

可以发现多邻域登山法比起直接登山法更为灵活, 不太会受制于局部最优解.

再来看模拟退火算法和遗传算法的解:

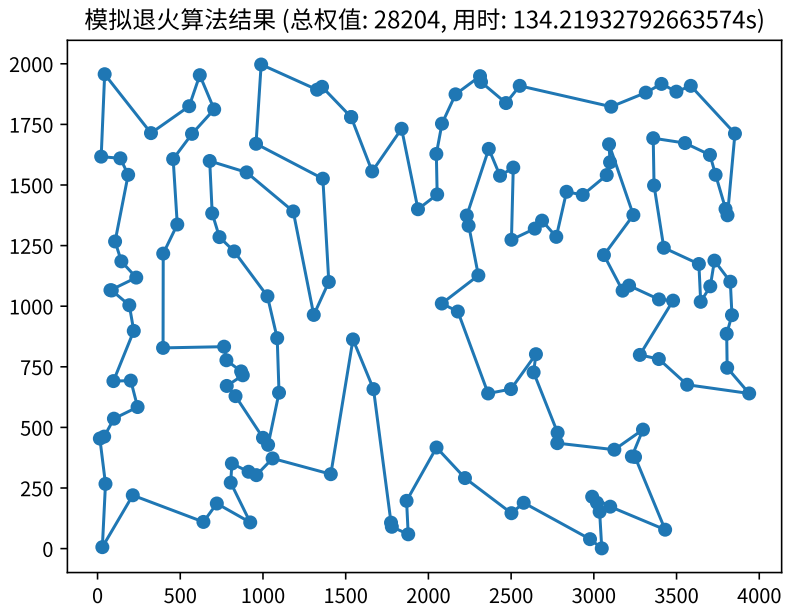


图 5: 模拟退火的结果

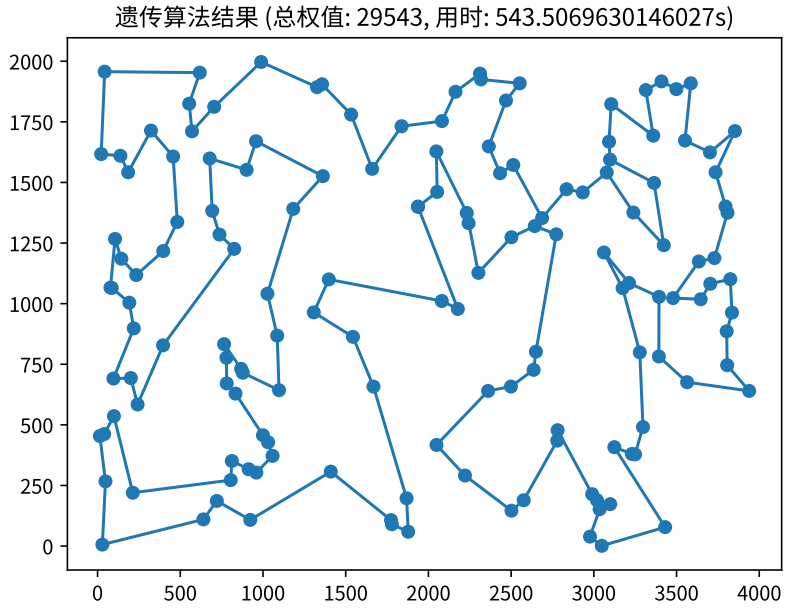


图 6: 遗传算法的结果

这两个结果都很优秀, 证明了算法的高效. 再来看算法运行时的趋势:

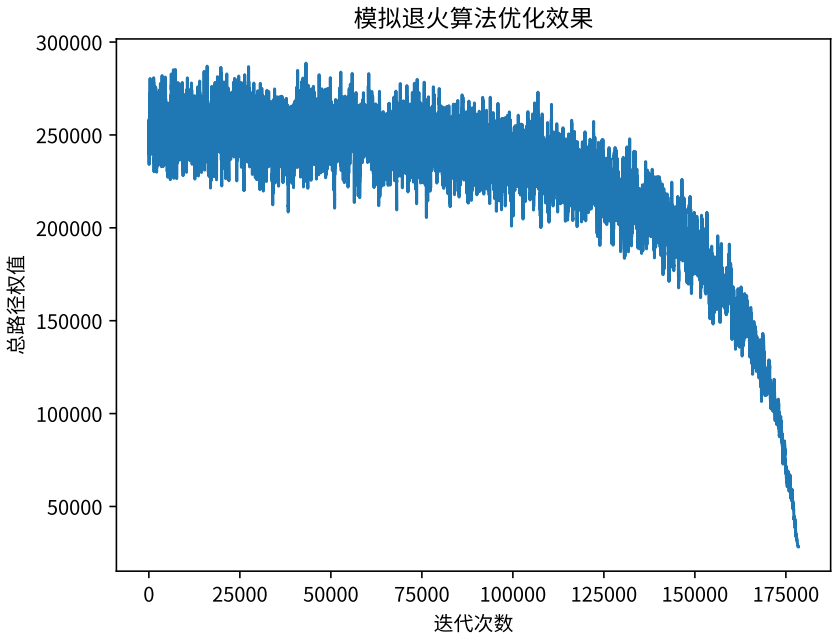


图 7: 模拟退火的趋势

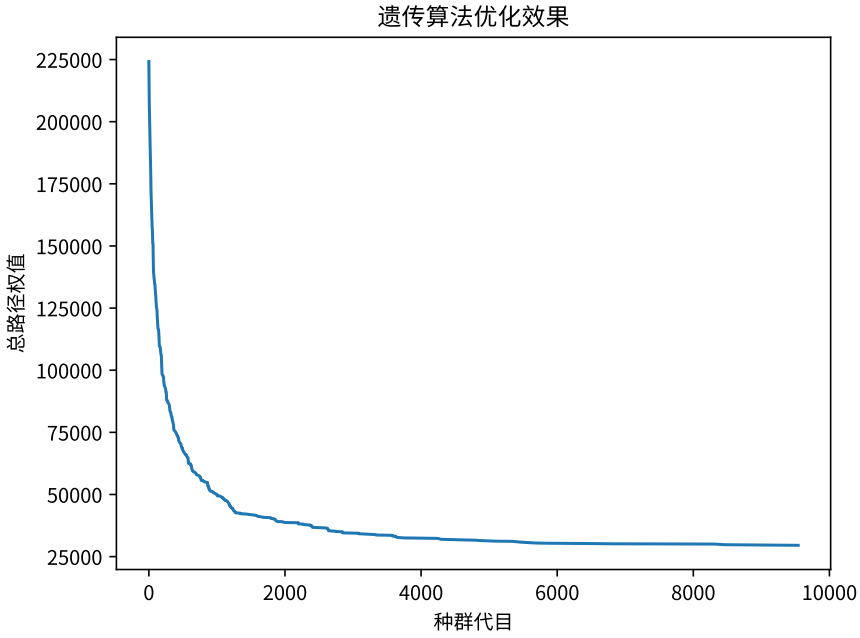


图 8: 遗传算法的趋势

这两个算法的不同显而易见: 模拟退火算法在开始时状态非常不稳定, 很容易接受更差的解, 但是到最后会愈发收敛; 而遗传算法会快速寻找到较优的种群, 然后依靠交叉互换和基因变异设法跳出局部最优解.

对于遗传算法, 其种群数量、基因突变率以及自然选择概率的设置非常重要, 我们需要在陷入局部最优解以及陷入庞大计算量两个极端情况中进行平衡.

单点交叉互换易于实现, 但是有可能造成 “近亲繁殖” 的现象导致算法陷入局部最优解; 而多点交叉互换与之相反.