苏州大学实验报告

院、系	计算机学院	年级专业	19 计科图灵	姓名	张昊	学号	1927405160
课程名称	人工智能与知识工程实验					成绩	
指导教师	陈文亮	同组实验者	无		实验日期	2021/12/2	

实 验 名 称 遗传算法

一. 实验题目

1. 求 y=10*sin(5*x)+7*abs(x-5)+10 最大值，0<=x<=10。
要求：精度为 0.01；种群个体数目不少于 6 个。

2. 求解 FSP 的遗传算法实例
Ho 和 Chang(1991) 给出的 5 个工件、4 台机器问题。加工时间表：

工件 j	t_{j1}	t_{j2}	t_{j3}	t_{j4}
1	31	41	25	30
2	19	55	3	34
3	23	42	27	6
4	13	22	14	13
5	33	5	57	19

用穷举法求得最优解：4-2-5-1-3，加工时间：213；
最劣解：1-4-2-3-5，加工时间：294；平均解的加工时间：265。
用遗传算法求解。选择交叉概率$P_c = 0.6$，变异概率$P_m = 0.1$，种群规模为 20。

二. 实验过程

1. 遗传算法工作流程的实现：
定义了一个遗传算法类 Genetic，由该类对象控制算法的流程。
构造函数中定义了一系列的运行参数，如下：

```
class Genetic:
    def __init__(self, population_size, decoder, encode_len, fit,
                 crossover_prob=0.5, mutation_prob=0.01, best_save_rate=0.1,
                 minimize=False, **cfg):
        self.population = []  # 种群
        self.population_size = population_size  # 种群大小
        self.crossover_prob = crossover_prob  # 交叉互换概率
        self.mutation_prob = mutation_prob  # 变异概率
        self.save_num = math.ceil(self.population_size*best_save_rate)#最佳个体保留个数
        self.decoder = decoder  # 解码器
        self.fit = fit  # 适应度计算函数
```

```
        self.encode_len = encode_len  # 编码长度
        self.history = {}  # 保存历史
        self.cfg = cfg
        self.population_print_mode = self.cfg.get('population_print_mode',
                                                  'init').lower()  # 运行记录打印
        self.minimize = minimize  # 是否进行最小化，默认最大化
        if minimize:
            self._min = max
            self._max = min
        else:
            self._min = min
            self._max = max
```

（1）初始化种群

调用时传入能够随机构造个体的生成器 generator，回调函数 cb（一般为初始化完成后打印）以及种群大小（可选）。使用 generator 生成目标大小的种群。

```
def init_population(self, generator, cb=None, size=None):  # 初始化种群
    self.population = []
    if size is None:
        size = self.population_size
    else:
        self.population_size = size
    for _ in range(size):
        self.population.append(generator())
    ret = self._eval_this_epoch(0)
    if cb is not None:
        cb.add(**ret)
        cb.print()
    if self.population_print_mode in ('init', 'each'):
        self._print_population(self.population, '[Initialize Population]', cb=cb)
```

（2）选择：轮盘赌法和最佳个体保存方法相结合

根据构造时传入的最佳个体保存率（最佳个体保存率为 0 则不保留最佳个体，退化为完全的轮盘赌方法），计算出每个迭代轮次中保存下来的最佳个体数量。首先找出最佳个体，将其复制到下一代中，之后再使用轮盘赌方法选择个体，直至达到目标种群数量。

```
def _do_select(self):  # 选择
    # 轮盘赌法和最佳个体保存方法相结合的选择
    selected = []
    fitness = [self.fit(self.decoder(item)) for item in self.population]
    best=self._max(self.population, key=lambda p: self.fit(self.decoder(p)))#最佳个体
    for _ in range(self.save_num):
        selected.append(copy(best))
    selected.extend(map(copy, random.choices(self.population, weights=fitness,
                                             k=self.population_size - self.save_num)))
    return selected
```

（3）交叉：对于每一个个体，根据交叉概率判断是否交叉。如果交叉则作为父本，等概率地在原种

群中选择母本，根据构造时传入的交叉方法（单点左侧交叉、单点左侧交叉、两点交叉），等概率地选择交叉点，使用个体定义的交叉方法来实现交叉。只取其中一个子代，替换掉父本得到新的种群。如果不交叉，则复制原种群中的父本得到新的种群。

注意保存下的最佳个体不参与交叉。（轮盘赌若选择了最佳个体，则应该参与交叉）

```
def _do_cross_over(self, population):  # 交叉
    selected = population[:]
    if self.population_size <= 1:
        return selected
    for index in range(self.save_num, self.population_size):  # 保留下来的最佳个体不交叉
        if not Genetic._random_decision(self.crossover_prob):
            continue
        while True:
            mother_index = random.randrange(0, self.population_size)
            if index != mother_index:
                break
        # 交叉类型：double 两点交叉，simple 单点右侧交叉，reversed_simple 单点左侧交叉
        cross_over_type = self.cfg.get('crossover_type', 'double')
        if cross_over_type == 'double':
            st = random.randrange(0, self.encode_len)
            ed = random.randrange(1, self.encode_len + 1)
            selected[index] = population[index].cross_over(population[mother_index],
                                                           start=st, end=ed)
        elif cross_over_type == 'simple':
            point = random.randrange(0, self.encode_len)
            selected[index] = population[index].cross_over(population[mother_index],
                                                           start=point)
        elif cross_over_type == 'reversed_simple':
            point = random.randrange(0, self.encode_len)
            selected[index] = population[index].cross_over(population[mother_index],
                                                           end=point)
        else:
            raise RuntimeError('Unsupported crossover type!')
    return selected
```

（4）变异

对每个个体根据变异概率判断是否发生变异，使用个体提供的变异方法，得到的新个体替换原种群中的个体得到新种群。注意保留下来的最佳个体不会变异。

```
def _do_mutation(self, population):  # 变异
    selected = population[:]
    for i in range(self.save_num, self.population_size):  # 保留下来的最佳个体不会变异
        if Genetic._random_decision(self.mutation_prob):
            point = random.randrange(0, self.encode_len)
            selected[i] = population[i].mutation(point=point)
    return selected
```

（5）执行遗传算法

提供迭代轮次的参数 epoch，迭代这些轮次，并返回末代种群的统计信息（最好，最坏等）。
在迭代过程中保存每轮的迭代评价结果，并输出到文件。

```
def construction(self, epoch: int, cb=None):  # 构建环境
    assert self.population_size == len(self.population)
    start = max(self.history.keys())
    for i in range(1, epoch + 1):
        next_population = self._do_select()
        next_population = self._do_cross_over(next_population)
        self.population = self._do_mutation(next_population)
        ep = start + i
        ret = self._eval_this_epoch(ep)
        if cb is not None:
            cb.add(**ret, epoch=ep)
        if self.population_print_mode == 'each':
            self._print_population(self.population,f'[Epoch {ep} Population]',cb=cb)
    if cb is not None:
        cb.print()
    if self.population_print_mode == 'init':
        self._print_population(self.population, '[Final Population]', cb=cb)
    return self.statistics()  # 统计末代种群
```

2. 个体的抽象

定义了一个抽象类 Individual，表示个体，具体问题继承该类并实现抽象方法即可。

```
class Individual(abc.ABC):
    def __init__(self, dna):
        self.dna = dna
    @abc.abstractmethod
    def copy(self):  # 复制个体
        pass
    @abc.abstractmethod
    def cross_over(self, another, start=None, end=None):  # 交叉互换
        pass
    @abc.abstractmethod
    def mutation(self, point=None):  # 变异
        pass
```

3. y=10*sin(5*x)+7*abs(x-5)+10 最大值

（1）编码方式：使用 10 个二进制位构成的无符号整数编码。

可以证明：$[0,10] \sim \{x \mid 0 \le x \le 1000, x \in N\} \sim [0,1] \sim \{x \mid 0 \le x \le 2^{10}-1, x \in N\}$，即这四个集合是等势的。由此，我们可以构造三个双射：

$$f_1(x) = [100x]; g_1(x) = \frac{x}{100}$$

$$f_2(x) = \frac{x}{1000}; g_2(x) = [1000x]$$

$$f_3(x) = [(2^{10}-1)x]; g_3(x) = \frac{x}{2^{10}-1}$$

因此，利用 $[0,10] \sim \{x \mid 0 \le x \le 2^{10}-1\}$ ，可以使用如下的映射来实现从 0<=x<=10（精度 0.01）到 10 个二进制位的 DNA 的编解码。

$$f(x) = \left[\frac{(2^{10}-1)[100x]}{1000}\right], 0 \le x \le 10$$

$$g(x) = \frac{\left[\frac{1000x}{2^{10}-1}\right]}{100}, 0 \le x \le 2^{10}-1, x \in N$$

这种编码的优点是每个 0 到 10 的精度为 0.01 的实数都会一一映射到一个 10 位的二进制数，不会产生二进制数没有与之对应的实数的问题。

（2）交叉：两点交叉（若未提供交叉点，则等概率地随机选择）

```
new_self_dna = self.dna[:start] + another.dna[start:end] + self.dna[end:]
```

（3）变异：随机选一个变异点，将其位反转

```
bit = self.dna[point]
if bit == '0':
    bit = '1'
else:
    bit = '0'
return Number(self.dna[:point] + bit + self.dna[point + 1:])
```

（4）适应度计算：直接使用函数表达式即可

4. FSP 的遗传算法实例

（1）编码：使用实值编码，优点是无需解码。将工序保存为五元组，作为 DNA。

这种编码的方式在交叉时需做出一定的修改，以免产生不合法的子代。

（2）交叉：使用次序交叉法。

次序交叉法：随机地在双亲中选择两个交叉点，再交换杂交段，其他基因的位置根据双亲相应基因的相对位置确定。这样来保证产生的子代仍然是合法的，即染色体各位无重复无遗漏。

下面举例说明该算法：

A = <0, 8, 5, 4, 7, 3, 1, 6, 9, 2>

B = <1, 5, 4, 6, 9, 0, 3, 2, 8, 7>

选择随机交叉点：start=4, end=8（不含） 交换杂交段：（ % 表示暂时留空）

A' = <%, %, %, %,|9, 0, 3, 2,|%, %>

B' = <%, %, %, %,|7, 3, 1, 6,|%, %>

之后从 A 第二个杂交点后开始循环遍历整个染色体：

9 2 0 8 5 4 7 3 1 6

去除杂交段的元素得到：

8 5 4 7 1 6

从 A' 第二个杂交点后依次填入空位，得到子代：

A' = <4, 7, 1, 6,|9, 0, 3, 2,|8, 5>

同理可得：（这里默认只产生一个子代，故不返回 B'）

B' = <4, 9, 0, 2,|7, 3, 1, 6,|8, 5>

实现如下：

```
crossover = list(another.dna[start:end])
new_self_dna = [None for _ in range(start)] + crossover + [None for _ in range(end, CODE_LEN)]
tmp = self.dna[end:] + self.dna[:end]
```

```
    unused = list(filter(lambda x: x not in crossover, tmp))
    for i, v in zip(list(range(end, CODE_LEN)) + list(range(start)), unused):
        new_self_dna[i] = v
    new_num = Schedule(new_self_dna)
    return new_num
```

（3）变异: 与后一个数字交换。如果是最后一位，则与前一位交换。

```
    if point == Schedule.artifact_number - 1:
        another = point - 1
    else:
        another = point + 1
    dna = list(self.dna)
    dna[point], dna[another] = dna[another], dna[point]
```

（4）适应度计算。使用公式：

$$c(j_1,1) = t_{j_1 1}$$

$$c(j_1,k) = c(j_1,k-1) + t_{j_1 k}, \quad k = 2,...,m$$

$$c(j_i,1) = c(j_{i-1},1) + t_{j_i 1}, \quad i = 2,..., n$$

$$c(j_i,k) = \max\{c(j_{i-1},k), c(j_i,k-1)\} + t_{j_i k}, \quad i = 2,..., n; k = 2,..., m$$

最大流程时间： $c_{\max} = c(j_n, m)$

注：在实现的过程中调换 max 和 min 函数的含义，从而在不改变适应度的前提下实现最小化。

三. 实验结果

- 运行环境：Python 3.8
- 依赖项：prettytable
- 运行方法：
  - 遗传算法实验 1：运行 task1.py
  - 遗传算法实验 2：运行 task2.py

实验 1：种群规模: 15, 交叉概率: 0.75, 变异概率: 0.1, 最佳个体保留率: 0.1

运行结果：（每次运行、各轮次的结果详见 task1.out 文件）

遗传算法运行的结果（迭代轮次 10 轮）

总运行次数	最好解	最坏解	平均	最好解的频率	最好解的平均迭代次数
20	0.29 (52.89713)	0.48 (48.394632)	2.51 (27.266301)	0.15	1.5

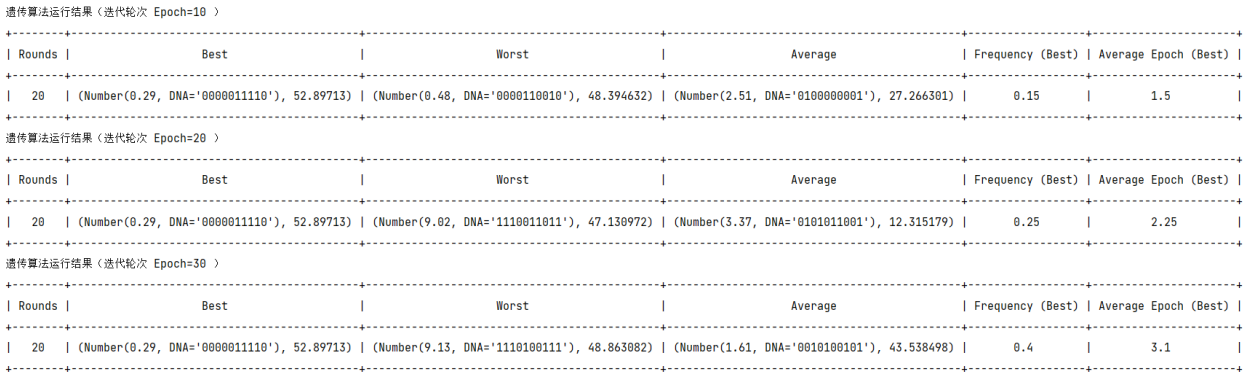
遗传算法运行的结果（迭代轮次 20 轮）

总运行次数	最好解	最坏解	平均	最好解的频率	最好解的平均迭代次数
20	0.29 (52.89713)	9.02 (47.130972)	0.29 (12.315179)	0.25	2.25

遗传算法运行的结果（迭代轮次 30 轮）

总运行次数	最好解	最坏解	平均	最好解的频率	最好解的平均迭代次数
20	0.29 (52.89713)	9.13 (48.863082)	1.61 (43.538498)	0.4	3.1

运行截图：



实验 2：种群规模: 20, 交叉概率: 0.6, 变异概率: 0.1, 最佳个体保留率: 0.1
运行结果：（每次运行、各轮次的结果详见 task2.out 文件）

遗传算法运行的结果（迭代轮次 10 轮）

总运行次数	最好解	最坏解	平均	最好解的频率	最好解的平均迭代次数
20	213	234	219.1	0.45	2.4

遗传算法运行的结果（迭代轮次 20 轮）

总运行次数	最好解	最坏解	平均	最好解的频率	最好解的平均迭代次数
20	213	226	216.45	0.55	1.9

遗传算法运行的结果（迭代轮次 30 轮）

总运行次数	最好解	最坏解	平均	最好解的频率	最好解的平均迭代次数
20	213	229	214.2	0.85	5.433333

运行截图：

```
/Users/holger/codes/GA/venv/bin/python3.8 /Users/holger/codes/GA/task2.py
遗传算法运行结果（迭代轮次 Epoch=10 ）
+--------+--------------------------+--------------------------+---------+-----------------+---------------------+
| Rounds |           Best           |          Worst           | Average | Frequency (Best) | Average Epoch (Best) |
+--------+--------------------------+--------------------------+---------+-----------------+---------------------+
|   20   | (Schedule(4-2-5-1-3), 213) | (Schedule(1-5-3-2-4), 234) |  219.1  |       0.45       |         2.4          |
+--------+--------------------------+--------------------------+---------+-----------------+---------------------+
遗传算法运行结果（迭代轮次 Epoch=20 ）
+--------+--------------------------+--------------------------+---------+-----------------+---------------------+
| Rounds |           Best           |          Worst           | Average | Frequency (Best) | Average Epoch (Best) |
+--------+--------------------------+--------------------------+---------+-----------------+---------------------+
|   20   | (Schedule(4-2-5-1-3), 213) | (Schedule(4-1-5-2-3), 226) |  216.45 |       0.55       |         1.9          |
+--------+--------------------------+--------------------------+---------+-----------------+---------------------+
遗传算法运行结果（迭代轮次 Epoch=30 ）
+--------+--------------------------+--------------------------+---------+-----------------+---------------------+
| Rounds |           Best           |          Worst           | Average | Frequency (Best) | Average Epoch (Best) |
+--------+--------------------------+--------------------------+---------+-----------------+---------------------+
|   20   | (Schedule(4-2-5-1-3), 213) | (Schedule(1-5-4-2-3), 229) |  214.2  |       0.85       |       5.433333       |
+--------+--------------------------+--------------------------+---------+-----------------+---------------------+
```

四. 实验总结和反思

本次实验加深了我对遗传算法的理解，让我明白了遗传算法中选择、交叉、变异的概念，并掌握了遗传算法的实现与应用。