机器作曲的遗传算法

程飞林　　冯皓宇　　林昊齐　　胡秦垚　　李嘉棠

2024 年 12 月 10 日

目录

1 小组分工

姓名	主要工作
程飞林	担任组长，组织讨论，分配工作，改进适应度函数和遗传算法，整理报告框架
冯皓宇	负责代码主体部分，撰写遗传算法，并对遗传算法进行分析
林昊齐	负责搜集遗传算法，并对遗传算法进行分析
胡秦垚	撰写适应度函数主体部分，探究适应度函数的调整对程序效率和实验结果的影响
李嘉棠	负责搜集初始音乐种群，探究适应度函数的调整对程序效率和实验结果的影响

2 程序框架

2.1 Individual 类简介

`Individual` 类用于表示种群中的一个个体[1]，其主要功能是生成音乐音符序列并计算其适应度。该类包含以下主要属性和方法：

- **属性**：
  - `HIGHEST_PITCH`：最高音高。
  - `NUM_PITCHES`：音高数量。
  - `DURATION`：每个音符的持续时间[2]。
  - `TICKS`：MIDI 文件的时间单位。
  - `SAMPLE_RATE`：音频采样率。
  - `PITCH_TO_FREQ`：音高到频率的映射。音高编码，该字典将音高值映射到相应的频率。例如，音高 1 对应的频率为 174.61 Hz，音高 27 对应的频率为 783.99 Hz。
  - `PITCH_TO_MIDI`：音高到 MIDI 编码的映射。用于生成 MIDI 文件。
  - `pitches`：音符序列。
  - `fitness`：个体的适应度。

[1] 每个个体由一个 32 位的数组表示，根据属性中的 `PITCH_TO_FREQ`，建立起与音高的映射。

[2] 在 32 位数组中我们用 0 表示音高的持续，即与前面第一个不是 0 的数对应的音高保持一致，从而能形成不同的节奏型。

- **方法**:
  - `__init__(self, pitches=None)`: 构造函数并计算适应度。支持两种方式创建个体: 如果未提供音符序列，则随机生成音符序列；如果提供了音符序列，则使用输入的音符序列。
  - `compute_fitness(self)`: 计算个体的适应度。
  - `generate_wave(self)`: 生成音频波形并保存为 WAV 文件。
  - `generate_midi(self)`: 生成 MIDI 文件。

2.2 Population 类简介

`Population` 类用于表示一个种群，其主要功能是通过遗传算法对种群进行进化。

2.2.1 主要属性和方法

- **属性**:
  - `NUM_IND`: 种群中的个体数量。
  - `CROSSOVER_IND`: 每次迭代中通过交叉生成的个体数量。
  - `individuals`: 种群中的个体列表。
- **方法**:
  - `__init__(self, pitches=None)`: 构造函数，初始化种群。如果未提供音符序列，则随机生成种群；如果提供了音符序列，则使用输入的音符序列。
  - `evolve(self, crossover, threshold=1500, N=1000)`: 通过指定的交叉方式对种群进行进化，直到达到适应度阈值或达到最大迭代次数。
  - `crossover_routellet(self)`: 轮盘赌选择交叉操作。
  - `crossover_tournament(self)`: 锦标赛选择交叉操作。
  - `crossover_random(self)`: 随机选择交叉操作。
  - `crossover_rank(self)`: 按排名选择交叉操作。

- `crossover_operation(self, parent1, parent2)`：对选择出来的亲代进行交叉产生两个子代。
- `mutation(self, prob)`：对种群进行变异操作。
- `selection(self, pop1, pop2)`：选择下一代种群。
- `plot(self, mean_fit, max_fit, min_fit, title)`：绘制适应度变化图。
- `get_best_clip(self)`：获取适应度最高的个体。
- `get_max_fitness(self)`：获取种群中最高的适应度。
- `get_min_fitness(self)`：获取种群中最低的适应度。
- `compute_prob(self, x)`：计算选择概率。
- `sort(self)`：按适应度对种群进行排序。

2.2.2 进化过程

`Population` 类通过遗传算法对种群进行进化，主要包括交叉（crossover）和变异（mutation）两个步骤。以下是详细介绍：

- **初始化种群**：在 `__init__` 方法中，初始化种群中的个体。如果未提供音符序列，则随机生成种群；如果提供了音符序列，则使用输入的音符序列。
- **进化**：在 `evolve` 方法中，通过指定的交叉方式对种群进行进化，直到达到适应度阈值或达到最大迭代次数。每次迭代包括以下步骤：
  - **交叉**：根据指定的交叉方式生成子代。
  - **变异**：对种群进行变异操作。
  - **选择**：选择下一代种群。
  - **记录适应度**：记录种群的平均适应度、最大适应度和最小适应度。

2.2.3 交叉（crossover）

`Population` 类提供了多种交叉方式，以下仅为简单介绍，在实验中有具体介绍：

- **轮盘赌选择（crossover_routellet）**：根据个体的适应度概率随机选择父代进行交叉。

- **锦标赛选择（crossover_tournament）**：随机选择若干个体进行锦标赛，选择适应度最高的两个个体作为父代进行交叉。
- **随机选择（crossover_random）**：随机选择两个个体作为父代进行交叉。
- **排名选择（crossover_rank）**：根据个体的排名概率选择父代进行交叉，排名越高的个体越容易被选择。

交叉操作在 `crossover_operation` 方法中进行。对于每对父代，随机选择两个交叉点，并交换这两个交叉点之间的基因片段，生成两个子代。

2.2.4 变异 (mutation)

`Population` 类提供了多种变异操作, 包括：半音变异，全音变异，八度变异，时值变异，重复变异，重排变异，移调变异，逆行变异和倒影变异。变异操作在 `mutation` 方法中进行。根据变异概率，随机选择变异操作并应用于种群中的个体。这些在实验部分有具体介绍。

2.2.5 选择 (selection)

选择操作在 `selection` 方法中进行。根据适应度选择下一代种群，包括以下步骤：

- **高适应度个体**：选择适应度最高的若干个体进入下一代。
- **随机选择个体**：从剩余个体中随机选择若干个体进入下一代。

通过上述步骤，`Population` 类实现了种群的进化过程，不断优化个体的适应度。

2.3 fitness 函数简介

2.3.1 初始设置的参数

`fitness` 函数中包含以下初始设置的参数：

- **main_pitch**：主音或调式，设为 8（C4）。
- **idl_mean**：理想均值，设为 12。
- **idl_variance**：理想方差，设为 12。
- **coeff_mean**：均值偏离惩罚系数，设为 8。
- **coeff_variance**：方差偏离惩罚系数，设为 10。

- **coeff_pitch_jump**：音高跳跃惩罚系数，设为 0.3。
- **coeff_rhythm**：节奏评分系数，设为 0.5。
- **coeff_mode**：调式符合奖励系数，设为 0.2。
- **coeff_melody**：旋律组合奖励系数，设为 0.4。
- **coeff_re**：旋律重复奖励系数，设为 0.3。
- **de_major_notes**：大调音阶，设为 [0, 2, 4, 5, 7, 9, 11]。
- **de_xmajor_notes**：非大调音阶，设为 [1, 3, -1, 6, 8, 10]。
- **major_notes**：选择调式，此处以 C 大调为例，设为 [8, 10, 12, 13, 15, 17, 19, 20]。
- **jump_weights**：音高跳跃权重，设为 {1: 0.2, 2: 1.0, 3: 0.8, 4: 1.0, 5: 0.4, 6: 0.4, 7: 0.9, 12: 0.4}。
- **JUMP_SCORE**：音高跳跃评分的总分，设为 100 分。
- **weights**：各评分标准的权重，包括节奏、均值方差、音高跳跃、音符种类、调式符合性、旋律合理性和旋律组合，初始权重均设为 1.0。
- **test_flag**：测试标志，用于控制是否输出调试信息，初始设为 `False`。

2.3.2 评分组成部分的各个函数

`fitness` 函数通过以下几个子函数对个体的音符序列进行评分：

- **rhythm(pitch)**：根据音符序列的节奏模式进行评分，奖励节奏重复和多样性，惩罚切分音结尾和节奏种类过少或过多。
- **various_average(pitch)**：根据音符序列的均值和方差进行评分，奖励接近理想均值和方差的序列，惩罚偏离理想值的序列。
- **pitch_jump(pitch)**：根据音符序列的音高间隔进行评分，奖励合理的音高跳跃，惩罚过大的跳跃和频繁的半音跳跃。
- **pitch_variety(pitch)**：根据音符序列中不同音符的种类进行评分，奖励适中的音符种类数量，惩罚种类过少或过多。

- **scale_in_major_notes(pitch)**：根据音符序列是否符合特定调式进行评分，奖励符合调式的音符，惩罚不符合调式的音符。
- **calculate_melodic_reasonableness(pitch)**：按小节评估音乐片段的音阶合理性，奖励合理的跳跃和平滑的方向变化。
- **melody(pitch)**：根据音符序列的旋律组合进行评分，奖励常见的旋律模式和合理的音阶组合。

`fitness` 函数首先对各评分标准的权重进行归一化，然后根据各评分标准对音符序列进行评分，并将各评分标准的得分加权求和，得到最终的适应度分数。该分数用于评估个体在遗传算法中的表现，适应度越高的个体在进化过程中越有可能被选择。

2.4 主程序简介

主程序部分主要用于初始化种群并执行遗传算法的进化过程，具体步骤如下：

- **导入模块**：首先导入 `Population` 类和 `fitness_test` 模块。
- **初始化种群**：创建一个 `Population` 对象 `population`，初始化种群。
- **执行进化**：调用 `population.evolve(crossover='random')` 方法，自定义交叉方式对种群进行进化。
- **获取最佳个体**：调用 `population.get_best_clip()` 方法，获取进化过程中适应度最高的个体 `best_clip`。
- **生成音频**：调用 `best_clip.generate_wave()` 方法，生成最佳个体的音频波形并保存为 WAV 文件。
- **输出音符序列**：打印最佳个体的音符序列 `best_clip.pitches`。
- **测试适应度**：将 `test_flag` 设为 `True`，并调用 `fitness_test.fitness(best_clip)` 方法，输出最佳个体的适应度分数。

通过上述步骤，主程序实现了种群的初始化、进化、最佳个体的选择和音频生成，并对最佳个体的适应度进行测试和输出。

3 遗传算法分析

3.1 亲本选择方式

在使用不同的算法进行选择的同时，我们采用了经过改造的“**精英主义**”策略，即新一代种群中将会保留父代种群中适应度最高的的数个个体, 以保证演化会朝着适应度更好的方向进行, 并随机选择子代种群中适应度最差的数个个体加入子代种群, 以维持子代种群的个体多样性。

- 1、轮盘赌算法 (routellet): 其算法流程如下
  - 将当前种群中的每个个体的适应度依照下述过程转化为概率：依照下述公式处理每一个个体的适应度 (其中 inf_{norm} 为适应度值的的无穷范数，即其中的最大绝对值。其可以防止在计算指数时出现数值溢出)

$$x = e^{\frac{fitness}{inf_{norm}}}$$

  - 将计算结果归一化，使之成为一个有效的概率分布。
  - 基于上述方法计算出的概率从种群中随机选择个体作为父代。
  - 然后我们使被我们选择的父代两两交叉得到子代种群。这样我们达成了根据个体的适应度概率随机选择父代进行交叉。
- 2、锦标赛算法 (tournamen): 其算法流程如下
  - 每次从种群中随机选择出 n 个个体。
  - 然后选择这 n 个个体中适应度最高者和次高者（即“在锦标赛中胜出”）。
  - 两者交叉得到子代。
  - 如此重复上述操作直到得到所需规模的子代种群。这样也达到了适应度较高的个体有更大概率产生子代的结果。
- 3、随机选择算法 (random): 该算法每次随机从种群中选取两个个体作为父代进行交叉得到子代，选择是不放回的，重复此过程直到得到所需规模的子代种群。
- 4、按照排名选择算法 (rank): 其算法流程如下:
  - 首先将种群中的个体按照适应度按从低到高进行排名，适应度最低的个体为第一名。
  - 然后将每个个体的排名除以所有个体排名的总和。这样，每个个体的排名就被转化为了它们被选择的概率。

- 基于上述方法计算出的概率从种群中随机选择个体作为父代，选择过程中不放回。
- 然后我们使被我们选择的父代两两交叉得到子代种群。这样我们达成了根据个体的适应度排名随机选择父代进行交叉。

3.2 遗传算法交叉操作简介

我们将交叉操作与基因中的染色体交换相类比，染色体交换可以发生在任意位点，我们也设计了可以随机生成交叉位点的算法。在每次交叉操作中，我们会随机选择两个交叉位点，然后将两个父代进行交叉操作得到两个子代。

3.3 遗传算法变异操作简介

`Population` 类提供了多种变异操作：

- **半音变异**：随机选择一个音符，升高或降低一个半音。
- **全音变异**：随机选择一个音符，升高或降低两个半音。
- **八度变异**：随机选择连续的四个音符，升高或降低一个八度。
- **时值变异**：随机选择一个音符，改变其时值。
- **重复变异**：随机选择连续的四个音符，重复这些音符。
- **重排变异**：随机选择四个小节，随机重排这些小节。
- **移调变异**：随机选择一个音符，升高或降低一个音程。
- **逆行变异**：将音符序列逆行。
- **倒影变异**：将音符序列倒影。

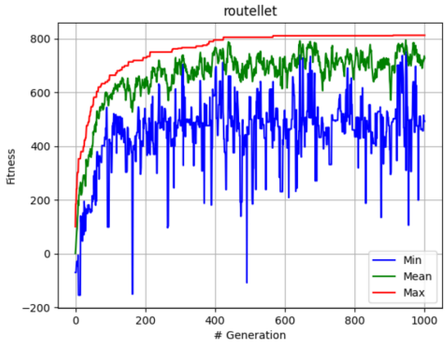
变异操作在 `mutation` 方法中进行。根据变异概率，随机选择变异操作并应用于种群中的个体。

3.4 遗传算法选择方式对于整体收敛方式的影响

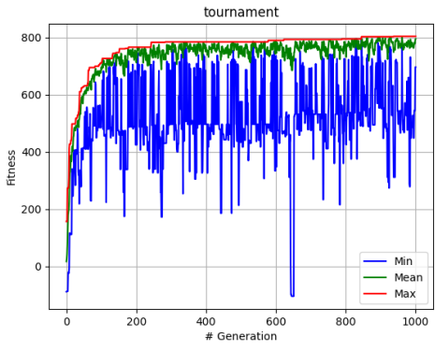
我们绘制了每个算法的种群适应度均值，最大值，最小值随迭代次数变换的图像以对不同遗传算法选择方式对于整体收敛效率的影响。

3.4.1 实验结果

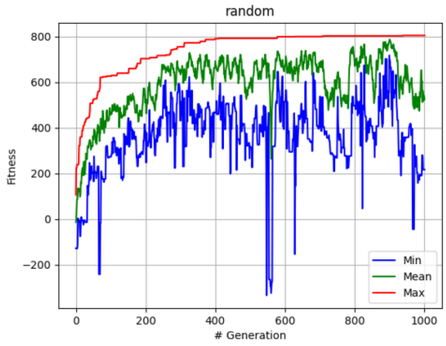
- 轮盘赌算法



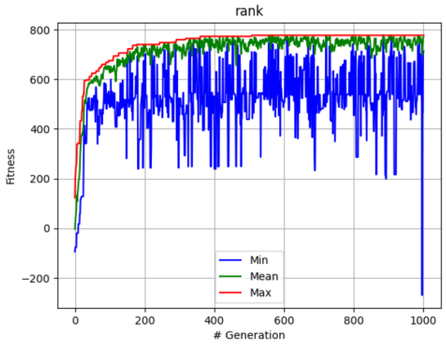
- 锦标赛算法



- 随机选择算法



- 排名选择算法



3.4.2 分析与讨论

比图像结果可以得出，随机选择算法效果是最差的，其均值上下波动，并未在 1000 轮内得到较为收敛的结果，其适应度最大值也在约 400 轮时才达到较为稳定的结果；但考虑到随机选择算法每次选择过程中高适应度个体并无优势，最大适应度达到稳定也应是我们的“精英主义”策略对于适应度较高的个体保留到下一代的结果。

而剩余三种算法中，按排名选择算法和锦标赛算法得到了较快的收敛结果，它们种群的最大值在约 200 轮就得到了较为收敛的结果，而轮盘赌算法种群最大值收敛较慢且其均值的收敛性也较差。在锦标赛算法中，适应度低的个体很难被选中，从而它们被淘汰的很快，这导致了锦标赛算法最大值和均值能够很快收敛；而对于轮盘赌算法，我们仍有的概率选择出适应度较低的个体，这也可能是由于轮盘赌选择的概率对不同适应度的个体区分不显著导致。对于按排名选择算法，不同适应度的个体被选择概率差异较为明显，故能得到较快收敛的结果。

种群最小值的跳跃变化显示了变异的无定向性和我们改进“精英主义”算法的作用，这可以很好地维持种群的个体多样性。

4 fitness 函数分析

4.1 测试条件

- crossover=’random’
- threshold=1000, N=500
- NUM_IND = 100

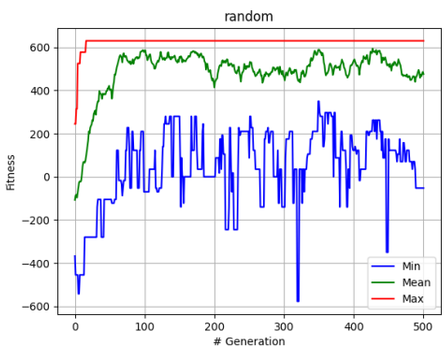
- CROSSOVER_IND = 70
- $mut\_prob = min(1.0, (threshold - self.get\_max\_fitness())/r + 0.1)$

4.2 实验结果及分析

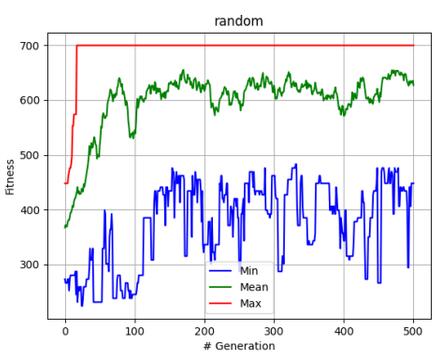
4.2.1 各子函数的实际影响

1. `rhythm`:
   在仅使用 rhythm 模块情况下，种群在 30 次左右便迭代收敛，产生的旋律也如预期一样仅有合理的节奏感，音高是随机产生的。

   而在不使用 rhythm 模块时，长音出现的概率（即 0 出现的概率）显著下降，这种现象应该是种群个体随机产生的机制和长音变异概率的机制导致。

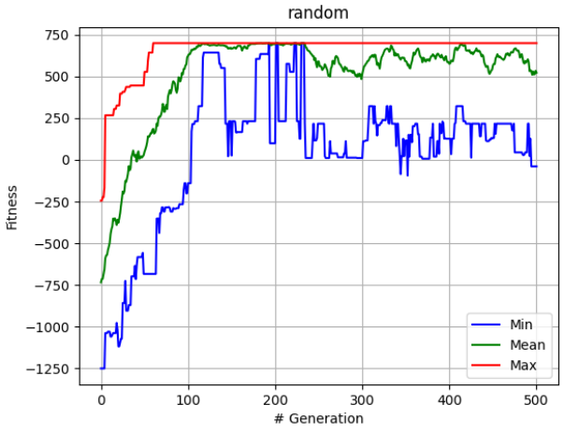


2. `various_average`:
   仅使用 various_average 模块情况下，种群在 30 次左右便迭代收敛，产生的旋律也如预期一样仅有合理的均值和方程，实际上旋律完全不合理。



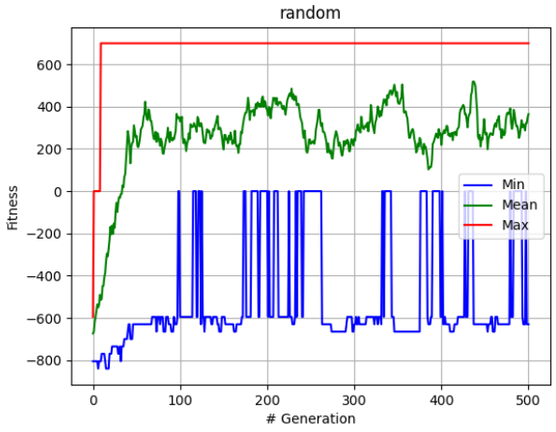
3. `pitch_jump`:

   在仅使用 pitc_jump 模块下，种群在 70 次左右迭代收敛，由于仅对大音程跳跃和半音跳跃有惩罚，产生的旋律基本上是大段重复——与重复变异的机制有关，音程则主要是一度和大二度。



4. `pitch_variety`:

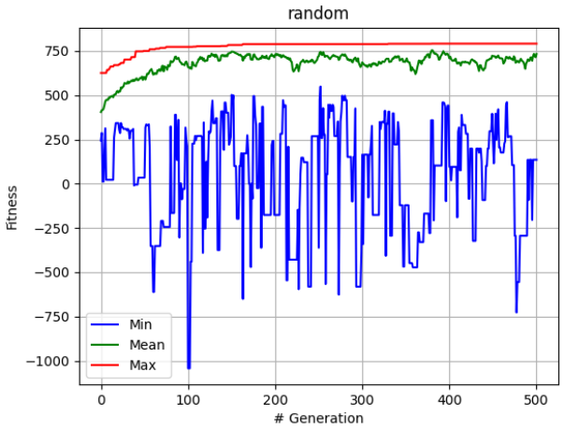
   在仅使用 pitch_variety 模块的情况下，种群在 20 次左右迭代收敛，产生的旋律仅有音高的种类数合理（在设定区间内）



5. `scale_in_major_notes`:

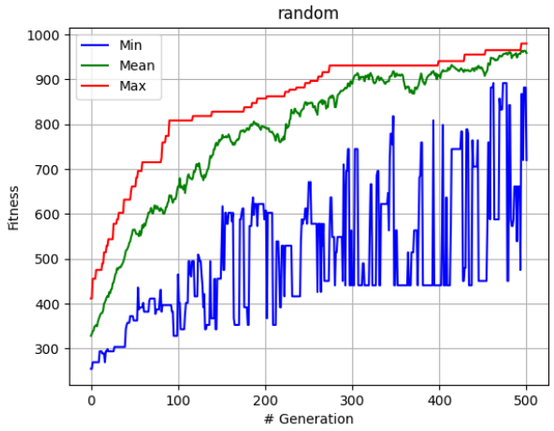
   在仅使用 scale_in_major_notes 模块的情况下，由于该部分模块原始得分自身偏高，threshold=1000 不适用，原始得分奖励系数调小至原先的 0.75，收敛需要约 150 代且不明显，产生的旋律音高几乎都是调式音（即默认 C 大调白键），但音程关系随机，方差接近随机方

差——可能与八度音同等处理有关。



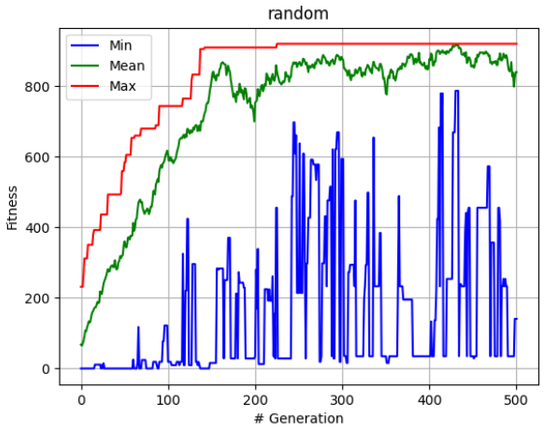
6. `calculate_melodic_reasonableness`:

   在仅使用 melodic_reasonableness 模块情况下，由于该部分模块原始得分自身偏高，threshold=1000 不适用，原始得分奖励系数同样调小至原先的 0.7，收敛速度往往大于 300 代，产生旋律几乎全是长音，且同样依赖重复变异机制，可能是算法发生了作弊，可以在更少的音符数情况下获得更高分。



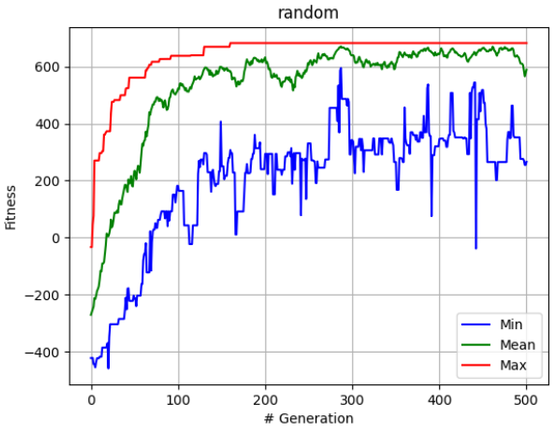
7. `melody`:

   在仅使用 melody 模块情况下，由于该部分模块原始得分自身偏高，threshold=1000 不适用，原始得分奖励系数同样调小至原先的 0.75，收敛速度约 200 至 400 代，产生的旋律依赖重复变异机制，四小节旋律会大部分一致——由于有重复奖励。



4.2.2 综合分析

前 4 个模块的收敛速度较快，可以认为他们在变异中基本为辅助作用，同时使用这 4 个模块，收敛速度上升至约 150 代，说明其互相之间也存在一定的条件牵制。



第 6 个模块和第 7 个模块实际上有不独立性，但如果第 6 模块权重过大，长音比例会明显变大，而如果第 7 模块权重过大，旋律重复现象又过于明显。巧妙地调和各部分的权重可以避免由于机制缺陷产生的过拟合效应。

附录

报告中的所有代码及参考输出，均包含在文件夹中，其中包括：

- `report.pdf`：本报告的 PDF 版本。
- `code` 文件夹：包含所有的源代码文件。
- `output` 文件夹：包含参考实验图像和音频文件。

参考文献

[1] Dušan Matić et al. Genetic algorithms for the traveling salesman problem: A review of representations and operators. *Journal of Theoretical and Applied Information Technology*, 19(1):1–10, 2010.