# 第 5 章

# 约 束 满 足

本章将介绍如何利用遗传算法解决约束满足问题。首先，阐述约束满足的概念以及它如何应用于搜索问题和组合优化，然后，将讨论几个约束满足问题的实际例子，这些例子使用基于 Python 的 DEAP 框架求解，包括著名的 $N$-皇后问题、护士排班问题以及图着色问题，最后，在讲解过程中，将学习硬约束和软约束之间的区别，以及如何将它们运用于求解的过程中。

本章主要涉及以下主题：

- 理解约束满足问题的本质；
- 在 DEAP 框架下使用遗传算法编程求解 $N$-皇后问题；
- 在 DEAP 框架下使用遗传算法编程求解护士排班问题；
- 在 DEAP 框架下使用遗传算法编程求解图着色问题；
- 理解硬约束和软约束的概念以及如何运用它们解决问题。

## 5.1 技术要求

本章将在 Python 3 中使用以下支持库：deap、numpy、matplotlib、seabornnetworkx。

## 5.2 搜索问题中的约束满足

第 4 章讨论了如何解决搜索问题，重点是对状态和状态之间的转换进行系统的评估，每一个状态转换通常包含一个成本或收益，并且搜索的目标是使成本最小化或收益最大化。约束满足问题是搜索问题的一种变体，它的状态必须满足一些约束或限制，如果能够将各种各样的违约转化为成本，并努力使成本最小化，那么求解约束满足问题就可以转化为求解一般搜索问题。

与组合优化问题一样，约束满足问题在人工智能、运筹学和模式识别领域也有着重要的应用，理解这些问题有助于解决初步看上去不相关的各种类型的问题。约束满足问题往往表现出很高的复杂度，这使得遗传算法成为解决约束满足问题的一种合适方法。

5.3 节介绍 $N$-皇后问题，阐述约束满足问题的概念，并用第 4 章讨论问题的类似方法，演示如何解决这些问题。

## 5.3　求解 $N$-皇后问题

经典的 $N$-皇后问题最初被称为八皇后难题，它起源于 8×8 棋盘上的国际象棋，八皇后难题的规则是在棋盘上放置 8 个皇后而不让其中任何两个互相攻击，即任意两个皇后不能处于同一行、同一列或同一斜线。$N$-皇后问题与 8 个皇后难题类似，只不过使用的是 $N \times N$ 棋盘和 $N$ 个棋后。

除了 $n=2$ 和 $n=3$ 的情况外，$N$-皇后问题对任何自然数 $n$ 都有一个解，对于原始的八皇后实例有 92 个解，如果认为对称解是相同的，则有 12 个独立解。其中的一个解如图 5-1 所示。

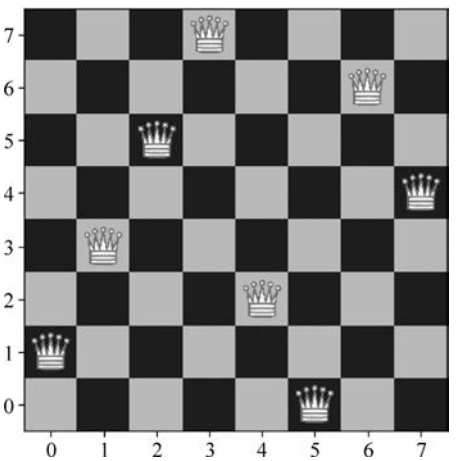


图 5-1　八皇后问题的 92 种解之一

通过应用组合数学，在 8×8 棋盘上放置 8 块棋子，可以产生组合数量达 4 426 165 368 种，但是，如果以不在同一行或同一列上放置两个皇后的方式创建候选解，则组合数量将显著减少到 8!(8 的阶乘)，总计 40 320。5.3.1 节将利用这一思想确定解的表示方式。

### 5.3.1　解的表示方式

在求解 $N$-皇后问题时，可以利用这样一个规则，即每一行或每一列只容纳一个皇后，这意味着可以将任何候选解表示为一个有序的整数列表或者一个索引列表，每个索引代表一个皇后占用了当前行和列。

例如，在 4×4 棋盘上的四皇后问题中，有索引列表(3,2,0,1)，它可以转化为如图 5-2 所示的以下位置。

(1) 在第一行位置 3 处放置皇后(第四列)。

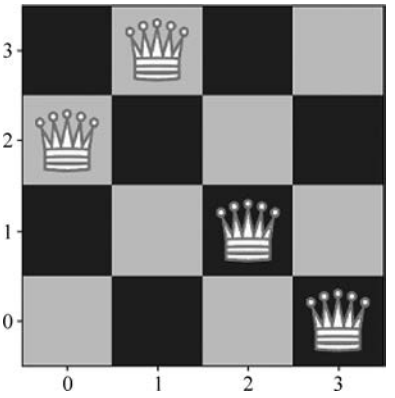


图 5-2　列表(3,2,0,1)表示的皇后排列图

(2) 在第二行位置 2 处放置皇后(第三列)。

(3) 在第三行位置 0 处放置皇后(第一列)。

(4) 在第四行位置 1 处放置皇后(第二列)。

类似地,另外一种索引排列(1,3,0,2)表示的候选解如图 5-3 所示。

在使用有序的整数列表或者索引列表方式表示的候选解中,唯一可能的约束冲突是两对皇后之间的斜线。例如,讨论的第一个候选解包含两个违约,如图 5-4 所示。同时,讨论的第二个候选解(见图 5-3)没有违约。

这意味着,当评价以有序的整数列表或者索引列表方式表示的解时,只需要找到并计算它们所代表的位置之间是否有共享**斜线**即可。

上述讨论的解的表示方式是 Python 类的核心部分,5.3.2 节将对 Python 类进行描述。

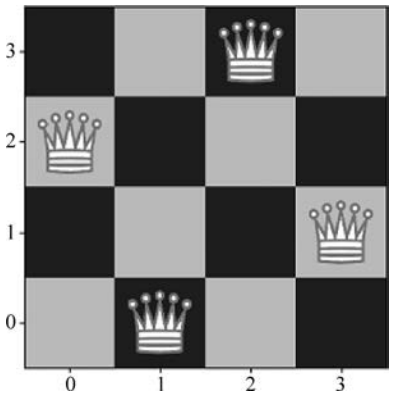


图 5-3　列表(1,3,0,2)表示的皇后排列图

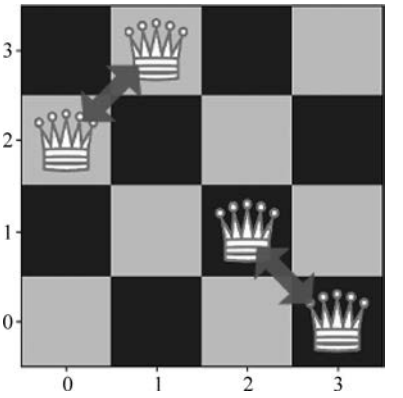


图 5-4　(3,2,0,1)表示的违反约束的皇后排列图

## 5.3.2　Python 对问题的表示方式

为了封装 *N*-皇后问题,这里创建了一个名为 NQueensProblem 的 Python 类,这个类可以在 queens.py 文件中找到,文件可在提供的示例代码中查看。

该类通过问题所需的阶数进行初始化,并提供以下通用方法。

(1) **getViolationsCount(positions)**:计算给定解中的违约数量,这个解如前所述,是用一个索引列表表示的。

(2) **plotBoard(positions)**:根据给定的解,绘制皇后在棋盘上的位置。

通过在该类 main 方法中创建一个八皇后问题类并测试以下候选解来练习该类。

```
(1, 2, 7, 5, 0, 3, 4, 6)
```

下面绘制候选解并计算违反约束的数量。输出结果如下：

```
Number of violations = 3
```

绘制图如图 5-5 所示，你能发现 3 种违约吗？

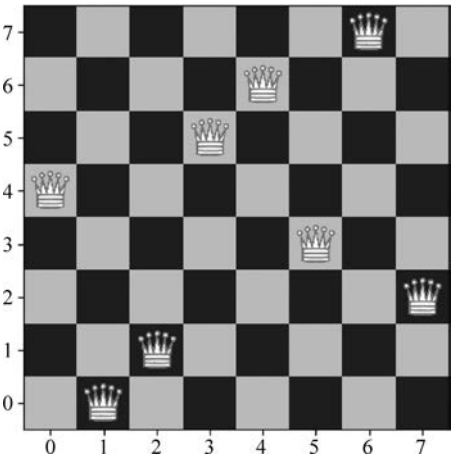


图 5-5　列表(1,2,7,5,0,3,4,6)表示的八皇后排列图

在下一小节中，将应用遗传算法来解决 *N*-皇后问题。

### 5.3.3　遗传算法求解 *N*-皇后问题

为了使用遗传算法解决 *N*-皇后问题，这里创建了名为 01-solve-N-queens.py 的 Python 程序，可在提供的示例代码中查看。

采用索引列表(或数组)作为 *N*-皇后问题的解的表示形式，这与第 4 章对**旅行商问题**(**TSP**)和**车辆路径问题**(**VRP**)所用的解的表示方式相似，所以与组合优化一样，*N*-皇后问题也可以使用类似的遗传算法求解。通过重用我们为 DEAP 遗传算法流程而创建的精英保留版程序，可以继续利用精英保留的优势。

下面步骤描述了求解的主要步骤。

(1) 程序根据要解决的问题规模创建 NQueensProblem 类实例：

```
nQueens = queens.NQueensProblem(NUM_OF_QUEENS)
```

(2) 当前目的是最小化违约次数(期望值为 0)，因此定义了一个单一的目标，即最小化适应度策略：

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

(3) 由于解是由一个有序的整数列表表示的，其中每个整数代表皇后的列位置，因此使用以下 toolbox 定义创建初始种群。

```
# create an operator that generates randomly shuffled indices:
toolbox.register("randomOrder", random.sample,
range(len(nQueens)),len(nQueens))

toolbox.register("individualCreator", tools.initIterate,
creator.Individual, toolbox.randomOrder)
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

(4) 实际适应度函数设置为每个解决方案中根据棋盘上皇后位置所计算的违约次数。

```
def getViolationsCount(individual):
    return nQueens.getViolationsCount(individual),
toolbox.register("evaluate", getViolationsCount)
```

(5) 对于遗传算子,使用规模为 2 的锦标赛选择,以及专门针对有序列表的交叉算子和变异算子。

```
# Genetic operators:
toolbox.register("select", tools.selTournament, tournsize = 2)
toolbox.register("mate",tools.cxUniformPartialyMatched,
indpb = 2.0/len(vrp))
toolbox.register("mutate",tools.mutShuffleIndexes,
indpb = 1.0/len(vrp))
```

(6) 继续采用精英策略,即名人堂(HOF)的成员(当前最优秀的个体)原封不动地传给下一代。正如在第 4 章中发现的,精英策略非常适合与规模为 2 的锦标赛选择搭配使用:

```
population, logbook = elitism.eaSimpleWithElitism(population,
                                        toolbox,
                                        cxpb = P_CROSSOVER,
                                        mutpb = P_MUTATION,
                                        ngen = MAX_GENERATIONS,
                                        stats = stats,
                                        halloffame = hof,
                                        verbose = True)
```

(7) 由于每个 $N$-皇后问题都有多个可能的解,因此需要输出名人堂的所有成员,而不仅仅是最前面的一个,如此就可知道找到了多少个有效的解:

```
print("- Best solutions are:")
for i in range(HALL_OF_FAME_SIZE):
    print(i, ": ", hof.items[i].fitness.values[0], " -> ", hof.items[i])
```

正如前面看到的,将八皇后实例的解的表示方式减少到大约 40 000 个组合,这使得它成为一个相对较小的问题。为了让问题更有趣,将规模增加到 16 个皇后,其中候选解的数量将达到 16!,即 20 922 789 888 000 这样一个巨大的数值。这个问题的有效解也相当多,将近 1500 万,将其与组合数进行比较,寻找一个有效的解就如同大海捞针。

在运行程序之前，设置算法常量，如下所示：

```
NUM_OF_QUEENS = 16
POPULATION_SIZE = 300
MAX_GENERATIONS = 100
HALL_OF_FAME_SIZE = 30
P_CROSSOVER = 0.9
P_MUTATION = 0.1
```

在这些设置下，程序运行后输出如下：

```
gen nevals min avg
0 300  3  10.4533
1 246  3  8.85333
..
23 250  1  4.38
24 227  0  4.32
..
- Best solutions are:
0 : 0.0 -> Individual('i', [7, 2, 8, 14, 9, 4, 0, 15, 6, 11, 13, 1, 3, 5,10, 12])
1 : 0.0 -> Individual('i', [7, 2, 6, 14, 9, 4, 0, 15, 8, 11, 13, 1, 3, 5,12, 10])
..
7 : 0.0 -> Individual('i', [14, 2, 6, 12, 7, 4, 0, 15, 8, 11, 3, 1, 9, 5,10, 13])
8 : 1.0 -> Individual('i', [2, 13, 6, 12, 7, 4, 0, 15, 8, 14, 3, 1, 9, 5,10, 11])
..
```

从输出中，可以看到在第 24 代中首次找到了一个解，其中适应度值显示为 0，这意味着没有违约。此外，输出的最优解表明，在运行期间发现了 8 种不同的解，这些解是名人堂中 0～7 的条目，其适应度值为 0，下一个条目的适应度值已经为 1，表示存在违约。

程序生成的第一幅图描绘了 16 个皇后在 16×16 棋盘上的位置，如图 5-6 所示，这是找到的第一个有效解(7,2,8,14,9,4,0,15,6,11,13,1,3,5,10,12)。

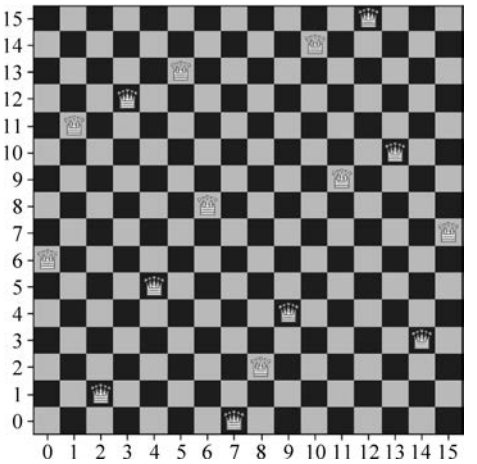


图 5-6　程序找到的一个可行的 16 皇后问题解的示意图

图 5-7 为包含了各代的最小适应度值和平均适应度值的曲线图。从这张图中可以看到，程序在早期(第 24 代左右)就找到了最佳适应度值 0，同时随着更多有效解被找到，平均适应度值不断下降。

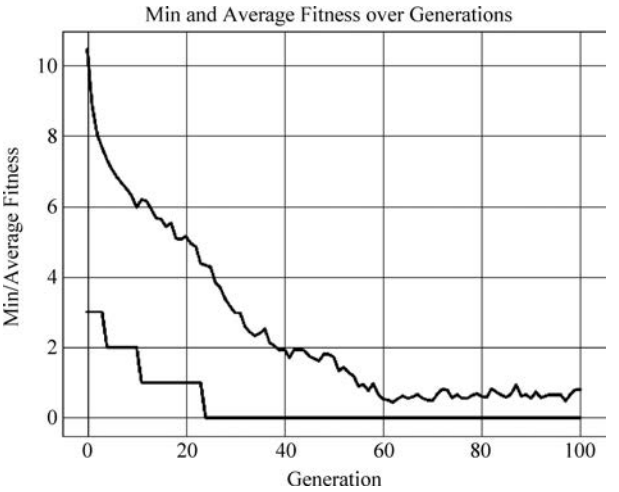


图 5-7 求解 16 皇后问题的迭代曲线

将 MAX_GENERATIONS 的值增加到 400 而不做其他任何更改，得到了 38 个有效的解。如果增加 MAX_GENERATIONS 到 500，名人堂的 50 名成员都包含了有效的解。建议尝试采用不同组合设置的遗传算法解决其他阶数的 $N$-皇后问题。

5.4 节将从棋盘上排列棋子过渡到工人工作时间表的安排问题上。

## 5.4 求解护士排班问题

假如由你负责安排本周医院护士的排班，一天有 3 个班次(早上、下午和晚上)，对于每一个班次，需要安排 8 个护士中的一个或多个在你的部门工作，如果这听起来像是一个简单的任务，那么看看医院的相关规则列表：

(1) 护士不允许连续工作两个班次。

(2) 护士每周工作不允许超过 5 个班次。

(3) 你所在科室的每班护士人数应符合以下规定：早班需要 2 或 3 名护士；午班需要 2～4 名护士；夜班需要 1 或 2 名护士。

此外，每个护士可以有值班偏好，例如，一名护士只愿意上早班，另一名护士不愿意上午班等等。

这个任务是**护士排班问题**(**Nurse Scheduling Problem**，**NSP**)的一个例子，它可以有许多变化，包括不同护士的不同专长，顶班(加班)的能力，甚至不同类型的值班(如 8 小时值班和 12 小时值班)。

可想而知，编写一个程序来安排值班可能是个好主意，为什么不应用遗传算法知识来实现这样一个程序呢？和前面一样，从问题的解的表示方式开始。

## 5.4.1　解的表示方式

为了解决护士排班问题，使用一个二进制列表（或数组）来表示值班安排，因为这种表示很直观，并且遗传算法可以轻松地处理这种表示方式。

可以用一个二进制字符串表示每个护士一周的 21 个轮班，值为 1 的代表一个班次，表示该护士被安排值班，例如，下面二进制列表：

```
(0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0)
```

这个二进制列表可以每 3 个值分为一组。如表 5-1 所示，每组代表该护士每周每天的工作班次。

**表 5-1　护士每周每天的工作班次**

| 周日 | 周一 | 周二 | 周三 | 周四 | 周五 | 周六 |
|---|---|---|---|---|---|---|
| (0,1,0) | (1,0,1) | (0,1,1) | (0,0,0) | (0,0,1) | (1,0,0) | (0,1,0) |
| 午班 | 早班，晚班 | 午班，晚班 | （无） | 晚班 | 早班 | 午班 |

然后将所有护士的值班表连接在一起，创建一个长的二进制列表，用于表示整个解。

当评估一个解时，这个长列表可以分解成每个护士的值班表，并且可以用于检查是否违反了约束，例如，表 5-1 中的护士值班示例中，包含两组连续的 1 值，表示护士连续值班（午班接着晚班，晚班接着早班）；通过统计该列表中的二进制值，可以计算同一个护士的每周值班数，其结果是 8 个值班；通过检查护士每天的班次和指定的优先班次，还可以很容易地检查是否符合其值班偏好。

最后，为了检查每班护士人数的限制，可以汇总所有护士每周的值班，并查找大于允许的极大值或小于允许的极小值的条目。

在算法实现之前，还需要讨论硬约束和软约束之间的区别。

## 5.4.2　硬约束与软约束

在解决护士排班问题时，有些约束代表医院的规则，它是不能被违反的，其中包含一个或多个违反规则的值班表都将被认为无效，这些规则被称为**硬约束**。

另一方面，可以将护士的偏好称为**软约束**，我们希望尽可能符合这些约束，尊重护士的偏好，不违反或少违反这些软约束的解要比含有更多违反行为的解相对更好，但违反这些软约束并不会使解无效。

在 *N*-皇后问题中，所有的行、列和斜线限制都是硬约束，如果找到一个违约次数为零的解，就得到一个有效的解。在这里，医院的规定是硬约束，护士的偏好是软约束。因此，实际上是在寻找一种解，这种解既不违反医院的任何规定，又尽量减少违反护士意愿的次数。

虽然处理软约束与任何优化问题所做的事情类似，即使其最小化，但如何处理伴随它们而来的硬约束呢？下面提供几种策略。

（1）找到解的特定表示方式（编码），以**消除违反硬约束的可能性**。在求解 $N$-皇后问题时，我们能够以一种消除 3 个约束中的两个约束（行和列）的方式来表示一个解，这大大简化了求解过程，但一般来说，这样的编码可能很难找到。

（2）在评估解时，**丢弃**违反任何硬约束的候选解。这种方式的缺点是同时丢失了解中包含的信息，而这些信息对于问题可能是有价值的，所以这种方式会大大减慢优化过程。

（3）在评估解时，**修复**违反任何硬约束的候选解。换句话说，找到一种方法处理解并对其修改，使其不再违反约束。但是，对于大多数问题，创建这样的修复程序是困难的，甚至是不可能的，同时，修复过程可能会导致重要的信息丢失。

（4）在评估解时，**惩罚**违反任何硬约束的候选解。这种方式会降低解的得分并使其不那么理想，但不会完全消除它。因此，它包含的信息不会丢失。实际上，这会导致硬约束被视为惩罚更重的软约束。使用这种方法时，面临的难题是如何找到适当的处罚幅度，过于严厉的惩罚可能导致这些解被排除，而惩罚太小可能导致这些解看起来是最优的。

本书案例选择应用第(4)种方法，对违反硬约束的行为进行惩罚，惩罚程度比软约束更大。这是通过创建一个成本函数来实现的，其中违反硬约束的成本大于违反软约束的成本，然后将总成本作为适应度函数并使其最小化。

### 5.4.3 基于 Python 的问题表示

为了封装前述的护士排班问题，创建了一个名为 NurseSchedulingProblem 的 Python 类，此类包含在 nurses.py 文件中，可在提供的示例代码中查看。

该类的构造函数接收 hardConstraintPenalty 参数，该参数表示违反硬约束的惩罚因子，接着继续初始化该函数的各个参数，来描述排班问题：

```
# list of nurses:
self.nurses = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']

# nurses' respective shift preferences - morning, evening, night:
self.shiftPreference = [[1, 0, 0], [1, 1, 0], [0, 1, 1], [0, 1, 0], [0, 0, 1], [1, 1, 1], [0,
1, 1], [1, 1, 1]]

# min and max number of nurses allowed for each shift - morning, evening,night:
self.shiftMin = [2, 2, 1]
self.shiftMax = [3, 4, 2]

# max shifts per week allowed for each nurse
self.maxShiftsPerWeek = 5
```

该类使用以下方法将给定的排班表转换为字典，表示每个护士的值班表：

```
getNurseShifts(schedule)
```

以下方法用于统计各种类型的违规行为：

```
countConsecutiveShiftViolations(nurseShiftsDict)
countShiftsPerWeekViolations(nurseShiftsDict)
countNursesPerShiftViolations(nurseShiftsDict)
countShiftPreferenceViolations(nurseShiftsDict)
```

此外，该类还提供以下公共方法。

（1）getCost(schedule)：计算给定排班表中各种违规的总成本，此方法使用 hardConstraintPenalty 变量的值。

（2）printScheduleInfo(schedule)：输出排班表和违规详细信息。

在该类的 main 方法中通过调用该类创建了一个护士调度问题的实例并且为该问题测试了随机的候选解，当 hardConstraintPenalty=10 时，结果输出如下所示：

```
Random Solution =
[1 0 0 0 0 1 1 1 0 0 0 1 1 0 0 0 1 0 1 0 1 1 0 1 0 1 1 1 0 1 0 1 0 1 1 1 1
 0 0 1 0 1 0 0 1 0 1 1 0 1 1 0 1 1 0 1 1 1 1 0 1 0 1 0 1 0 1 1 0 1 0 1 1 1
 1 0 1 0 0 0 1 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 0 1 0 0 1 1 0 1 1 1 1 0 0 0 0
 0 1 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1 1 1 1 0 0 0 0 1 1 0 1 1 0
 0 1 0 1 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 1 1]
 Schedule for each nurse:
 A : [1 0 0 0 0 1 1 1 0 0 0 1 1 0 0 0 1 0 1 0 1]
 B : [1 0 1 0 1 1 1 1 0 1 0 1 0 1 1 1 1 1 0 0 1 0 1]
 C : [0 0 1 0 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1 0 1]
 D : [0 1 0 1 1 1 0 1 0 1 1 1 1 1 0 1 0 0 0 1 1 0 1]
 E : [1 1 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 1 1 0 1 1]
 F : [1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 1 0 0 0 0 0]
 G : [0 0 0 0 1 1 0 0 1 1 1 1 1 0 0 0 0 1 1 0 1 1]
 H : [0 0 1 0 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 1 1]
 consecutive shift violations = 40

 weekly Shifts = [9, 13, 13, 12, 15, 5, 10, 9]
 Shifts Per Week Violations = 46

 Nurses Per Shift = [4, 2, 4, 1, 6, 6, 4, 4, 5, 4, 5, 5, 2, 4, 4, 4, 5, 3, 4, 3, 7]
 Nurses Per Shift Violations = 30

 Shift Preference Violations = 30

 Total Cost = 1190
```

从这些结果中可以明显看出，随机生成的解很可能会产生大量违约，从而产生较大的成本值，5.4.4 节尝试使用一种基于遗传算法的方法来最小化成本并消除所有硬约束违规。

## 5.4.4 遗传算法求解护士排班问题

为了使用遗传算法解决护士排班问题，创建了名为 02-solve-nurses.py 的 Python 程

序,可在提供的示例代码中查看。

由于护士排班问题的解的表示方式是一个二进制的列表(或数组),因此,可使用在第 4 章描述的 0-1 背包问题的遗传算法,此算法已经解决了多个问题。

主要求解过程描述如下。

(1) 程序创建一个 NurseSchedulingProblem 类的实例,期望值是 hardConstraintPenalty,该值由 HARD_CONSTRAINT_PENALTY 常量设置。

```
nsp = nurses.NurseSchedulingProblem(HARD_CONSTRAINT_PENALTY)
```

(2) 当前目标是最小化成本,因此定义一个单一目标,即最小化适应度值:

```
creator.create("FitnessMin", base.Fitness, weights = (-1.0,)
```

(3) 由于解是由 0 或 1 的列表表示,因此可使用以下 toolbox 定义创建初始化种群:

```
toolbox.register("zeroOrOne", random.randint, 0, 1)
toolbox.register("individualCreator", tools.initRepeat,
creator.Individual, toolbox.zeroOrOne, len(nsp))
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

(4) 实际适应度函数设置为每个解决方案中调度表的各种违约成本:

```
def getCost(individual):
    return nsp.getCost(individual),
toolbox.register("evaluate", getCost)
```

(5) 对于遗传算子,使用规模为 2 的锦标赛选择,以及适用于二进制编码的两点交叉和反转变异:

```
toolbox.register("select", tools.selTournament, tournsize = 2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit,indpb = 1.0/len(nsp))
```

(6) 仍采用精英保留的策略,即名人堂(HOF)成员(当前最优秀的个体)总是原封不动地传给下一代:

```
population, logbook = elitism.eaSimpleWithElitism(population,
toolbox, cxpb = P_CROSSOVER, mutpb = P_MUTATION,
ngen = MAX_GENERATIONS, stats = stats, halloffame = hof,verbose = True)
```

(7) 当算法结束时,输出找到的最优解的详细信息:

```
nsp.printScheduleInfo(best)
```

在运行程序之前,设置算法常量,如下所示:

```
POPULATION_SIZE = 300
```

```
P_CROSSOVER = 0.9
P_MUTATION = 0.1
MAX_GENERATIONS = 200
HALL_OF_FAME_SIZE = 30
```

然后，将违反硬约束的惩罚设置为1，这使得违反硬约束的成本与违反软约束的成本相似：

```
HARD_CONSTRAINT_PENALTY = 1
```

在这些设置下，运行程序产生以下输出：

```
-- Best Fitness = 3.0
-- Schedule =
Schedule for each nurse:
A : [0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0]
B : [1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0]
C : [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
D : [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0]
E : [0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0]
F : [0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0]
G : [0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1]
H : [1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0]
consecutive shift violations = 0
weekly Shifts = [5, 6, 2, 5, 4, 5, 5, 5]
Shifts Per Week Violations = 1
Nurses Per Shift = [2, 2, 1, 2, 2, 1, 2, 2, 1, 2, 2, 2, 2, 2, 1, 2, 2, 2,2, 2, 1]
Nurses Per Shift Violations = 0
Shift Preference Violations = 2
```

这似乎是一个很好的结果，因为最终只有3个违约，但是其中之一是违反每周值班的规定，即护士B每周被安排6个值班，超过了允许的最多5个值班，这足以使整个解无法被接受。

为了消除这种违约行为，将硬约束惩罚值增加到10：

```
HARD_CONSTRAINT_PENALTY = 10
```

现在，结果如下：

```
-- Best Fitness = 3.0
-- Schedule =
Schedule for each nurse:
A : [0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
B : [1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0]
C : [0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1]
D : [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0]
E : [0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
F : [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0]
G : [0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0]
H : [1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0]
```

```
consecutive shift violations = 0
weekly Shifts = [4, 5, 5, 5, 3, 5, 5, 5]
Shifts Per Week Violations = 0
Nurses Per Shift = [2, 2, 1, 2, 2, 1, 2, 2, 2, 2, 2, 1, 2, 2, 1, 2, 2, 2,2, 2, 1]
Nurses Per Shift Violations = 0
Shift Preference Violations = 3
```

同样，得到了 3 个违约，但这次都是违反软约束，这使得解是有效的。

图 5-8 描述了各代的最小适应度值和平均适应度值，表明在 40～50 代中，该算法能够消除所有硬约束违约，并且从那时起，每当消除一个软约束时，都有小的增量改进发生。

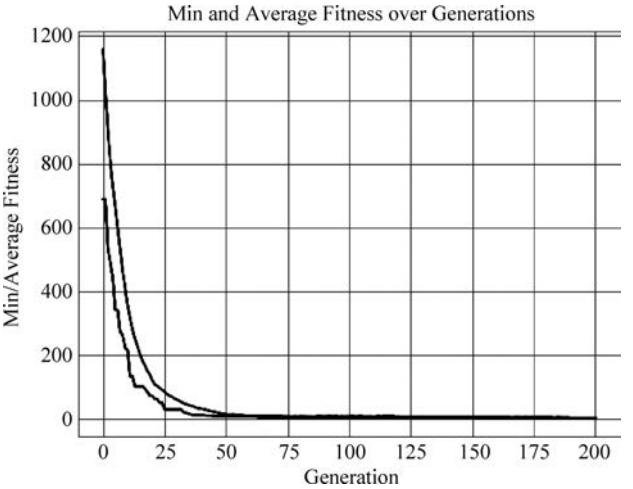


图 5-8 求解护士排班问题的迭代曲线

由此可知，在此案例中对违反硬约束的行为设定 10 倍的处罚就足够了，在其他问题中，可能需要设定为更高的值，我们鼓励通过改变问题的定义和遗传算法的设置来进行实验。

在下一个任务（图着色问题）中，将看到软约束和硬约束之间的权衡。

## 5.5 求解图着色问题

在图论的数学分支中，图是对象的结构化集合，表示对象之间的关系，在图中使用顶点（或节点）表示对象，使用边表示一对对象之间的关系，绘制图的一种常用方法是把顶点画成圆，边画成连接线。如图 5-9 所示为 Petersen 图，该图以丹麦数学家 Julius Petersen 的名字命名。

图是非常有用的，因为它可以表示和帮助我们研究各种各样的现实生活结构、模式和关系，例如社交网络、电网布局、网站结构、语言组合、计算机网络、原子结构、迁移模式等等。

图着色任务是为图中的每个节点分配一个颜色，规则是一对相连（相邻）的节点不能使用相同的颜色，这也被称为图的顶点着色问题。图 5-10 显示了正确着色的 Petersen 图形。

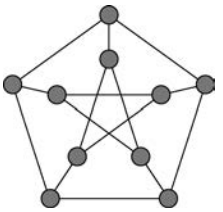

图 5-9　Petersen 图

（来源：https://commons.wikimedia.org/wiki/File:Petersen1_tiny.svg

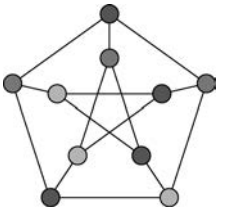

图 5-10　Petersen 图的正确着色

（来源：https://en.wikipedia.org/wiki/File:Petersen_graph_3-coloring.svg 公开发布）

颜色分配通常伴随着一个优化要求——即使用最少的颜色数量。例如图 5-10 所示，Petersen 图可以使用 3 种颜色正确地着色，但是使用两种颜色是不可能正确着色的，在图论中，这意味着 Petersen 图的色数是 3[①]。

为什么要关心图的节点着色呢？因为许多现实生活中的问题都可以转换成一个图进行表示，以这样的方式，图的着色将代表一个解。例如，学生的排课或员工的值班表都可以转换成一个图，其中相邻的节点表示引起冲突的排课或值班，这样的冲突可以是同时上课或者连续的值班（听起来是不是很熟悉？）。由于这种冲突，将同一个人分配给两个课程（或两个班次）将使排班无效，如果每种颜色代表不同的人，那么为相邻节点分配不同的颜色就可以解决这种冲突。本章开头的 $N$-皇后问题就可以表示为一个图着色问题，图中的每个节点代表棋盘上的一个正方形，共享一行、一列或斜线的每对节点由一条边连接，还有其他相关的例子，包括无线电台的频率分配、电网冗余规划、红绿灯计时、数独谜题求解等等。

希望以上描述能让大家相信图着色是一个值得解决的问题，与往常一样，这里先从为图着色问题的解设计合适的表示方式开始。

### 5.5.1　解的表示方式

在常用的二进制列表（或数组）基础上，可以使用一个整数列表，其中每个整数表示一个唯一的颜色，并且列表中的每个元素都匹配图的一个节点。

例如，Petersen 图有 10 个节点，则每个节点可以分配一个 0～9 的索引，然后使用 10 个元素的列表表示该图的节点着色。

例如，可以一起讨论下面特定的表示中有什么：

```
(0, 2, 1, 3, 1, 2, 0, 3, 3, 0)
```

下面详细说明：

（1）用整数 0、1、2、3 表示 4 种颜色。

（2）图的第一、第七和第十个节点用第一种颜色（0）着色。

① 扫描二维码查看彩图。

(3) 第三和第五个节点用第二种颜色(1)着色。

(4) 第二和第六个节点用第三种颜色(2)着色。

(5) 第四、第八、第九个节点用第四种颜色(3)着色。

为了对解进行评估,需要遍历每对相邻节点并检查它们是否使用同一种颜色,如果它们使用了同一种颜色,那么便是一个着色违约问题,因此需试图将违约的数量最小化到零,以实现图的正确着色。

但与此同时也在寻求最小化的颜色数量。如果恰好知道这个数字,那么可以使用与已知颜色数量一样多的整数值,但是如果不知道呢? 一种解决方法就是从估计(或猜测)所用颜色的数量开始,如果使用这个数字找到一个合适的解,那么可以减少这个数字再试一次;如果找不到解,那么可以增加这个数字再试一次,直到找到的最小数字,得到一个解。不过,也可以通过使用软约束和硬约束更快地得到这个数字,这个内容将在 5.5.2 节讲述。

## 5.5.2 使用硬约束和软约束解决图着色问题

在解决护士排班问题时,我们指出了硬约束(遵守这些约束才能将解视为有效)和软约束(努力最小化以获得更优解)之间的区别。在图着色问题中,颜色分配的规则(两个相邻的节点不可以有相同的颜色)是一个硬约束,只有将违反硬约束的次数减至零,才可以获得有效的解。

不过,可以把使用最小化颜色数量作为一个软约束引入,同时在不以违反硬约束为代价的前提下,尽量减少颜色的数量。

所以,可使用比估计值高的颜色数量来启动算法,并让算法最小化颜色数量,直到(理想情况下)达到实际的最小颜色数量。

与在护士排班问题中所做的类似,通过创建一个成本函数来实现这种方法,其中违反硬约束的成本大于使用更多颜色的成本,然后将总成本作为最小化的适应度函数,这个功能可以写进 Python 类中,5.5.3 节将对此进行描述。

## 5.5.3 基于 Python 的问题表示

为了封装图着色问题,创建了一个名为 GraphColoringProblem 的 Python 类。这个类可以在 graphs.py 文件找到,文件可在提供的示例代码中查看。

为了实现这个类,使用了 Python 的开源 NetworkX 包,它可以创建、操作和绘制图形。用作着色问题的图是 NetworkX 图形类的一个实例,所以不需要从头创建这个图,而是直接利用库中大量预先存在的图,如前面讲的 Petersen 图。

GraphColoringProblem 类的构造函数接收要着色的图作为参数。此外,它还接收 hardConstraintPenalty 参数,该参数表示违反硬约束的惩罚因子。

然后,构造函数创建一个图的节点列表和一个邻接矩阵,从中可以快速找出图中的任何两个节点是否相邻:

```
self.nodeList = list(self.graph.nodes)
self.adjMatrix = nx.adjacency_matrix(graph).todense()
```

该类使用下面方法计算给定颜色排列中的着色冲突数：

```
getViolationsCount(colorArrangement)
```

下面方法用于计算给定颜色排列中所使用的颜色数：

```
getNumberOfColors(colorArrangement)
```

此外，该类还提供以下公共方法，第一个方法计算给定颜色排列的总成本；第二个方法根据给定的颜色排列绘制着色节点的图。

```
getCost(colorArrangement)
plotGraph(colorArrangement)
```

在该类的 main 方法中，运用该类的方法创建了一个 Petersen 图的实例并且为该问题测试了随机颜色排列(最多 5 种颜色)的候选解，同时将 hardConstraintPenalty 的值设置为 10：

```
gcp = GraphColoringProblem(nx.petersen_graph(), 10)
solution = np.random.randint(5, size=len(gcp))
```

结果输出如下：

```
solution = [2 4 1 3 0 0 2 2 0 3]
number of colors = 5
Number of violations = 1
Cost = 15
```

由于这个特殊的随机解使用 5 种颜色并导致一种着色冲突，因此计算的成本是 15。这个解如图 5-11 所示，图中包含了一个违规着色。

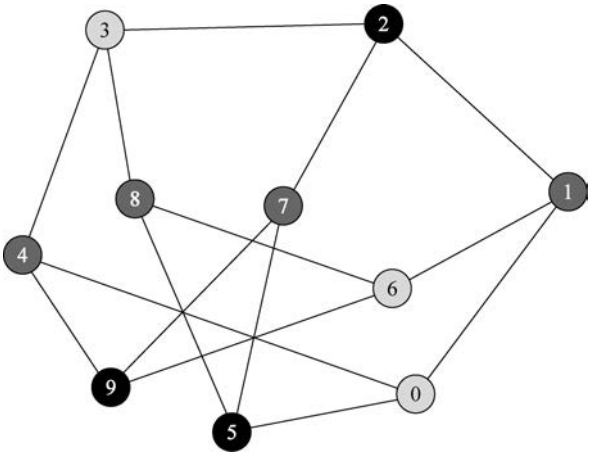


图 5-11 用 5 种颜色错误着色的 Petersen 图①

① 扫描二维码查看彩图。

5.5.4 节将应用遗传算法求解如何消除着色冲突同时减少使用的颜色数量的问题。

### 5.5.4 遗传算法求解

为了使用遗传算法求解图的着色问题，创建名为 03-solve-graphs.py 的 Python 程序，可在提供的示例代码中查看。

因为解的表示方式是一个整数列表，所以需要扩展一下使用二进制列表的遗传算法。以下步骤描述了求解要点。

(1) 创建一个 GraphColoringProblem 类的实例，实例参数为要求解的 NetworkX 图(本例中为熟悉的 Petersen 图)以及 hardConstraintPenalty 的期望值，该值通过设置 HARD_CONSTRAINT_PENALTY 的常量值完成：

```
gcp = graphs.GraphColoringProblem(nx.petersen_graph(),
HARD_CONSTRAINT_PENALTY)
```

(2) 当前目标是最小化成本，所以定义一个单一的目标，即最小化适应策略：

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,)):
```

(3) 由于解是由整数列表表示的，整数值代表参与的颜色，需要定义一个能够产生介于 0 到颜色数量减 1 的随机整数的生成器，生成的随机整数值表示参与的颜色之一。然后，定义一个解(个体)的生成器，它生成一个由这些随机整数组成，长度与给定图相匹配的列表，下面是为图中的每个节点随机分配颜色的方式。最后，定义创建整个种群的算子：

```
toolbox.register("Integers", random.randint, 0, MAX_COLORS - 1)
toolbox.register("individualCreator", tools.initRepeat,
creator.Individual, toolbox.Integers, len(gcp))
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

(4) 通过调用 GraphColoringProblem 类的 getCost()方法，将适应度评价函数设置为计算着色违规的总成本和颜色数量(与每个单独的解相关联)：

```
def getCost(individual):
    return gcp.getCost(individual),
toolbox.register("evaluate", getCost)
```

(5) 对于遗传算子，仍然可以使用选择和交叉操作，但是变异算子需要修改，位反转变异用于二进制列表的元素值在 0 和 1 之间反转，这里需要在一个允许范围内，将给定的整数更改为另一个随机生成的整数，mutUniformInt 算子就能够实现这个功能(只需要像前面的 integers 算子那样设置范围)：

```
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutUniformInt, low=0,
```

```
up = MAX_COLORS - 1, indpb = 1.0/len(gcp))
```

(6) 继续采用精英保留的方法,即名人堂(HOF)成员(当前最优秀的个体)总是原封不动地传给下一代:

```
population, logbook = elitism.eaSimpleWithElitism(population,
toolbox, cxpb = P_CROSSOVER,              mutpb = P_MUTATION,
ngen = MAX_GENERATIONS, stats = stats, halloffame = hof, verbose = True)
```

(7) 算法结束,输出最优解的详细信息,然后绘制图像。

在运行程序之前,设置算法常量值,如下所示:

```
POPULATION_SIZE = 100
P_CROSSOVER = 0.9
P_MUTATION = 0.1
MAX_GENERATIONS = 100
HALL_OF_FAME_SIZE = 5
```

此外,将违反硬约束的惩罚设置为 10,颜色数量设置为 10:

```
HARD_CONSTRAINT_PENALTY = 10
MAX_COLORS = 10
```

基于以上设置,程序输出如下:

```
-- Best Individual = [5, 0, 6, 5, 0, 6, 5, 0, 0, 6]
-- Best Fitness = 3.0
number of colors = 3
Number of violations = 0
Cost = 3
```

此结果意味着该算法能够使用 3 种颜色(用整数 0、5 和 6 表示)找到图的正确着色。正如前面提到的,实际的整数值并不重要,重要的是它们之间的区别。3 确实是已知的 Petersen 图的色数。图 5-12 为正确着色的 Petersen 图,说明了解的有效性。

图 5-13 描述了各代的最小适应度值和平均适应度值,可以看出,由于 Petersen 图规模相对较小,该算法很快就得到了解。

接下来尝试一个更大的图,用 5 阶的 Mycielski 图替换 Petersen 图,该图包含 23 个节点和 71 条边,已知的色数为 5:

```
gcp = graphs.GraphColoringProblem(nx.mycielski_graph(5),
HARD_CONSTRAINT_PENALTY)
```

使用与之前相同的参数,包括 10 种颜色的设置,得到以下结果:

```
-- Best Individual = [9, 6, 9, 4, 0, 0, 6, 5, 4, 5, 1, 5, 1, 1, 6, 6, 9, 5, 9, 6, 5, 1, 4]
```

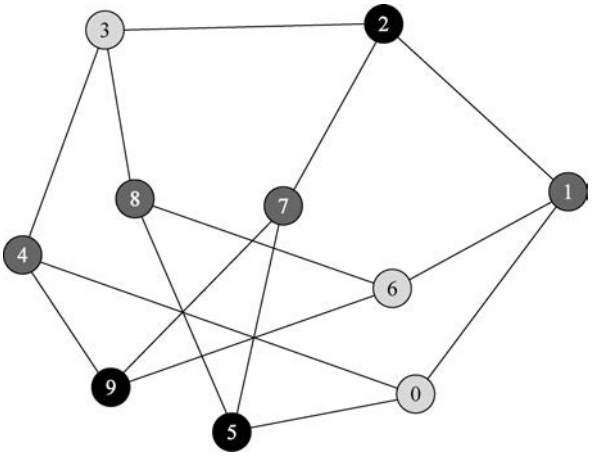


图 5-12　3 种颜色正确着色的 Petersen 图①

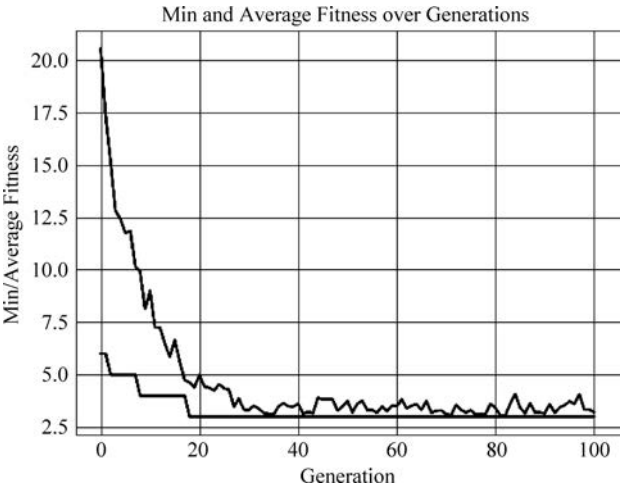


图 5-13　求解 Petersen 图的着色问题的迭代曲线

```
-- Best Fitness = 6.0
number of colors = 6
Number of violations = 0
Cost = 6
```

因为恰好知道这个图的色数是 5，所以虽然上面的解非常接近，但它并不是最优解。怎样才能实现最优解呢？如果事先不知道色数呢？实现这一点的一种方法是改变遗传算法的参数，例如，增加种群规模（可能还有名人堂规模）与/或增加迭代次数，另一种方法是重新开

① 扫描二维码查看彩图。

始相同的搜索，但颜色的数量要减少。因为算法找到了有 6 种颜色的解，所以我们将最大颜色数减少到 5 种，看看算法是否仍然可以找到有效的解：

```
MAX_COLORS = 5
```

为什么算法开始没有找到五色解而现在找到五色解呢？因为将颜色的数量从 10 个减少到 5 个，搜索空间大大减少了，在本例中，从 $10^{23}$ 个减少到 $5^{23}$ 个（因为在图中有 23 个节点），所以，即使在短时间运行和有限的种群规模下，算法也有更好的机会找到最优解。因此，虽然算法的第一次运行可能会接近解，但最好尝试不断减少颜色数，直到算法找不到更好的解为止。

在本例中，当算法从 6 种颜色开始时，可以很容易地找到五色解：

```
-- Best Individual = [0, 3, 0, 2, 4, 4, 2, 2, 2, 4, 1, 4, 3, 1, 3, 3, 4, 4, 2, 2, 4, 3, 0]
-- Best Fitness = 5.0
number of colors = 5
Number of violations = 0
Cost = 5
```

着色后的图见图 5-14。

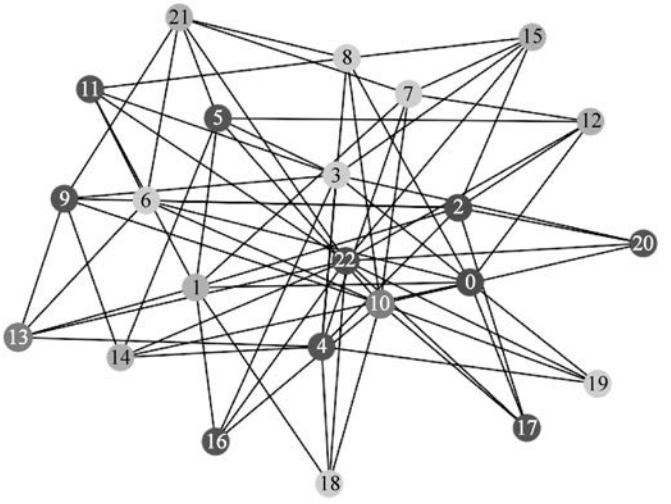


图 5-14　使用 5 种颜色正确着色的 Mycielski 图[①]

如果将最大颜色数减少到 4，我们将始终得到至少一个违约。

您也可以尝试使用该算法的其他参数设置处理图片。

① 扫描二维码查看彩图。

## 小结

本章首先介绍了约束满足问题，这是与先前研究的组合优化问题密切相关的一个问题。然后，探讨了 3 个典型的约束满足案例：*N*-皇后问题、护士排班问题和图着色问题，对于每一个问题，都遵循已经熟悉的流程，即找到一个解的适当的表示方式，创建一个封装问题和评估给定解的类，然后利用该类设计实现了一个遗传算法。最终，在熟悉硬约束与软约束的概念的同时，为这些问题找到有效的解。

到目前为止，我们一直在研究由状态和状态转换组成的离散搜索问题。第 6 章将研究连续空间中的搜索问题以证明遗传算法方法的通用性。

## 拓展阅读

[1] Prateek Joshi. Artificial Intelligence with Python[M]. USA: Packt, 2017.
[2] Boschetti A, Massaron L. Python Data Science Essentials[M]. 2nd. USA: Packt, 2016.
[3] NetworkX 教程[EB/OL]. https://networkx.github.io/documentation/stable/tutorial.html.