目录

第 10 章

进化算法

进化算法是一系列基于生物进化过程的全局优化算法，也是人工智能（AI）和软计算算法的子领域。在技术方面，它们是一类具有具有元启发式方法或随机优化特性的，基于群体的，通过不断试错来解决问题的算法。

进化算法领域支持许多新一代 AI 算法以动态方式生成新的处理规则，使算法能够自动适应新的数据集。这使机器学习能够适应需要处理的数据湖。

进化过程

基本的进化过程有三个步骤：

第一步

随机生成最初的种群个体，（第一代）。

第二步

评估种群中每个个体的适应值（时间限制、达到足够的适应值等）。

第三步

重复以下再生步骤，直到终止：

1. 选择最适合的个体进行繁殖（父母）。
2. 通过交叉和突变操作培育新个体，以生下后代。
3. 评估新个体的适应值。
4. 以新个体取代最不适合的个体。

进化算法技术主要涉及元启发式优化算法。从广义上讲，该领域包括我将在此章其余部分讨论的算法和方法。然而，这是一个正在大规模扩展的领域。

我的预测在五年内，我们周围的产品将由进化算法系统设计或测试。

蚁群算法

蚁群算法（ACO）是一种用来寻找优化路径的概率型算法。人工蚂蚁代表受真实蚂蚁行为启发的多代理方法。 生物蚂蚁基于信息素的通信方式往往是算法中用到的主要范式。

人工蚂蚁和局部搜索算法的组合已成为涉及一些图问题的众多优化任务的首选方法，例如车辆路线安排和网络路由。

从第 10 章的示例文件夹中找到示例代码：Chapter-10-01-Ant-Colony.ipynb

运行：

```
ant_colony = AntColony(distances, 1, 1, 100, 0.95, alpha=1, beta=1)

shortest_path = ant_colony.run()
```

Output is:

The results are:

```
([(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)], 16.0)

([(0, 1), (1, 3), (3, 4), (4, 2), (2, 0)], 17.0)

([(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)], 16.0)
```

(为了节省空间，我们移除了一些结果，在示例代码中可看到完整的结果数据集)

```
([(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)], 16.0)

([(0, 2), (2, 3), (3, 4), (4, 1), (1, 0)], 9.0)
```

下一步打印了最短路径:

```
print ("\n Shortest Path: {}".format(shortest_path))
```

结果是:

```
Shortest Path: ([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
```

文化算法

文化算法 （CA） 是进化算法的一个分支，其中除了种群组件之外，还有一个知识组件，称为信仰空间。从这个意义上说，CAs 可以被理解为遗传算法的预测方向的延伸。

信仰空间类别如下。

规范知识

种群组件中个体的可取的价值观的集合，例如，种群中的智能体（agent）的可接受行为。在人类社会中，规范性是将某些行动或结果指定为好的，可去的或允许的，而另一些行动或结果为坏的，不可取的过程。

规范过程通过使用机器学习生成的行为来精确演变其环境，在机器学习模型管理的解决方案中实现演化过程。ML 正在强制环境与规则规定的规范相匹配。

特定领域或通用领域知识

特定领域的发展学习理论认为，机器学习具有许多不连贯的、需要专业知识的结构，而不是一个内聚的知识结构。这种机器学习训练仅涵盖一个领域，不会影响另一个独立的知识领域。核心知识理论学家认为机器学习应该高度专业化并且独立于下一个机器学习模型和功能。

一般领域理论将进化知识视为机器学习模型和特征中一种相互选择的内聚知识结构，倾向于从激活学习的种子模型开始，由所有进化知识的总和创建一般领域学习模型。

情景知识

现在将讨论重要事件的具体示例，例如我们并不讨论解决方案的成功与否。

情景感知是网络、战争、经济、商业、关系等的重要组成部分，通常不可能使用所有可能的信息，因此基于情景的训练有助于在信息不完整的情况下做出决策，并能够感知比所看到更多的信息。

机器学习通常是使用分类学习实现的。这意味着一般来说，分类学习有两种不同的方法。

基于规则的机器学习制定一套规则来对情况进行分类，并存储规则作为所观察到的情况的类别。

基于信息的机器学习根据信息制定特征集以顺利对情况进行分类，并存储这些功能的是否存在（通过二进制的方法，就是置 0 还是置 1）作为所观察到的情况的类别。

时态知识

与机器学习在特征状态中观察到的时间或顺序相关的时间知识。

提示　阅读关于 Allen 的区间代数。微积分定义了时间间隔之间的可能关系，并提供了一个组合表，可以用作事件时间描述的基础。

以下为示例。

空间知识

空间知识是机器学习模型空间的搜索空间地形中所有可能状态的知识。它还可用于描述机器学习可以学习的信息的空间分布。

以下为示例。

分布式进化算法

Python 中的分布式进化算法（DEAP）是一个用于快速原型设计并测试想法的进化算法框架。它结合了实现最常见的进化计算技术如遗传算法、遗传编程、进化策略、粒子群优化、差分演化、流量和分布估计算法所需的数据结构和工具。

自 2009 年以来，它一直在拉瓦尔大学开发。

安装库：

conda install -c conda-forge deap

从第 10 章中的示例目录加载 Jupyter notebook：Chapter-10-02-Distributed-Evolutionary-Algorithms.ipynb

我建议你一步一步地执行代码，以深入了解它在技术上是如何工作的。

请注意，由于随机函数在这个进化过程中的性质，你可能得到的答案可能与下面打印的答案略有不同。

Results:

Population

[[9.521215056326895, 7.345527981285888], [4.8329426999153355,

6.20495203951839], [10.466145449758365, 7.299748993495174],

[9.756259353772032, 4.805180288174137], [8.9728086099149,

6.435624543518433]]

Log Book

gen evals avg std min max

0 5 0.0629536 0.0591294 0.00108562 0.16031

1 5 0.081051 0.0361767 0.0180054 0.126552

2 5 0.0916668 0.0385386 0.0312551 0.141644

3 5 0.0663885 0.0164441 0.0541062 0.0985648

4 5 0.0759395 0.0385668 0.0148436 0.136325

5 5 0.0851483 0.0419408 0.00816126 0.133337

<< Lines 6 to 996 - Removed due to volume of answer >>>

997 5 0.291057 0.150004 0.106485 0.44331

998 5 0.545407 0.481211 0.196919 1.4909

999 5 0.356074 0.108138 0.153049 0.476002

The best score is:

[8.647687147637413, 6.772257466226377]

打开第10 章中的: Chapter-10-03-DistributedEvolutionary-Algorithms-Loop.ipynb。

请注意主函数中的更改：

```
for i in range(10):
    pop, logbook, best = main_proc()
    print('### Population ###############################################')
    print(pop)
    print('### Best ###################################################')
    print(best)
```

执行代码以证明示例的随机性和进化过程得到解决的速度。

进化算法

我将引导您学习以下进化算法示例，以详细演示此过程的工作原理。

打开第 10 章目录中的代码：Chapter-10-04- Evolutionary-Algorithm-01.ipynb

这种进化算法使用灭绝的规则，它只让单个最好的成员生存到下一代。

加载库：

```
import numpy as np
import matplotlib.pyplot as plt
Setup the parameters
DNA_SIZE = 1 # DNA (real number)
DNA_BOUND = [0, 5] # solution upper and lower bounds
N_GENERATIONS = 200
POP_SIZE = 100 # population size
N_KID = 50 # n kids per generation
```

要查找此函数的最大值：

```
def F(x): return np.sin(10*x)*x + np.cos(2*x)*x
```

选择适应度不是 0 的。

```
def get_fitness(pred): return pred.flatten()
```

生成新成员：

```
def make_kid(pop, n_kid):
 # generate empty kid holder
 kids = {'DNA': np.empty((n_kid, DNA_SIZE))}
 kids['mut_strength'] = np.empty_like(kids['DNA'])
 for kv, ks in zip(kids['DNA'], kids['mut_strength']):
 # crossover (roughly half p1 and half p2)
 p1, p2 = np.random.choice(np.arange(POP_SIZE), size=2, replace=False)
 cp = np.random.randint(0, 2, DNA_SIZE, dtype=np.bool) # crossover points
 kv[cp] = pop['DNA'][p1, cp]
 kv[~cp] = pop['DNA'][p2, ~cp]
```

```
 ks[cp] = pop['mut_strength'][p1, cp]
 ks[~cp] = pop['mut_strength'][p2, ~cp]
 # mutate (change DNA based on normal distribution)
 ks[:] = np.maximum(ks + (np.random.rand(*ks.shape)-0.5), 0.)
# must > 0
 kv += ks * np.random.randn(*kv.shape)
 kv[:] = np.clip(kv, *DNA_BOUND) # clip the mutated value
 return kids
```

Extinct unfit population member:

```
def kill_bad(pop, kids):
 for key in ['DNA', 'mut_strength']:
 pop[key] = np.vstack((pop[key], kids[key]))
 fitness = get_fitness(F(pop['DNA']))
 idx = np.arange(pop['DNA'].shape[0])
 good_idx = idx[fitness.argsort()][-POP_SIZE:]
 for key in ['DNA', 'mut_strength']:
 pop[key] = pop[key][good_idx]
 return pop
```

Generate a population:

```
pop = dict(DNA=5 * np.random.rand(1, DNA_SIZE).repeat(POP_SIZE, axis=0),
 mut_strength=np.random.rand(POP_SIZE, DNA_SIZE))
```

Evolve through the generations:

```
%matplotlib inline
plt.ion()
```

```
x = np.linspace(*DNA_BOUND, 200)
for g in range(N_GENERATIONS):
 plt.plot(x, F(x))
 plt.scatter(pop['DNA'], F(pop['DNA']), s=200, lw=0, c='red', alpha=0.5)
 plt.text(0, -8, 'Generation=%.0f' % (g+1))
 plt.pause(0.05)
 kids = make_kid(pop, N_KID)
 pop = kill_bad(pop, kids) # good parent for elitism
```



```
plt.ioff()
plt.show()
```

结果：

第一代结果不太好（图 10-1）。

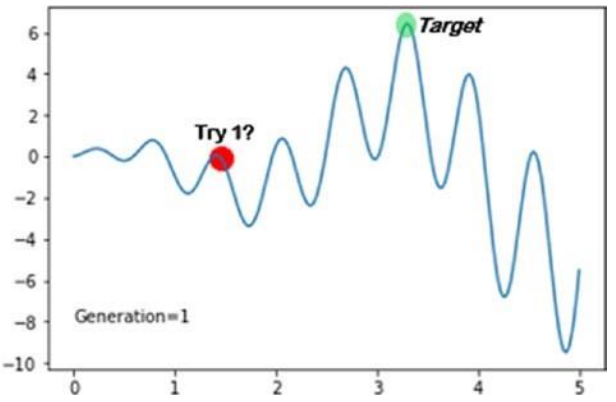

图 10-1. *第 1 代绘图*

红色的猜测点甚至不接近绿色目标。幸运的是，这只是进化开始发挥作用的地方。

第 1 代：

让我们监视几代人。

第 10 代：

在第 10 代期间，我们现在有多个潜在的下一代候选者（图 10-2）。

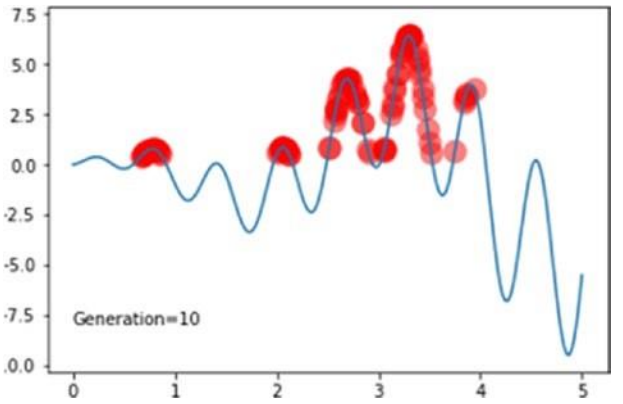

图10-2。 *第10 代绘图*

整个过程正在通过进一步的演变而得到改进。

第 200 代：

唯一的幸存者！参见图 10-3。

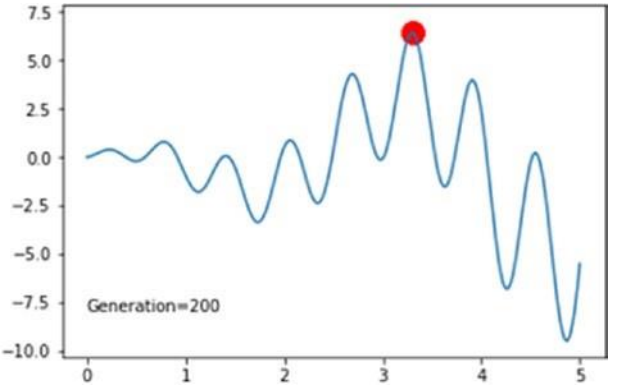

图10-3。 *第200 代图*

正中头彩 – 在 200 个周期或200代人内解决了问题。

提示 进化计算一般给人一种混乱的印象，但根据我的经验，该算法要么快速收敛，要么一直运行。后者通常是由于一个你还没有测量或尚未发现的特征。

打开第 10 章目录中的示例代码：Chapter-10-05- Evolutionary-Algorithm-02.ipynb

进化算法还使用灭绝规则，该规则仅保持当前状态和上一状态之间的最佳成员，以生存到下一代。现在我将向您展示一灭绝规则对几代人的影响。

```
import numpy as np

import matplotlib.pyplot as plt

DNA_SIZE = 1 # DNA (real number)

DNA_BOUND = [0, 5] # solution upper and lower bounds

N_GENERATIONS = 200

MUT_STRENGTH = 5. # initial step size (dynamic mutation strength)

def F(x): return np.sin(10*x)*x + np.cos(2*x)*x # to find the maximum

of this function

# find non-zero fitness for selection

def get_fitness(pred): return pred.flatten()

def make_kid(parent):

 # no crossover, only mutation

 k = parent + MUT_STRENGTH * np.random.randn(DNA_SIZE)

 k = np.clip(k, *DNA_BOUND)

 return k

def kill_bad(parent, kid):
```

```
 global MUT_STRENGTH
 fp = get_fitness(F(parent))[0]
 fk = get_fitness(F(kid))[0]
 p_target = 1/5
 if fp < fk: # kid better than parent
 parent = kid
 ps = 1. # kid win -> ps = 1 (successful offspring)
 else:
 ps = 0.
 # adjust global mutation strength
 MUT_STRENGTH *= np.exp(1/np.sqrt(DNA_SIZE+1) * (ps - p_target)/
(1 - p_target))
 return parent
parent = 5 * np.random.rand(DNA_SIZE) # parent DNA
plt.ion() # something about plotting
x = np.linspace(*DNA_BOUND, 200)
for g in range(N_GENERATIONS):
 kid = make_kid(parent)
 py, ky = F(parent), F(kid) # for later plot
 parent = kill_bad(parent, kid)
 plt.cla()
 plt.plot(x, F(x))
 plt.scatter(parent, py, s=200, lw=0, c='red', alpha=0.5,)
```

```
    plt.scatter(kid, ky, s=200, lw=0, c='blue', alpha=0.5)
    plt.text(0, -7, 'Mutation strength=%.2f' % MUT_STRENGTH)
    plt.text(0, -8, 'Generation=%.0f' % (g+1))
    plt.pause(0.05)
plt.ioff()
plt.show()
```

结果：

第 1 代（ 图 10-4）。

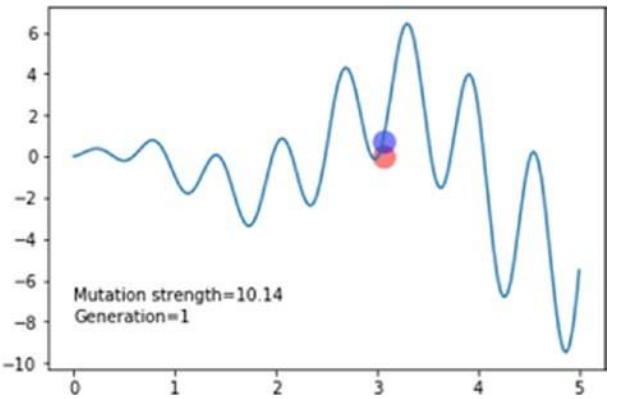

图 10 - 4 。 第 1 代图 -多代

让我们监视几代人。第10代（图10-5）：

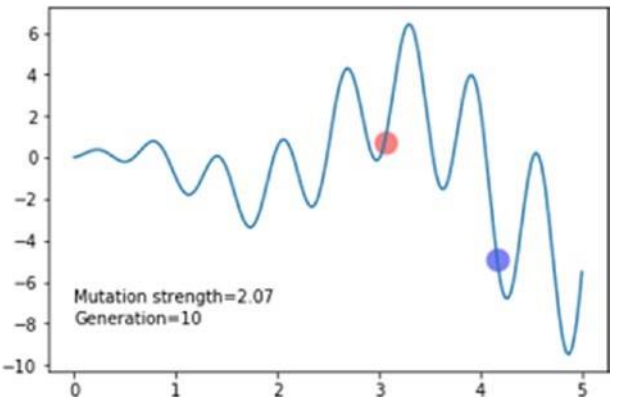

图10-5。 第 10 代图 - 多代

看起来更糟!!!

出什么事了...？

不，随机突变帮助解决局部最优。这是此版本的算法的主要优点。

此算法将得到最佳的解决方案，因为生成的负面影响将在一代内找到之前更好的父代。

第 200 代（ 图 10-6）。

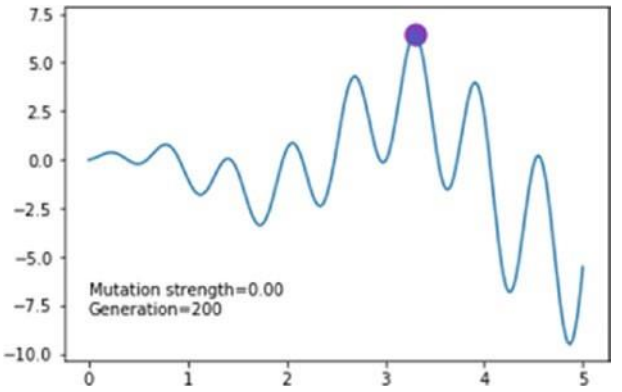

图10-6。 *第200 代图 - 多代*

找到的解决方案是最佳的！

提示 尝试提前停止以了解如何节省时钟周期并获得一个‘足够好’的结果。设置 mtype='B'并且 Mutation strength > 0.001。

它花了 76 代，只使用了 200 代的 38.000%。

接下来，我将展示同一问题的不同算法。该解决方案创建了一个称为交叉的概念。

这意味着新一代是幸存的父母的混合体，即交叉。

打开第 10 章目录中的代码：Chapter-10-06- Evolutionary-Algorithm-03.ipynb

结果：

第 1 代（图 10-7）。

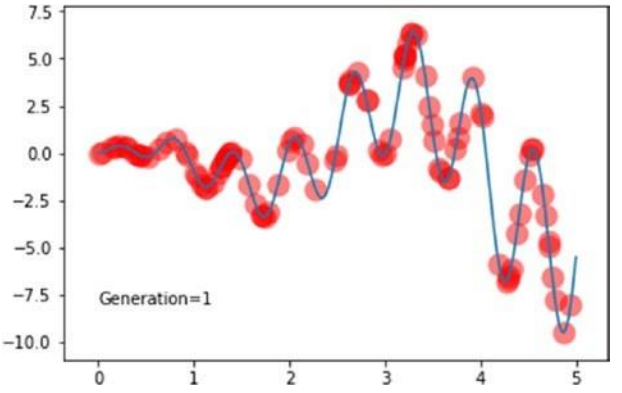

图10-7. 交叉的第1 代

在交叉期间会生成更多的下一代，然后只有适应值最高的下一代可以生存，然后他们才可以作为生成下一代的父母。参见图 10-8。

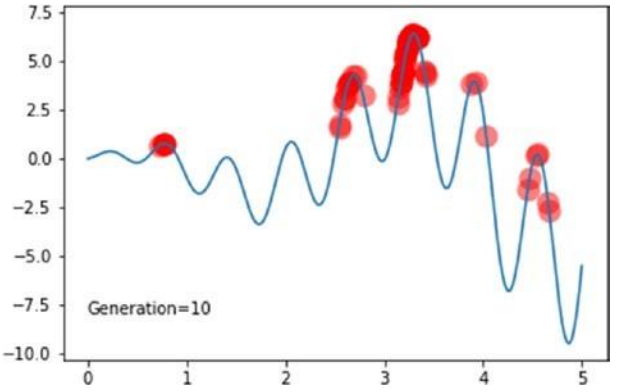

图10-8。 交叉第10 代

请注意，灭绝措施在交叉的过程中将高质量的个体传到下一代。

让我们看它什么时候收敛。参见图 10-9。

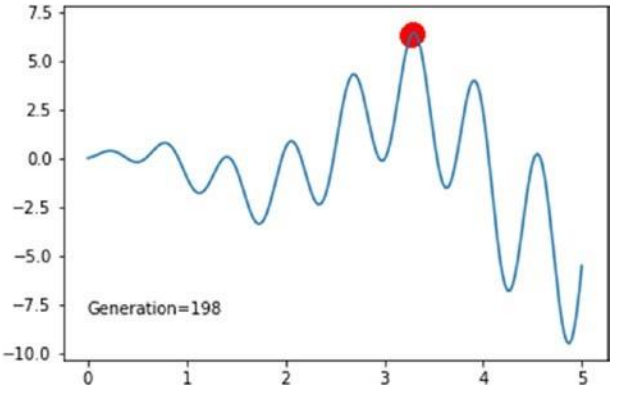

图10-9。 第198 代交叉结果

成功！

我建议你自己看下一个例子：

打开第 10 章目录中的示例：Chapter-10-07-Evolutionary-Algorithm-04.ipynb

基本要求是您的一代需要在两点之间移动。（找路径）

这是这些算法可以轻松解决的常见问题。

我建议你现在评估一下你的结果。

检查以下情况：

- 检查何时找到最佳路径。
- 检查 95%的个体在什么时候显著接近目标范围。

以下是我的发现（图 10-10）。

Gen: 0 | best fit: 0.023423059056103604

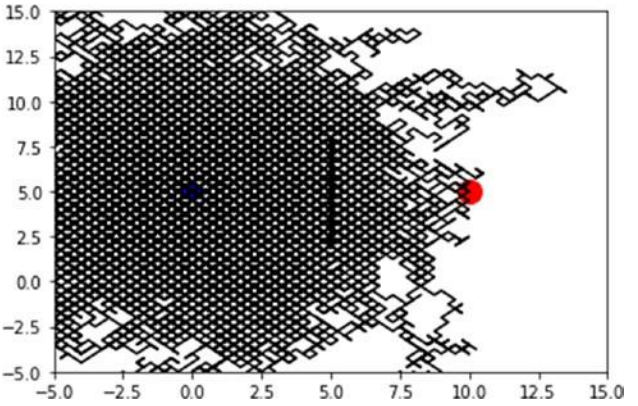

图 10-10. 第 0 代

在随机游走过程中，300 代人中只有 2 代到达了终点。这之成功了 0.667 % 。参见图 10-11。

Gen: 20 | best fit: 0.545819714368591

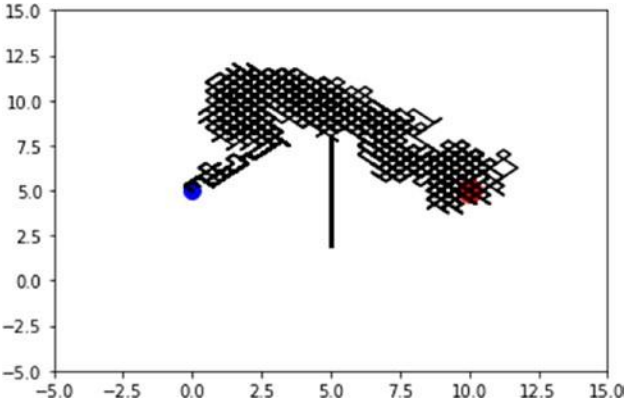

图10-11。 第20 代

到第 20 代，95%以上的代数（generation）最终位于端点上。

这很好， 但由于设计完成固定数量的循环， 我浪费了另外 279 个周期！

如果你看看最后一代 299（图 10-12）：

Gen: 299 | best fit: 0.545819714368591

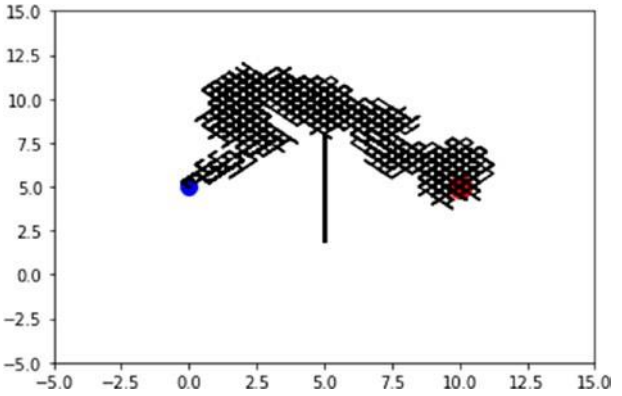

图10-12。 第299 代

第 20 代和第 299 代之间的增益为 0%。

警告 我们为什么总是研究进化计算中的提前退出策略，因为你的目标是一个足够好的解决方案，而不是一个最佳方案；如果你想得到一个最佳的结果，由于维度的诅咒，进化解决方案在高维空间下不会收敛到目标的 100%。你只是没有足够的数据，或者在这种情况下，没有一个足够大的种群以找到每一个可能的突变或交叉为 100%的解决方案。

打开第 10 章目录中的代码：Chapter-10-08- Evolutionary-Algorithm-05.ipynb 通过更改以下参数进行试验：

POP_SIZE_MIN = 30

POP_SIZE_MAX = 31

POP_SIZE_STEP = 1

N_GENERATIONS_MIN = 30

N_GENERATIONS_MAX = 31

N_GENERATIONS_STEP = 1

N_MOVES_MIN = 300

N_MOVES_MAX = 301

N_MOVES_STEP = 1

问题：

种群数量的影响是什么？

环境的最佳一代是哪一代？

移动的影响是什么？

线性规划

线性规划是采取与某些情况相关的线性不等式，并找到在这些条件下可获得的"最佳"值的过程。一个典型的例子是利用材料和劳动力的局限性，然后确定在这些条件下实现最大利润的"最佳"生产水平。它是数学的一个重要领域，称为"优化技术"。这个应用领域每天都在进行用于资源分配的研究。

考虑以下问题：

最小化：

f = -1*x[0] + 4*x[1]

Subject to:

-3*x[0] + 1*x[1] <= 6

1*x[0] + 2*x[1] <= 4

x[1] >= -3

where: -inf <= x[0] <= inf

打开第10 章目录中的代码：Chapter-10-09- Linear-programming-01.ipynb

```
c = [-1, 4]

A = [[-3, 1], [1, 2]]

b = [6, 4]

x0_bounds = (None, None)

x1_bounds = (-3, None)

res = opt.linprog(c, A_ub=A, b_ub=b, bounds=(x0_bounds, x1_bounds),
```

```
 options={"disp": True})
print(res)
```

Chapter 10 Evolutionary Computing

297

结果:

```
Optimization terminated successfully.
 Current function value: -22.000000
 Iterations: 1
 fun: -22.0
 message: 'Optimization terminated successfully.'
 nit: 1
 slack: array([39., 0.])
 status: 0
 success: True
 x: array([10., -3.])
```

线性规划显示，对于 X0=10 和 X1 = -3，该公式为最佳解决方案。

您是否注意到该问题很快就解决了？ 因为它只有一系列数学公式需要应用。这是线性规划的特征。

警告 限制是精确的或多方面的，线性规划的公式可能会变得极其复杂。

如果你需要任何公式优化要求，我建议你看看 SymPy (www.sympy.org)

打开第 10 章目录中的代码：Chapter-10-10- SymPy.ipynb

```
from sympy import *

solve(Eq(f, -1*x + 4*y),f) => [-x + 4*y]

solve(3*x + 1*y <= 6,x) => (-oo < x) & (x <= -y/3 + 2)

solve(3*x + 1*y <= 6,y) => (-oo < y) & (y <= -3*x + 6)

solve(1*x + 2*y <= 4,x) => (-oo < x) & (x <= -2*y + 4)

solve(1*x + 2*y <= 4,y) => (-oo < y) & (y <= -x/2 + 2)

solve(y >= -3) => (-3 <= y) & (y < oo)
```

这为您提供了确定 scipy opt.linprog 编程范围的快速方法。

线性规划领域在我的许多客户的系统中被广泛使用，因为对简单计算有许多要求，并且有助于创建机器学习。

恭喜你;您现在可以进行线性规划。

粒子群优化

粒子群优化 （PSO） 是一种计算方法，它通过迭代尝试改进有关度量值来优化问题。它通过拥有一组候选的解决方案（此处标记为粒子）并且根据粒子位置和速度的简单数学公式在搜索空间中移动这些粒子来解决问题。

每个粒子的移动受其局部已知位置的影响，但也被引导到搜索空间中最佳的位置，这些位置会随着其他粒子找到更好的位置而更新。这有望将种群推向最佳解决方案。

例子：

打开第 10 章目录中的代码：Chapter-10-11- Particle-Swarm-01.ipynb

结果：

```
Running PSO ...
Iteration 0: Best Cost = 132.48185745142655
Iteration 1: Best Cost = 132.48185745142655
Iteration 2: Best Cost = 132.48185745142655
```

Iteration 3: Best Cost = 132.48185745142655
Iteration 4: Best Cost = 132.48185745142655
Iteration 5: Best Cost = 132.48185745142655
<Removed part of results>
Iteration 46: Best Cost = 1.8639730250806885
Iteration 47: Best Cost = 1.8639730250806885
Iteration 48: Best Cost = 1.8471628852322097
Iteration 49: Best Cost = 1.845248848436588
Global Best:

{'position':

array([-0.11522734, -0.04638722, 0.41577969, 0.21634509, -0.4200067 ,

-0.84273085, -0.08707212, 0.83846401, -0.10769234, -0.03660312]),

'cost': 1.845248848436588}

这应该让您对 PSO 有一个基本的了解。

强化学习

你已经学习了一整章的强化学习 （RL)， 但我介绍这两个特定的算法作为进化计算的一部分，因为这两种算法更基于进化计算。

RL 算法一

打开第10 章目录中的代码：Chapter-10-12-RL-01.ipynb

结果：

Optimized Policy:

[0. 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 12. 11. 15. 16. 17.

18. 6. 20. 21. 3. 23. 24. 25. 1. 2. 3. 4. 5. 6. 7. 8. 9. 10.

11. 12. 38. 11. 10. 9. 42. 7. 44. 5. 46. 47. 48. 49. 50. 1. 2. 3.

4. 5. 6. 7. 8. 9. 10. 11. 12. 13. 11. 10. 9. 17. 7. 19. 5. 21.

22. 23. 24. 25. 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 12. 11.

10. 9. 8. 7. 6. 5. 4. 3. 2. 1.]

Optimized Value Function:

[0.00000000e+00 7.24792480e-05 2.89916992e-04 6.95257448e-04
1.16010383e-03 1.76906586e-03 2.78102979e-03 4.03504074e-03
4.66214120e-03 5.59997559e-03 7.08471239e-03 9.03964043e-03
1.11241192e-02 1.56793594e-02 1.61464431e-02 1.69517994e-02
1.86512806e-02 1.98249817e-02 2.24047303e-02 2.73845196e-02
2.83388495e-02 3.04937363e-02 3.61633897e-02 3.84953022e-02
4.44964767e-02 6.25000000e-02 6.27174377e-02 6.33700779e-02
6.45857723e-02 6.59966059e-02 6.78135343e-02 7.08430894e-02
7.46098323e-02 7.64884604e-02 7.93035477e-02 8.37541372e-02
8.96225423e-02 9.58723575e-02 1.09538078e-01 1.10939329e-01
1.13360151e-01 1.18457374e-01 1.21977661e-01 1.29716907e-01
1.44653559e-01 1.47520113e-01 1.53983246e-01 1.70990169e-01
1.77987434e-01 1.95990576e-01 2.50000000e-01 2.50217438e-01
2.50870078e-01 2.52085772e-01 2.53496606e-01 2.55313534e-01
2.58343089e-01 2.62109832e-01 2.63988460e-01 2.66803548e-01
2.71254137e-01 2.77122542e-01 2.83372357e-01 2.97038078e-01
2.98439329e-01 3.00860151e-01 3.05957374e-01 3.09477661e-01
3.17216907e-01 3.32153559e-01 3.35020113e-01 3.41483246e-01
3.58490169e-01 3.65487434e-01 3.83490576e-01 4.37500000e-01
4.38152558e-01 4.40122454e-01 4.43757317e-01 4.47991345e-01
4.53440603e-01 4.62529268e-01 4.73829497e-01 4.79468031e-01
4.87912680e-01 5.01265085e-01 5.18867627e-01 5.37617932e-01

5.78614419e-01 5.82817988e-01 5.90080452e-01 6.05372123e-01

6.15934510e-01 6.39150720e-01 6.83960814e-01 6.92560339e-01

7.11950883e-01 7.62970611e-01 7.83963162e-01 8.37972371e-01

0.00000000e+00]

第一个 RL 算法完成。

RL 算法二

这里是第二个 RL 算法。

打开第 10 章目录中的代码：Chapter-10-13-RL-02. ipynb

结果如图 10-13 所示。

	1	2	3	4	5
1	3.31	8.79	4.43	5.32	1.49
2	1.52	2.99	2.25	1.91	0.55
3	0.05	0.74	0.67	0.36	-0.4
4	-0.97	-0.44	-0.35	-0.59	-1.18
5	-1.86	-1.35	-1.23	-1.42	-1.98

	1	2	3	4	5
1	21.98	24.42	21.98	19.42	17.48
2	19.78	21.98	19.78	17.8	16.02
3	17.8	19.78	17.8	16.02	14.42
4	16.02	17.8	16.02	14.42	12.98
5	14.42	16.02	14.42	12.98	11.68

图 10-13. *第二个 RL 算法*

旅行推销员问题

我之前已经介绍了这个问题，现在我将向您展示如何用进化计算处理这个问题。

打开第 10 章目录中的代码：Chapter-10-14-TSP-01.ipynb

您的结果如图 10-14 所示。

Gen: 499 | best fit: 53661.69

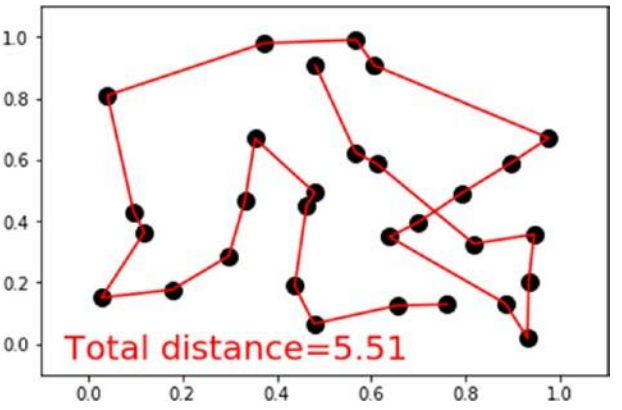

图10-14。 *旅行推销员*

示例 2：

与25个城市，一个蛮力的方法将不得不测试超过 ***3 百万的六次方个路径***！进化编程很有效，但许多因素效率更高。

打开第10 章目录中的代码：Chapter-10-15-TSP-02.ipynb

结果如图 10-15所示。

Initial distance: 2052.2221426003775
Final distance: 802.6666546711987

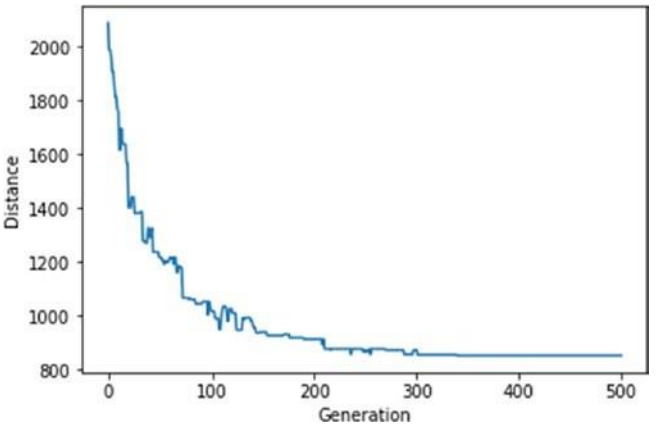

图10-15。 Tsp 为25 个城市

解决 Ny words

我之前已经注意到，进化计算善于找到具有高维数的数据集，因为它可以在发现整个系统时适应该过程。

我会带你在一个我写的句子上进行测试， 你的程序将解决这个问题。

打开第 10 章目录中的代码：Chapter-10-16- Words-01.ipynb

代码应生成以下句子：

'I love Denise very much!'

EP 在 14 个字母上比暴力破解快 240.8000 倍。

这显示了 EP 对文本字符串发现是很高效的。

我将向您展示下一个示例，以演示如何使用 EP 在地图上查找不同的路径。

科尼斯堡七桥

柯尼斯堡的七桥问题是数学历史上著名的问题。1736 年莱昂哈德·欧拉证明了这道题不可解，这奠定了图论的基础，并预示了拓扑学的理念。

普鲁士的科尼斯堡市（现在为俄罗斯的加里宁格勒）位于普雷格尔河两岸，包括两个大岛屿——克奈普霍夫岛和洛美岛——通过七座桥梁相互连接，或这说与该市的两个大陆部分相连。他的问题是设计一个步行方案穿过城市，并且跨越只每一个桥梁一次，只有一次。

通过明确地指定这个任务的逻辑规则，通过其中一座桥以外的岛屿或大陆沿岸访问任何桥梁而不跨越其另一端的解决方案显然不可接受。欧拉证明这个问题没有解决方案。

看你能否解决这个问题。

打开第 10 章目录中的代码：Chapter-10-17-Konigsberg-01.ipynb

```
import matplotlib
import numpy as np
matplotlib.use('TkAgg')
%matplotlib inline
import networkx as nx
import matplotlib.pyplot as plt
# Create the graph
graph = nx.MultiGraph()
graph.add_nodes_from([0, 3])
edges = [
    (0, 1, {'key': 'Grüne Brücke', 'color': 'blue', 'width': 5}),
    (0, 1, {'key': 'Köttelbrücke', 'color': 'red', 'width': 2}),
    (1, 2, {'key': 'Krämerbrücke', 'color': 'blue', 'width': 5}),
    (1, 2, {'key': 'Schmiedebrücke', 'color': 'red', 'width': 2}),
    (2, 3, {'key': 'Holzbrücke', 'color': 'blue', 'width': 5}),
    (3, 1, {'key': 'Dombrücke', 'color': 'blue', 'width': 5}),
    (3, 0, {'key': 'Hohe Brücke', 'color': 'blue', 'width': 5})
]
graph.add_edges_from(edges);
```

```
# Plot the graph
plt.figure()
labels = {}
for i, j, label in graph.edges(data='key'):
    if (i, j) not in labels:
        labels[(i, j)] = label
    else:
        labels[(i, j)] += '\n' + label
colors = [x for _, _, x in graph.edges(data='color')]
widths = [x for _, _, x in graph.edges(data='width')]
pos = nx.spring_layout(graph)
nx.draw_networkx(graph, pos, edge_color=colors, with_labels=True,
width=widths)
nx.draw_networkx_edge_labels(graph, pos, edge_labels=labels);
```

结果如图 10-16 所示 – 科尼斯堡的七座桥。

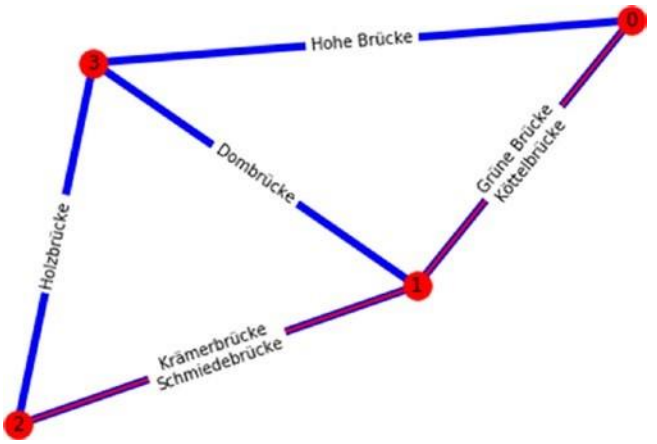

***图10-16。** 科尼斯堡七桥*

做得好，您已经模拟了这个问题，但这个问题没有事先给出的确定解决方案。

提示 不是所有的问题都有解决办法。 聪明的标志就是当你能够用最少的努力来确定这个事实，使你既有效又快速的解决方案。

接下来，我们将研究一个有解决方案的问题。

多仓库车辆调度问题

使用朴素贪婪算法进行多车辆路线模拟。

红色/绿色三角形：车辆

蓝色圆圈：尚未达到的目标

粉红色圆圈：已达到的目标

例子：

打开第 10 章目录中的代码：Chapter-10-18- MVRS-01.ipynb

卡车在给定点之间重定向，直到他们能够访问他们所有的结点。

结果如图 10-17 所示。

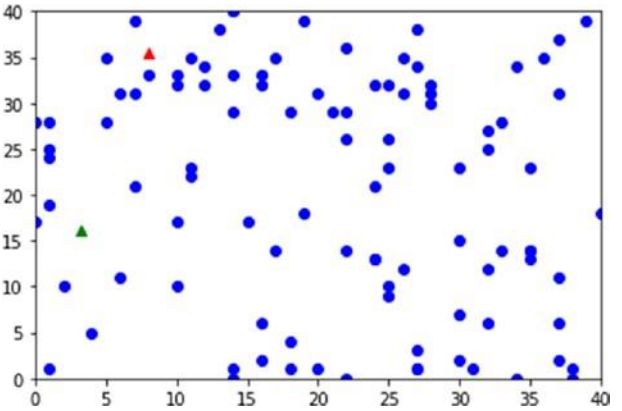

图10-17。 *多仓库车辆调度*

使用计划进行模拟

我将向您介绍一个 Python 库，它可以使您能够模拟规划问题的解决方案。

我建议你打开目录第 10 章的示例代码：Chapter-10-19- Basic-Scheduling-01.ipynb

pip install pyschedule

#加载 pyschedule 并且创建一个计划范围为十步的情景。

```
from pyschedule import Scenario, solvers, plotters

S = Scenario('dog_pyschedule',horizon=10)

# Create two resources

Angus, Jock = S.Resource('Angus'),
S.Resource('Jock')

# Create three tasks with lengths 1,2 and 3

mail, food, hole = S.Task('steal_mail',1),
S.Task('eat_food',2),

S.Task('dig_hole',3)

# Assign tasks to resources, either Angus or Jock,

# the %-operator connects tasks and resource

mail += Angus|Jock

food += Angus|Jock

hole += Angus|Jock

# Solve and print solution

S.use_makespan_objective()

solvers.mip.solve(S,msg=1)

# Print the solution

print(S.solution())
```

结果如图 10-18所示。

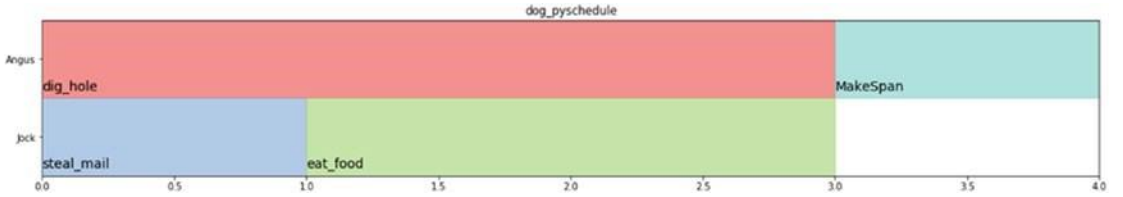

图10-18。 *乔克和安格斯在工作*

让我们来研究一个更复杂的模拟规划问题。

例子：

我建议你打开目录中的代码：Chapter-10- 20-Bicycle-Shop.ipynb

安德烈和劳伦斯使用 pyschedule 优化他们的自行车油漆店。

安德烈和劳伦斯正在经营一家自行车油漆店，在那里他们回收旧自行车并涂上新鲜颜色。今天，他们必须喷涂一辆绿色和红色的自行车。他们开始处理问题：他们导入 pyschedule 并创建新的方案。我们以小时作为粒度，预计工作日最多为 10 小时，因此将规划范围设置为 10 小时。某些方法不需要此参数，但默认的方案需要以下参数：

```
from pyschedule import Scenario, solvers, plotters S =
Scenario('Bicycle_Paint_Shop', horizon=10)
```

然后，他们创建自己作为资源：

```
Andre = S.Resource('Andre')

Laurence = S.Resource('Laurence')
```

喷一辆自行车需要两个小时。此外，自行车涂漆后，需要进行后期处理（例如，抽送轮胎），这需要一个小时（这是默认的）。这总共转换为四个任务：

```
Green_Paint, Red_Paint = S.Task('Green_Paint', length=2), S.Task('Red_
Paint', length=2)
Green_Prep, Red_Prep = S.Task('Green_Prep'), S.Task('Red_Prep')
```

显然，一个人只能在喷漆后做后期处理，两者之间有任意间隙。对于红色油漆，我们更严格一点；在这里，我们想在喷漆一小时之后开始后期处理，因为这是让颜色干燥的时间：

```
S += Green_Paint < Green_Prep, Red_Paint + 1 <= Red_Prep
```

每项任务都可以由安德烈或劳伦斯完成：

```
Green_Paint += Andre|Laurence

Green_Prep += Andre|Laurence

Red_Paint += Andre|Laurence

Red_Prep += Andre|Laurence
```

因此，让我们来看看这个场景：

```
S.clear_solution()

print(S)
```

我们还没有确定一个目标。我们希望尽早完成所有任务，因此我们使用MakeSpan并再次检查方案：

```
S.use_makespan_objective()
print(S)
```

因此，我们希望将 MakeSpan 任务的位置降至最低，但受所有其他任务的约束。因此，MakeSpan 的位置是我们时间表的长度。现在，我们已经有场景的第一个版本，让我们看一看结果：

为任务设置一些颜色：

```
task_colors = { Green_Paint : '#A1D372',
                Green_Prep : '#A1D372',
                Red_Paint : '#EB4845',
                Red_Prep : '#EB4845',
                S['MakeSpan'] : '#7EA7D8'}
```

一种用于求解和绘制图片的小方法：

```
def run(S) :

    if solvers.mip.solve(S):

        get_ipython().run_line_magic('matplotlib', 'inline')

        plotters.matplotlib.plot(S,task_colors=task_colors,fig_size=(15,2))
```

```
else:
    print('no solution exists')
run(S)
```

参见图 10-19 中的结果。

图 10-19. 场景（**1**）

请注意，有可能发生有人需要油漆红色自行车，然后做绿色后处理。这将很烦人；换自行车需要太多时间。我们使用以下约束来确保绿色/红色喷漆和后期处理始终由相同的人员完成：

```
#Green_Prep will use the same resources as Green_Paint if there is an
overlap in resource requirement
Green_Prep += Green_Paint*[Andre,Laurence]
# Same for Red_Prep and Red_Paint
Red_Prep += Red_Paint*[Andre,Laurence]
run(S)
```

结果如图 10-20所示。

图 10-20. 场景（**2**）

这个计划表在四个小时后完成，并建议同时绘制两辆自行车。然而，安德烈和劳伦斯只有一个油漆店，他们需要共享：

```
Paint_Shop = S.Resource('Paint_Shop')
Red_Paint += Paint_Shop
```

Green_Paint += Paint_Shop

run(S)

结果如图 10-21所示。

图 10-21. 场景（3）

很棒；五个小时后， 每个人都可以回家吃午餐！

不幸的是，安德烈接到一个电话，说红色自行车将在两个小时后到达：

S += Red_Paint > 2

run(S)

结果如图 10-22所示。

图 10-22. 场景（4）

太糟糕了；任务将在几小时后完成。因此，安德烈和劳伦斯决定安排在第三小时后和第五小时之前共进午餐：

Lunch = S.Task('Lunch')

Lunch += {Andre, Laurence}

S += Lunch > 3, Lunch < 5

task_colors[Lunch] = '#7EA7D8'

S.clear_objective() #we need to remove the objective and readd it because

of the new lunch task

S.use_makespan_objective()

task_colors[S['MakeSpan']] = '#7EA7D8'

run(S)

结果如图 10-23所示。

图 10-23. 场景（5）

安德烈是一个提早完成任务的人，想在午饭前完成三个小时的工作，也就是说，在第三个小时之前：

S += Andre['length'][0:3] >= 3

run(S)

结果如图 10-24 所示。

图 10-24. 场景（6）

天气预报说下午天气真的很好，所以安德烈和劳伦斯决定在午饭后关门，也就是说，他们把范围固定到 5 个小时。不幸的是，会发生以下情况：

S.horizon = 5

run(S)

结果：

> 不存在解决方案。安德烈和劳伦斯需要7个小时，他们把地平线固定到7小时。

S.horizon = 7

run(S)

结果如图 10-25所示。

图 10-25. **场景（7）**

做得好；你现在可以经营油漆店了

奖金：

我建议你看看两个奖金的例子。两者都有三个资源，这个代码可以说明如何规划更大的团队。

- Chapter-10-21-Basic-Scheduling-Three-Way.ipynb.
- Chapter-10-22-Bicycle-Shop-Three-way.ipynb.

提示 我已经成功地运行了多达 20 个资源和 100 多个任务。您也可以将该过程用于更大的流程。

你取得了什么成就？

做得好；现在，您可以创建一个需要进化计算的机器学习问题。

您可以处理小规模的调度问题。

接下来是什么？

您已成功完成进化计算章节。

现在，您可以使用模拟和进化技术来发现和解决环境中日常问题。

本书的其余部分将介绍一些您已经获得的知识的应用。

接下来，我将向您介绍第 11 章中机电一体化的有趣领域。