第 5 章

# 优化

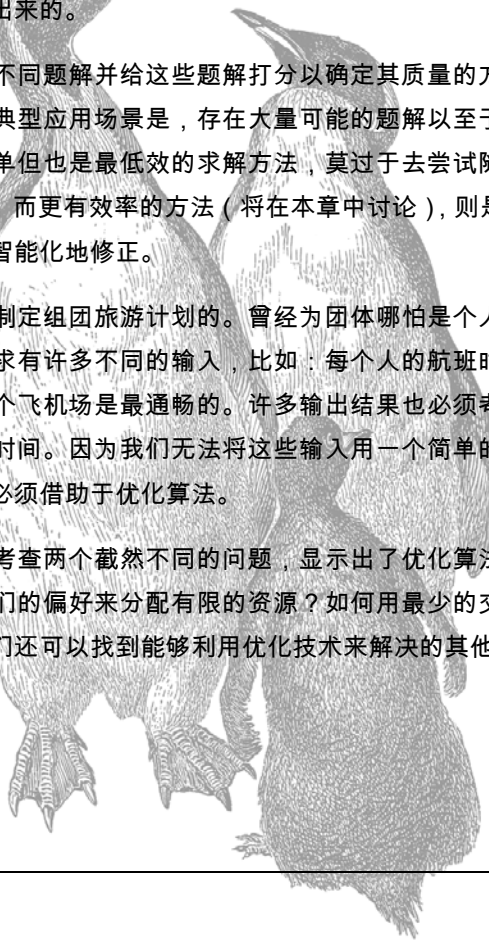
# Optimization

本章我们将向大家介绍，如何使用一系列被称为随机优化（stochastic optimization）的技术来解决协作类问题。优化技术特别擅长于处理：受多种变量的影响，存在许多可能解的问题，以及结果因这些变量的组合而产生很大变化的问题。这些优化技术有着大量的应用：在物理学中，我们用它们来研究分子的运动；在生物学中，我们用它们来预测蛋白质的结构；在计算机科学中，我们用它们来测定算法的最坏可能运行时间。NASA（美国国家航空和宇宙航行局）甚至使用优化技术来设计具有正确操作特性的天线，而这些天线看起来似乎不像是人类设计者创造出来的。

优化算法是通过尝试许多不同题解并给这些题解打分以确定其质量的方式来找到一个问题的最优解的。优化算法的典型应用场景是，存在大量可能的题解以至于我们无法对它们进行一一尝试的情况。最简单但也是最低效的求解方法，莫过于去尝试随机猜测的上千个题解，并从中找出最佳解来。而更有效率的方法（将在本章中讨论），则是以一种对题解可能有改进的方式来对其进行智能化地修正。

本章的第一个例子是关于制定组团旅游计划的。曾经为团体哪怕是个人安排过旅游计划的人都知道，计划的制定要求有许多不同的输入，比如：每个人的航班时间表应该是什么，须要租用多少辆汽车，哪个飞机场是最通畅的。许多输出结果也必须考虑，比如，总的成本、候机的时间、起飞的时间。因为我们无法将这些输入用一个简单的公式映射到输出，所以要想找到最优解，就必须借助于优化算法。

本章的其他两个例子通过考查两个截然不同的问题，显示出了优化算法的灵活性。这两个例子分别是：如何基于人们的偏好来分配有限的资源？如何用最少的交叉线来可视化社会网络。在本章的末尾，我们还可以找到能够利用优化技术来解决的其他类型的问题。



# 组团旅游

## Group Travel

为来自不同地方去往同一地点的人们(本例中是 Glass 一家)安排一次旅游是一件极富挑战性的事情，它引出了一个非常有趣的优化问题。首先，我们创建一个文件，命名为 *optimization.py*，然后加入下面的代码：

```
import time
import random
import math

people = [('Seymour','BOS'),
          ('Franny','DAL'),
          ('Zooey','CAK'),
          ('Walt','MIA'),
          ('Buddy','ORD'),
          ('Les','OMA')]

# New York 的 LaGuardia 机场
destination='LGA'
```

家庭成员们来自全国各地，并且他们希望在纽约会面。他们将在同一天到达，并在同一天离开，而且他们想搭乘相同的交通工具往返飞机场。每天有许多航班从任何一位家庭成员的所在地飞往纽约，飞机的起飞时间都是不同的。这些航班在价格和续航时间上也都不尽相同。

我们可以从这个网址*http://kiwitobes.com/optimize/schedule.txt*下载有关航班数据的样本文件。

该文件包含一组以逗号分隔的、具有如下数据格式的航班数据：起点、终点、起飞时间、到达时间、价格。

```
LGA,MIA,20:27,23:42,169
MIA,LGA,19:53,22:21,173
LGA,BOS,6:39,8:09,86
BOS,LGA,6:17,8:26,89
LGA,BOS,8:23,10:28,149
```

我们将这些数据载入到一个字典中，并以起止点为键，以可能的航班详情明细为值。请将如下这段加载数据的代码添加到 *optimization.py* 文件中：

```
flights={}
#
for line in file('schedule.txt'):
  origin,dest,depart,arrive,price=line.strip().split(',')
  flights.setdefault((origin,dest),[])

  # 将航班详情添加到航班列表中
  flights[(origin,dest)].append((depart,arrive,int(price)))
```

我们还在此处定义了一个非常有用的工具函数 `getminutes`，该函数用于计算某个给定时间在一天中的分钟数。这使飞行时间和候机时间的计算变得非常容易。现在我们将这个函数添加到 *optimization.py* 文件中：

```
def getminutes(t):
  x=time.strptime(t,'%H:%M')
  return x[3]*60+x[4]
```

接下来的问题是，家庭中的每个成员应该乘坐哪个航班呢？当然，使总的票价降下来是一个目标，但是最优解将要考虑和尝试最小化的还有许多其他可能的因素，比如：总的候机时间或总的飞行时间。稍候我们将详细讨论这些因素。

## 描述题解

### Representing Solutions

当处理类似这样的问题时，我们有必要明确潜在的题解将如何表达。后面我们将看到的优化函数是非常通用的，它能应用于许多不同类型的问题上，因此，选择一个题解的简单表示方法而不局限于组团旅游问题是非常重要的。有一种非常通用的表达方式，就是数字序列。在本例中，一个数字可以代表某人选择乘坐的航班——0 是这天中的第一次航班，1 是第二次，依次类推。因为每个人都须要往返两个航班，所以列表的长度是人数的两倍。

例如：

```
[1,4,3,2,7,3,6,3,2,4,5,3]
```

上述列表代表了一种题解：Seymour 搭乘当天的第 2 次航班从 Boston 飞往 New York，并搭乘当天的第 5 次航班返回到 Boston。Franny 搭乘第 4 次航班从 Dallas 飞往 New York，并搭乘第 3 次航班返回。

因为要从一列数字中解释清楚题解是很困难的，所以我们需要一个程序，能将人们决定搭乘的所有航班打印成表格。请将下列函数加入 *optimization.py* 中：

```
def printschedule(r):
  for d in range(len(r)/2):
    name=people[d][0]
    origin=people[d][1]
    out=flights[(origin,destination)][r[2*d]]
    ret=flights[(destination,origin)][r[2*d+1]]
    print '%10s%10s %5s-%5s $%3s %5s-%5s $%3s' % (name,origin,
                                                  out[0],out[1],out[2],
                                                  ret[0],ret[1],ret[2])
```

上述函数将打印出一行，包括：人名和起点、出发时间、到达时间，以及往返航班的票价。请在你的 Python 会话中尝试执行该函数。

```
>>> import optimization
>>> s=[1,4,3,2,7,3,6,3,2,4,5,3]
>>> optimization.printschedule(s)
    Seymour      BOS   12:34-15:02 $109  12:08-14:05 $142
     Franny      DAL   12:19-15:25 $342   9:49-13:51 $229
      Zooey      CAK    9:15-12:14 $247  15:50-18:45 $243
       Walt      MIA   15:34-18:11 $326  14:08-16:09 $232
      Buddy      ORD   14:22-16:32 $126  15:04-17:23 $189
        Les      OMA   15:03-16:42 $135   6:19- 8:13 $239
```

即使不考虑价格，上述安排仍然是有问题的。尤其是，因为家庭成员都一起往返于机场，因此以 Les 的返程航班为例，即使有些人直到下午 4 点才会飞离机场，每个人也都必须在早晨 6 点到达机场。为了确定最佳组合，程序需要一种方法来为不同日程安排的各种属性进行评估，从而决定哪一个方案是最好的。

## 成本函数

### The Cost Function

成本函数是用优化算法解决问题的关键，它通常是最难确定的。任何优化算法的目标，就是要寻找一组能够使成本函数的返回结果达到最小化的输入（在本例中，输入即为航班信息），因此成本函数须要返回一个值用以表示方案的好坏。对于好坏的程度并没有特定的衡量尺度，唯一的要求就是函数返回的值越大，表示该方案越差。

通常，根据众多变量来鉴别方案的好坏是比较困难的。我们来考查一些在组团旅游的例子中能够被度量的变量。

价格

所有航班的总票价，或者有可能是考虑财务因素之后的加权平均。

旅行时间

每个人在飞机上花费的总时间。

等待时间

在机场等待其他成员到达的时间。

出发时间

早晨太早起飞的航班也许会产生额外的成本，因为这要求旅行者减少睡眠的时间。

汽车租用时间

如果集体租用一辆汽车，那么他们必须在一天内早于起租时刻之前将车辆归还，否则将多付一天的租金。

为了让旅途体验能够更加愉快一些，对某一特定的时间安排从更多方面进行考查并非难事。每当我们为一个复杂问题寻求最佳方案时，都须要明确什么是最重要的因素。尽管这可能是有难度的，但是这样做的最大好处在于，一旦找到这些最重要的因素，只须做少量的修改，我们就可以运用本章中介绍的优化算法解决几乎任何一个问题。

选择好对成本产生影响的变量之后，我们就须要找到办法将它们组合在一起形成一个值。例如在本例中，我们就有必要明确，在飞机上的时间或在机场等待时所消耗的时间价值是多少。或许我们可以假定，在飞行旅行中节省的每一分钟价值 1 美元（这相当于，再加 90 美元选择乘坐直达航班，就可以节省一个半小时的时间），而在机场等待中所节省的每一分钟则价值 0.50 美元。如果每个人回到机场的时刻都晚于最初租用汽车的时刻，那么我们还可以将多付一天的租车费用也算在内。

此处定义的`schedulecost`函数有很多种可能的结果。该函数考查了总的旅行成本以及不同家庭成员在机场总的等待时间。如果汽车是在租用时间点之后归还的，则还会追加 50 美元的罚款。请将该函数添加到*optimization.py*文件中，并且你还可以随意追加额外的成本，或者调整金额和时间的相关重要性（译注1）：

译注 1：schedulecost 中对 latestarrival>earliestdep 的判断似乎有误，应改为 latestarrival<earliestdep。根据上下文，租用汽车的时间不应小于 latestarrival，而归还汽车的时间则不应大于 earliestdep。在 latestarrival>earliestdep 的情况下，一定会有租车时间>还车时间，那么此时不应罚款才对。只有当 latestarrival<earliestdep，且假定租车时间为 latestarrival，而还车时间为 earliestdep 时，我们才可以判定须要罚钱。

```
def schedulecost(sol):
  totalprice=0
  latestarrival=0
  earliestdep=24*60

  for d in range(len(sol)/2):
    # 得到往程航班和返程航班
    origin=people[d][1]
    outbound=flights[(origin,destination)][int(sol[2*d])]
    returnf=flights[(destination,origin)][int(sol[2*d+1])]

    # 总价格等于所有往程航班和返程航班价格之和
    totalprice+=outbound[2]
    totalprice+=returnf[2]

    # 记录最晚到达时间和最早离开时间
    if latestarrival<getminutes(outbound[1]): latestarrival=getminutes(outbound[1])
    if earliestdep>getminutes(returnf[0]): earliestdep=getminutes(returnf[0])

  # 每个人必须在机场等待直到最后一个人到达为止
  # 他们也必须在相同时间到达，并等候他们的返程航班
  totalwait=0
  for d in range(len(sol)/2):
    origin=people[d][1]
    outbound=flights[(origin,destination)][int(sol[2*d])]
    returnf=flights[(destination,origin)][int(sol[2*d+1])]
    totalwait+=latestarrival-getminutes(outbound[1])
    totalwait+=getminutes(returnf[0])-earliestdep
```

```
    # 这个题解要求多付一天的汽车租用费用吗？如果是，则费用为 50 美元
    if latestarrival>earliestdep: totalprice+=50

    return totalprice+totalwait
```

上述函数中的逻辑虽然非常简单，但是它却阐明了关键的因素。我们还可以采用若干方法对其功能进行增强——目前，总的等待时间的计算是假定：当家庭成员中的最后一人到达时所有人才会一起离开机场，并且所有人会一起赶往机场搭乘最早一班飞机离开。我们可以对此进行修改，允许任何要面临两小时或更长时间等待的人可以独自租车离开，我们还可以对其中的价格和等待时间做出相应的调整。

我们可以在 Python 会话中尝试运行一下该函数：

```
>>> reload(optimization)
>>> optimization.schedulecost(s)
5285
```

成本函数既已建立，那么应该很清楚，我们的目标就是要通过选择正确的数字序列来最小化该成本。理论上，我们可以尝试每种可能的组合，但在这个例子中，一共有 12 个航班，每种航班又有 10 种可能，因此会得到 $10^{12}$（大约 1000 亿）种组合。测试每种组合能确保我们得到最优的答案，但是在大多数计算机上，这会花费非常长的时间。

## 随机搜索

### Random Searching

随机搜索不是一种非常好的优化算法，但是它却使我们很容易领会所有算法的真正意图，并且它也是我们评估其他算法优劣的基线（baseline）。

这个函数接受两个参数。Domain 是一个由二元组（2-tuples）构成的列表，它指定了每个变量的最小最大值。题解的长度与此列表的长度相同。在当前的例子中，每个人都有 10 个往程航班和 10 个返程航班，因此列表中的 domain 是(0,9)，每个人重复两次。

第二个参数，`costf`，是成本函数，本例中即是 `schedulecost`。将其作为参数传入是为了让这个函数能够为其他优化问题所重用。此函数会随机产生 1 000 次猜测，并对每一次猜测调用 `costf`。它会跟踪最佳猜测（即具有最低成本的题解）并将结果返回。请将该函数加入 *optimization.py*：

```
def randomoptimize(domain,costf):
  best=999999999
  bestr=None
  for i in range(1000):
    # 创建一个随机解
    r=[random.randint(domain[i][0],domain[i][1])
       for i in range(len(domain))]
```

```
    # 得到成本
    cost=costf(r)

    # 与到目前为止的最优解进行比较
    if cost<best:
      best=cost
      bestr=r
  return r
```

当然，1 000 次猜测在全部可能性中仅占非常小的一部分。然而在本例中，我们可能会从中找到不少表现尚可的题解（即便不是最优的），因此在尝试一千次之后，该函数有可能会找到一个看似不算很差的解。请在你的 Python 会话中尝试执行一下该函数：

```
>>> reload(optimization)
>>> domain=[(0,9)]*(len(optimization.people)*2)
>>> s=optimization.randomoptimize(domain,optimization.schedulecost)
>>> optimization.schedulecost(s)
3328
>>> optimization.printschedule(s)
    Seymour      BOS    12:34-15:02  $109  12:08-14:05  $142
     Franny      DAL    12:19-15:25  $342   9:49-13:51  $229
      Zooey      CAK     9:15-12:14  $247  15:50-18:45  $243
       Walt      MIA    15:34-18:11  $326  14:08-16:09  $232
      Buddy      ORD    14:22-16:32  $126  15:04-17:23  $189
        Les      OMA    15:03-16:42  $135   6:19- 8:13  $239
```

由于是随机的原因，你所得到的结果将会与此处给出的结果有所不同。上述结果不算很好，因为 Zooey 须要在机场等候 6 个小时直至 Walt 到达，但是这样的结果显然不算是最差的。多执行几次该函数，看看成本值是否变化很大，或者把循环次数增加到 10 000，看看是否能按此方向找到更优的结果。

## 爬山法

### Hill Climbing

随机尝试各种题解是非常低效的，因为这种方法没有充分利用已经发现的优解。在我们的例子中，拥有较低总成本的时间安排很可能接近于其他低成本安排。因为随机优化是到处跳跃的（jumps around），所以它不会自动去寻找与已经被发现的优解相接近的题解。

随机搜索的一个替代方法叫做爬山法。爬山法以一个随机解开始，然后在其临近的解集中寻找更好的题解（具有更低的成本）。这类似于从斜坡上向下走，如图 5-1 所示。

想象一下你就是图中所示的那个人，不经意间陷入了这块区域中，并且想走到最低点去寻

找水源。为此我们可以选择任何一个方向，然后朝着最为险峻的斜坡向下走去。你可以朝

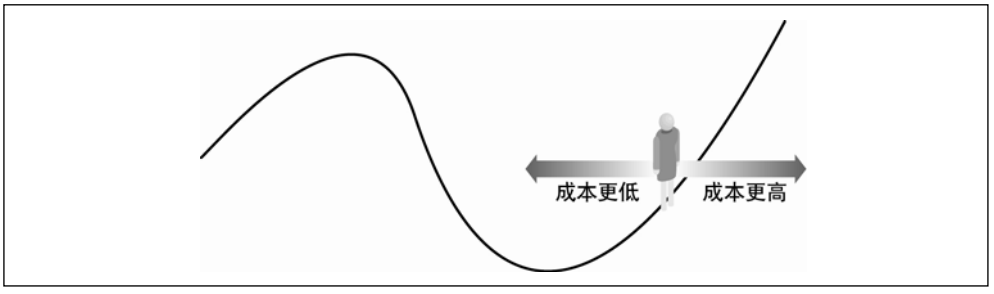



图 5-1：用爬山法寻找最低成本

着最为险峻的斜坡方向一直走下去，直至到达地势平坦或坡度开始向上倾斜的区域。

我们可以应用这种爬山法为 Glass 一家找到最好的旅行安排方案。先从一个随机的时间安排开始，然后再找到所有与之相邻的安排。在本例中，亦即找到所有相对于最初的随机安排，能够让某个人乘坐的航班稍早或者稍晚一些的安排。我们对每一个相邻的时间安排都进行成本计算，具有最低成本的安排将成为新的题解。重复这一过程直到没有相邻安排能够改善成本为止。

为了实现这一功能，请将 `hillclimb` 添加到 *optimization.py* 文件中：

```
def hillclimb(domain,costf):
  # 创建一个随机解
  sol=[random.randint(domain[i][0],domain[i][1])
     for i in range(len(domain))]

  # 主循环
  while 1:

    # 创建相邻解的列表
    neighbors=[]
    for j in range(len(domain)):

      # 在每个方向上相对于原值偏离一点
      if sol[j]>domain[j][0]:

        neighbors.append(sol[0:j]+[sol[j]-1]+sol[j+1:])
      if sol[j]<domain[j][1]:
        neighbors.append(sol[0:j]+[sol[j]+1]+sol[j+1:])

    # 在相邻解中寻找最优解
    current=costf(sol)
    best=current
    for j in range(len(neighbors)):
      cost=costf(neighbors[j])
      if cost<best:
        best=cost
        sol=neighbors[j]
```

```
    # 如果没有更好的解，则退出循环
    if best==current:
      break

  return sol
```

该函数在给定域内随机生成一个数字列表，用以构造初始的题解。它通过循环遍历列表中的每一个元素，找到当前解的所有相邻题解，然后创建出两个新的列表：一个列表中的元素加 1，另一个列表中的元素减 1。相邻解中最优的一个将成为新的当前题解。

请在你的 Python 会话中尝试执行该函数，看看与随机搜索方法相比，其效果如何：

```
>>> s=optimization.hillclimb(domain,optimization.schedulecost)
>>> optimization.schedulecost(s)
3063
>>> optimization.printschedule(s)
    Seymour      BOS 12:34-15:02 $109 10:33-12:03 $ 74
     Franny      DAL 10:30-14:57 $290 10:51-14:16 $256
      Zooey      CAK 10:53-13:36 $189 10:32-13:16 $139
       Walt      MIA 11:28-14:40 $248 12:37-15:05 $170
      Buddy      ORD 12:44-14:17 $134 10:33-13:11 $132
        Les      OMA 11:08-13:07 $175 18:25-20:34 $205
```

该函数的运行速度很快，并且找到的解通常要比随机搜索方法找到的更好。然而，爬山法有一个较大的缺陷。如图 5-2 所示：

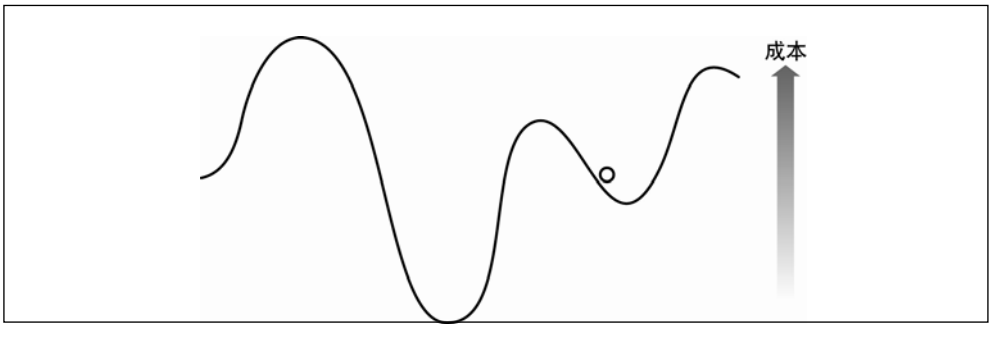



图 5-2：陷入局部范围内的最小值

从上图中我们可以很明显地看出，简单地从斜坡滑下不一定会产生全局最优解。最后的解会是一个局部范围内的最小值，它比邻近解的表现都好，但却不是全局最优的。全局最优解就是全局最小值，它是优化算法最终应该找到的那个解。解决这一难题的一种方法被称为随机重复爬山法（random-restart hill climbing），即让爬山法以多个随机生成的初始解为起点运行若干次，借此希望其中有一个解能够逼近全局的最小值。在后面两节中，我们将向大家展示避免陷入局部最小值的其他方法，它们分别是：模拟退火算法和遗传算法。

# 模拟退火算法

## Simulated Annealing

模拟退火算法是受物理学领域启发而来的一种优化算法。退火是指将合金加热后再慢慢冷却的过程。大量的原子因为受到激发而向周围跳跃，然后又逐渐稳定到一个低能阶的状态，所以这些原子能够找到一个低能阶的配置（configuration）。

退火算法以一个问题的随机解开始。它用一个变量来表示温度，这一温度开始时非常高，尔后逐渐变低。每一次迭代期间，算法会随机选中题解中的某个数字，然后朝某个方向变化。在我们的例子中，Seymour 的返程航班也许会从当天的第二趟移到第三趟。其成本会在这一变化前后分别计算出来，并进行比较。

算法最为关键的部分在于：如果新的成本值更低，则新的题解就会成为当前题解，这和爬山法非常相似。不过，如果成本值更高的话，则新的题解仍将可能成为当前题解。这是避免图 5-2 中局部最小值问题的一种尝试。

某些情况下，在我们能够得到一个更优的解之前转向一个更差的解是很有必要的。模拟退火算法之所以管用，不仅因为它总是会接受一个更优的解，而且还因为它在退火过程的开始阶段会接受表现较差的解。随着退火过程的不断进行，算法越来越不可能接受较差的解，直到最后，它将只会接受更优的解。更高成本的题解，其被接受的概率由下列公式给出：

$$p=e^{(-(highcost-lowcost)/temperature)}$$

因为温度（接受较差解的意愿）开始非常之高，指数将总是接近于 0，所以概率几乎为 1。随着温度的递减，高成本值和低成本值之间的差异越来越重要——差异越大，概率越低，因此该算法只倾向于稍差的解而不会是非常差的解。

请在 *optimization.py* 文件中创建一个名为 `annealingoptimize` 的新函数，实现上述算法：

```
def annealingoptimize(domain,costf,T=10000.0,cool=0.95,step=1):
  # 随机初始化值
  vec=[float(random.randint(domain[i][0],domain[i][1]))
       for i in range(len(domain))]

  while T>0.1:
    # 选择一个索引值
    i=random.randint(0,len(domain)-1)

    # 选择一个改变索引值的方向
    dir=random.randint(-step,step)
```

```
    # 创建一个代表题解的新列表，改变其中一个值
    vecb=vec[:]
    vecb[i]+=dir
    if vecb[i]<domain[i][0]: vecb[i]=domain[i][0]
    elif vecb[i]>domain[i][1]: vecb[i]=domain[i][1]

    # 计算当前成本和新的成本
    ea=costf(vec)
    eb=costf(vecb)

    # 它是更好的解吗？或者是趋向最优解的可能的临界解吗？
    if (eb<ea or random.random()<pow(math.e,-(eb-ea)/T)):
      vec=vecb

    # 降低温度
    T=T*cool
  return vec
```

为了退火，函数首先创建一个具有合适长度的随机解，其中的所有值都位于定义域参数指定的范围内。温度和冷却率是两个可选的参数。函数会在每次迭代时，将 `i` 设为题解的一个随机索引，并将 `dir` 设为介于`-step` 与 `step` 之间的某个随机数。该算法会计算当前函数的成本，以及以 `dir` 为增量对 `i` 处的值进行修改时的函数成本。

粗体显示的代码行是关于概率计算的，概率的值随着 `T` 的降低而降低。如果介于 0 和 1 之间的某个随机浮点数小于该值，或者如果新的题解更优，那么该函数将接受新的题解。函数中的循环过程直到温度几乎等于 0 为止，每次循环会将温度值与冷却率相乘。

现在，我们可以在自己的 Python 会话中尝试使用模拟退火算法来进行优化了：

```
>>> reload(optimization)
>>> s=optimization.annealingoptimize(domain,optimization.schedulecost)
>>> optimization.schedulecost(s)
2278
>>> optimization.printschedule(s)
    Seymour      BOS 12:34-15:02 $109 10:33-12:03 $ 74
     Franny      DAL 10:30-14:57 $290 10:51-14:16 $256
      Zooey      CAK 10:53-13:36 $189 10:32-13:16 $139
       Walt      MIA 11:28-14:40 $248 12:37-15:05 $170
      Buddy      ORD 12:44-14:17 $134 10:33-13:11 $132
        Les      OMA 11:08-13:07 $175 15:07-17:21 $129
```

这种优化算法在保持成本持续下降的同时，在减少总的执行时间方面也表现不俗。很显然，你得到的结果可能会有所不同，甚至有可能碰巧会得到一个较差的结果。对于任何一个给定的问题，不妨使用不同的参数（初始温度和冷却率）做一做试验。你还可以改变代表随机推进的 `step` 值的大小。

# 遗传算法

## Genetic Algorithms

另一类优化技术也是受自然科学的启发，被称为遗传算法。这类算法的运行过程是先随机生成一组解，我们称之为种群（population）。在优化过程中的每一步，算法会计算整个种群的成本函数，从而得到一个有关题解的有序列表。示例如表 5-1 所示。

表 5-1：题解及成本的有序列表

| 题解 | 成本 |
|---|---|
| [7, 5, 2, 3, 1, 6, 1, 6, 7, 1, 0, 3] | 4394 |
| [7, 2, 2, 2, 3, 3, 2, 3, 5, 2, 0, 8] | 4661 |
| … | … |
| [0, 4, 0, 3, 8, 8, 4, 4, 8, 5, 6, 1] | 7845 |
| [5, 8, 0, 2, 8, 8, 8, 2, 1, 6, 6, 8] | 8088 |

在对题解进行排序之后，一个新的种群——我们称之为下一代——被创建出来了。首先，我们将当前种群中位于最顶端的题解加入其所在的新种群中。我们称这一过程为精英选拔法（elitism）。新种群中的余下部分是由修改最优解后形成的全新解所组成的。

有两种修改题解的方法。其中较为简单的一种被称为变异（mutation），其通常的做法是对一个既有解进行微小的、简单的、随机的改变。在本例中，要完成变异只须从题解中选择一个数字，然后对其进行递增或递减即可。图 5-3 中给出了两个示例。

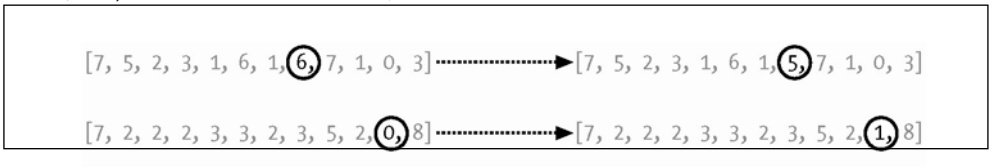



图 5-3：针对题解的变异示例

修改题解的另一种方法称之为交叉（crossover）或配对（breeding）。这种方法是选取最优解中的两个解，然后将它们按某种方式进行结合。在本例中，要实现交叉的一种简单方式是，从一个解中随机取出一个数字作为新题解中的某个元素，而剩余元素则来自另一个题解，如图 5-4 所示。

一个新的种群是通过对最优解进行随机的变异和配对处理构造出来的，它的大小通常与旧的种群相同。尔后，这一过程会一直重复进行——新的种群经过排序，又一个种群被构造出来。达到指定的迭代次数，或者连续经过数代后题解都没有得到改善，整个过程就结束了。

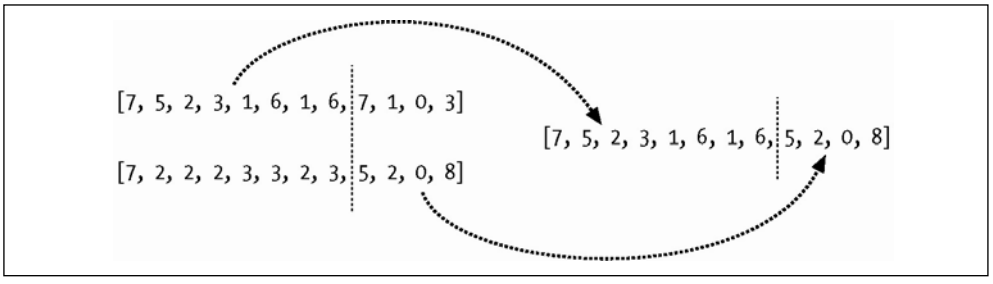



图 5-4：交叉示例

请将 `geneticoptimize` 添加到 *optimization.py* 文件中：

```
def geneticoptimize(domain,costf,popsize=50,step=1,
                    mutprob=0.2,elite=0.2,maxiter=100):
  # 变异操作
  def mutate(vec):
    i=random.randint(0,len(domain)-1)
    if random.random()<0.5 and vec[i]>domain[i][0]:
      return vec[0:i]+[vec[i]-step]+vec[i+1:]
    elif vec[i]<domain[i][1]:
      return vec[0:i]+[vec[i]+step]+vec[i+1:]

  # 交叉操作
  def crossover(r1,r2):
    i=random.randint(1,len(domain)-2)
    return r1[0:i]+r2[i:]

  # 构造初始种群
  pop=[]
  for i in range(popsize):
    vec=[random.randint(domain[i][0],domain[i][1])
        for i in range(len(domain))]
    pop.append(vec)

  # 每一代中有多少胜出者？
  topelite=int(elite*popsize)

  # 主循环
  for i in range(maxiter):
    scores=[(costf(v),v) for v in pop]
    scores.sort()
    ranked=[v for (s,v) in scores]

    # 从纯粹的胜出者开始
    pop=ranked[0:topelite]

    # 添加变异和配对后的胜出者
    while len(pop)<popsize:
      if random.random()<mutprob:
```

```
          # 变异
          c=random.randint(0,topelite)
          pop.append(mutate(ranked[c]))
        else:

          # 交叉
          c1=random.randint(0,topelite)
          c2=random.randint(0,topelite)
          pop.append(crossover(ranked[c1],ranked[c2]))

      # 打印当前最优值
      print scores[0][0]

    return scores[0][1]
```

上述函数引入了几个可选的参数：

`popsize`

种群大小

`mutprob`

种群的新成员是由变异而非交叉得来的概率。

`elite`

种群中被认为是优解且被允许传入下一代的部分。

`maxiter`

须运行多少代。

请尝试在你的 Python 会话中，运用遗传算法优化一下旅行计划：

```
>>> s=optimization.geneticoptimize(domain,optimization.schedulecost)
3532
3503
...
2591
2591
2591
>>> optimization.printschedule(s)
    Seymour      BOS 12:34-15:02 $109 10:33-12:03 $ 74
     Franny      DAL 10:30-14:57 $290 10:51-14:16 $256
      Zooey      CAK 10:53-13:36 $189 10:32-13:16 $139
       Walt      MIA 11:28-14:40 $248 12:37-15:05 $170
      Buddy      ORD 12:44-14:17 $134 10:33-13:11 $132
        Les      OMA 11:08-13:07 $175 11:07-13:24 $171
```

在第 11 章中，我们还将看到遗传算法的一个拓展，称为遗传编程（genetic programming），

在那里，我们采用了类似的思路来完整构造新的程序。

提示：计算机科学家 John Holland 因其在 1975 年所撰写的《自然与人造系统的适应性（Adaptation in Natural and Artificial Systems）》一书（密歇根大学出版社），而被公认为是遗传算法之父。但是相关的研究工作还可以追溯到 20 世纪 50 年代，那时的生物学家们已经开始尝试在计算机上进行进化建模了。从那以后，遗传算法和其他优化方法已经被广泛应用于许多不同的问题。

- 寻找能够给出最佳音效的音乐厅外形。
- 为超音速飞机设计最佳的机翼。
- 给出最佳的化学制品库以供研发前沿药物参考之用。
- 自动化设计语音识别芯片。

我们可以将这些问题的潜在解转化成数字列表。这样我们就可以很容易地应用遗传算法或模拟退火算法了。

一种优化方法是否管用很大程度上取决于问题本身。模拟退火算法、遗传算法，以及大多数其他优化方法都有赖于这样一个事实：对于大多数问题而言，最优解应该接近于其他的优解。来看一个优化可能不起作用的例子，如图 5-5 所示。

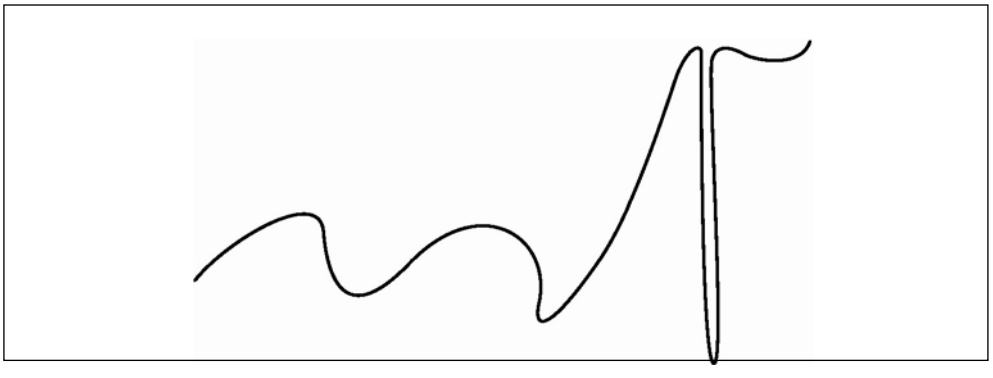


图 5-5：很难优化的问题

在图的最右边，成本的最低点实际上处在一个非常陡峭的区域。接近它的任何解都有可能被排除在外，因为这些解的成本都很高，所以我们永远都找不到通往全局最小值的途径。大多数算法都会陷入图中左边某个局部最小化的区域里。

优化算法对航班安排的例子之所以管用是因为，我们将一个人从当天的第二次航班转移到第三次航班，要比将他转移到第八次航班更有可能降低总成本。如果航班处于无序状态，

那么优化方法的效果是不会比随机搜索好多少的——事实上，在这种情况下，没有任何一种优化方法一定会比随机搜索更加有效。

# 真实的航班搜索

## Real Flight Searches

既然对于示例数据而言一切都没有问题了，那么是我们尝试利用真实的航班数据对前述优化算法的有效性进行考查的时候了。为此我们可以从 Kayak 网站下载数据，Kayak 提供了一套用于航班搜索的 API。真实的航班数据与你正在使用的示例数据的主要区别在于，在真实数据中，各大城市之间每天的航班远超过 9 次。

## Kayak API

如图 5-6 所示，Kayak是一个很受欢迎的旅游类垂直搜索引擎。尽管有许多在线的旅游站点，但是Kayak对本例而言还是很有价值的，因为它有一套非常不错的XML API，我们可以利用这套API在Python程序中执行真实的旅游搜索。为了使用这套API，我们必须去*http://www.kayak.com/labs/api/search*注册一个开发者密钥（developer key）。

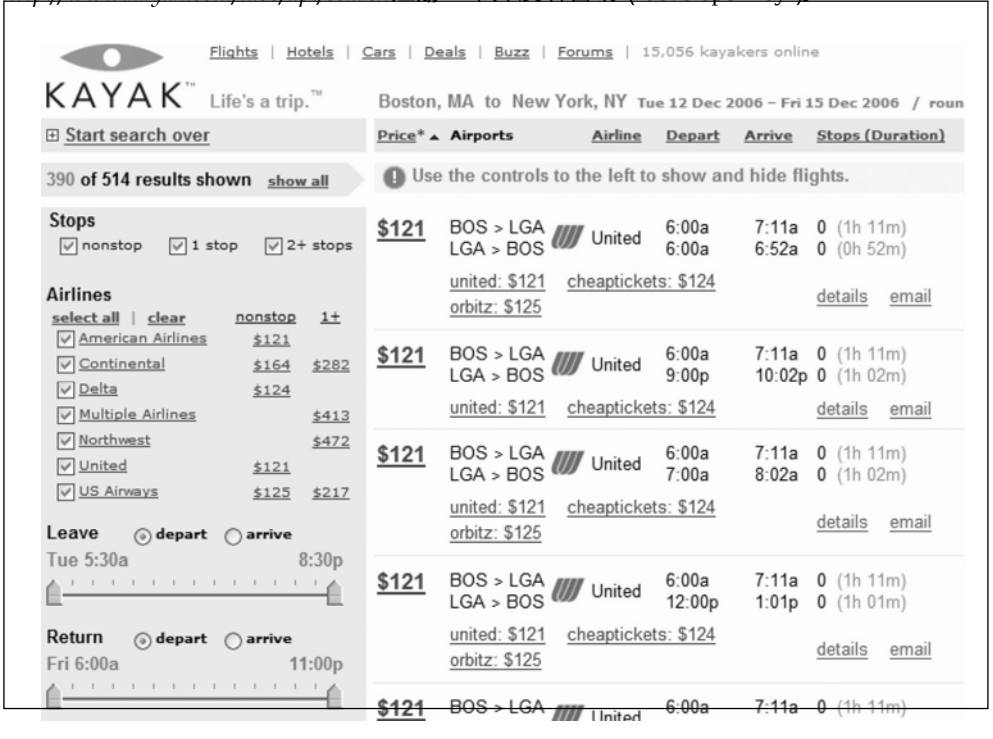



图 5-6：Kayak 旅游搜索界面的截屏图

开发者密钥是一个由数字与字母组合而成的长串，利用它我们可以在 Kayak 中进行航班搜

索（我们也可以用它来搜索旅馆，不过这不是本章要讨论的话题）。在本书撰写期间，还没

有像 del.icio.us 那样的专用于 Kayak 的 Python API，不过 Kayak 的 XML 接口是很好理解的。本章将为你示范怎样用 Python 的 *urllib2* 包和 *xml.dom.minidom* 包来建立搜索，这两个包都位于标准的 Python 发布包中。

## minidom 包
## The minidom Package

*minidom* 是标准 Python 发布包的一部分。它是文档对象模型（DOM）接口的一个轻量级实现，DOM 是一种将 XML 文档当作对象树来看待的标准方式。这个包接受字符串或包含 XML 的开放文件作为输入，然后返回一个对象，我们可以利用该对象轻松地提取信息。例如，我们可以在 Python 会话中输入下面的代码：

```
>>> import xml.dom.minidom
>>> dom=xml.dom.minidom.parseString('<data><rec>Hello!</rec></data>')
>>> dom
<xml.dom.minidom.Document instance at 0x00980C38>
>>> r=dom.getElementsByTagName('rec')
>>> r
[<DOM Element: rec at 0xa42350>]
>>> r[0].firstChild
<DOM Text node "Hello!">
>>> r[0].firstChild.data
u'Hello!'
```

由于许多 Web 站点现在都提供了利用 XML 接口来访问信息的方式，所以学习怎样使用 Python 的 XML 包对于集体智慧编程（collective intelligence programming）而言是非常有用的。下面这些是我们将要在 Kayak API 中用到的操作 DOM 对象的重要方法。

`getElementsByTagName(name)`

在整个文档范围内搜索标签名与 `name` 相匹配的元素，然后返回一个包含所有满足条件的 DOM 节点的列表。

`firstChild`

返回对象的首个子节点。在上面的例子中，`r` 的首个子节点就是代表文本“Hello”的节点

`data`

返回与对象有关的数据，大多数情况下这些数据就是该节点所内含的一个 Unicode 文本串。

## 航班搜索

### Flight Searches

首先请新建一个名为 *kayak.py* 的文件，然后加入下面的代码：

```
import time
import urllib2
import xml.dom.minidom

kayakkey='YOURKEYHERE'
```

我们要做的第一件事情是通过编写代码利用开发者密钥来获得一个新的 Kayak 会话。实现此功能的函数会向 apisession 发送一个带有 token 参数（设有你的开发者密钥）的请求。由该 URL 所返回的 XML 中会包含一个 sid 标签，内有 session ID：

```
<sid>1-hX4lII_wS$8b06aO7kHj</sid>
```

下面的函数只须对 XML 进行解析，以得到 sid 标签的内容。请将该函数加入 *kayak.py* 文件中：

```
def getkayaksession():
  # 构造 URL 以开启一个会话
  url='http://www.kayak.com/k/ident/apisession?token=%s&version=1' % kayakkey

  # 解析返回的 XML
  doc=xml.dom.minidom.parseString(urllib2.urlopen(url).read())

  # 找到<sid>xxxxxxxx</sid>标签
  sid=doc.getElementsByTagName('sid')[0].firstChild.data
  return sid
```

下一步是新建一个函数，开始进行航班搜索。用于该搜索的 URL 会很长，因为它包含了供航班搜索之用的所有参数。这其中最为重要的参数包括 sid（getkayaksession 函数返回的会话 ID）、destination，以及 depart_date。

返回的 XML 有个名为 searchid 的标签，函数将采用与 getkayaksession 同样的方式来提取 XML 中的内容。因为搜索也许会花费很长的时间，所以该调用最后实际上不会返回任何结果——它仅仅是启动搜索，然后返回一个可以用来获得结果的 ID。

请将该函数加入 *kayak.py* 文件中：

```
def flightsearch(sid,origin,destination,depart_date):

  # 构造搜索用的 URL
  url='http://www.kayak.com/s/apisearch?basicmode=true&oneway=y&origin=%s' % origin
  url+='&destination=%s&depart_date=%s' % (destination,depart_date)
  url+='&return_date=none&depart_time=a&return_time=a'
  url+='&travelers=1&cabin=e&action=doFlights&apimode=1'
  url+='&_sid_=%s&version=1' % (sid)

  # 得到 XML
  doc=xml.dom.minidom.parseString(urllib2.urlopen(url).read())

  # 提取搜索用的 ID
  searchid=doc.getElementsByTagName('searchid')[0].firstChild.data

  return searchid
```

最后，我们还需要一个函数来不断地请求结果，直到没有任何新结果获得为止。Kayak 提供了另外一个 URL——flight，它会给出航班的查询结果。在返回的 XML 中有一个 morepending 标签，直到搜索过程完成为止，该标签中始终会包含一个“true”的字样。这个函数须要一直请求页面到 morepending 不再为真（true）为止，这样函数才能够得到完整的结果。

请将该函数加入 *kayak.py* 文件中：

```
def flightsearchresults(sid,searchid):

  # 删除开头的$和逗号，并把数字转化成浮点类型
  def parseprice(p):
    return float(p[1:].replace(',',''))

  # 遍历检测
  while 1:
    time.sleep(2)

    # 构造检测所用的 URL
    url='http://www.kayak.com/s/basic/flight?'
    url+='searchid=%s&c=5&apimode=1&_sid_=%s&version=1' % (searchid,sid)
    doc=xml.dom.minidom.parseString(urllib2.urlopen(url).read())

    # 寻找 morepending 标签，并等待其不再为 true
    morepending=doc.getElementsByTagName('morepending')[0].firstChild
    if morepending==None or morepending.data=='false': break

  # 现在，下载完整的列表
  url='http://www.kayak.com/s/basic/flight?'
  url+='searchid=%s&c=999&apimode=1&_sid_=%s&version=1' % (searchid,sid)
  doc=xml.dom.minidom.parseString(urllib2.urlopen(url).read())

  # 得到不同元素组成的列表
  prices=doc.getElementsByTagName('price')
  departures=doc.getElementsByTagName('depart')
  arrivals=doc.getElementsByTagName('arrive')

  # 用 zip 将它们连在一起
  return zip([p.firstChild.data.split(' ')[1] for p in departures],
             [p.firstChild.data.split(' ')[1] for p in arrivals],
             [parseprice(p.firstChild.data) for p in prices])
```

注意：函数在结尾处得到了所有的 price、depart 和 arrive 标签。对它们三者而言，各自有相同数量的一组数据——每三个数据对应一次航班——因此我们可以用 zip 函数将它们连在一起，形成一个大列表中的若干元组。由于出发和到达的信息是以空格分隔的日期加时间的形式给出的，因此我们可以用函数将字符串分隔以得到时间值。函数还将价格传

递给了 parseprice，将其转化成浮点类型。

为了确保一切正常，现在我们可以在自己的 Python 会话中尝试一下实际的航班搜索（记住请把日期改成将来的某个时间）：

```
>>> import kayak
>>> sid=kayak.getkayaksession()
>>> searchid=kayak.flightsearch(sid,'BOS','LGA','11/17/2006')
>>> f=kayak.flightsearchresults(sid,searchid)
>>> f[0:3]
[(u'07:00', u'08:25', 60.3),
 (u'08:30', u'09:49', 60.3),
 (u'06:35', u'07:54', 65.0)]
```

航班数据是按价格排序的方式返回的，对于价格相同的航班，则按时间排序。这样的输出结果非常不错，因为正如此前提到的那样，出现于结果中的相似解是聚集在一起的。为了将该函数和其余代码整合在一起，我们唯一要做的就是，利用原先已从文件中加载进来的相同结构，给 Glass 一家的不同成员建立一个完整的日程安排。为此我们只须遍历人员列表，然后对往返航班进行搜索。请将 `createschedule` 函数加入 *kayak.py* 文件中：

```
def createschedule(people,dest,dep,ret):
  # 得到搜索用的会话 id
  sid=getkayaksession()
  flights={}

  for p in people:
    name,origin=p
    # 往程航班
    searchid=flightsearch(sid,origin,dest,dep)
    flights[(origin,dest)]=flightsearchresults(sid,searchid)

    # 返程航班
    searchid=flightsearch(sid,dest,origin,ret)
    flights[(dest,origin)]=flightsearchresults(sid,searchid)

  return flights
```

现在，我们可以尝试利用实际的航班数据为这一家进行航班安排的优化了。Kayak 的搜索过程可能要花费一定的时间，因此开始时可以将搜索范围限定在头两名家庭成员。请在你的 Python 会话中输入下列代码：

```
>>> reload(kayak)
>>> f=kayak.createschedule(optimization.people[0:2],'LGA',
... '11/17/2006','11/19/2006')
>>> optimization.flights=f
>>> domain=[(0,30)]*len(f)
>>> optimization.geneticoptimize(domain,optimization.schedulecost)
770.0
703.0
...
>>> optimization.printschedule(s)
   Seymour      BOS 16:00-17:20 $85.0 19:00-20:28 $65.0
    Franny      DAL 08:00-17:25 $205.0 18:55-00:15 $133.0
```

恭喜你！刚刚你已经根据实时的航班数据进行了一次优化的过程。由于搜索的范围变大了，因而我们不妨借此机会试验一下算法的最大执行速度（maximum velocity）和学习速率（learning rate）。

有很多种方式可以去扩展这一函数。我们可以将它与天气搜索结合起来，以找到价格合理、在旅游期间气候温暖的优化方案，或者将之与旅馆搜索结合起来，找到航班和旅馆住宿价格都合理的目标解。Internet 上有数以千计的站点提供了旅游目的地的相关数据，这些数据都可以为优化算法所利用。

尽管在日常搜索（searches per day）方面 Kayak API 还是有局限的，但是它的确为我们返回了可以直接与任何航班或旅馆进行交易的链接，这意味着我们可以很方便地将这个 API 集成到任何应用程序中去。

# 涉及偏好的优化

## Optimizing for Preferences

我们已经看到了一个可以用优化算法来解决问题的例子，但是还有许多表面上看似不相关的问题，也可以用同样的方法来解决。请记住，利用优化算法解决问题的基本要求是：问题本身有一个定义好的成本函数，并且相似的解会产生相似的结果。并非每一个具有此类特征的问题都能用优化算法来解决，但是优化算法很有可能会返回一些此前我们未曾考虑到的值得关注的结果。

本节我们将考查另一个不同的问题，这个问题很明显要借助优化算法来加以解决。其一般的表述是：如何将有限的资源分配给多个表达了偏好的人，并尽可能使他们都满意（或者根据他们的意愿，尽可能地满足他们的需要）。

## 学生宿舍优化问题

## Student Dorm Optimization

本节中的示例问题是，依据学生的首选和次选，为其分配宿舍。尽管这是一个非常具体化的例子，但是将这种情况推广到其他问题是非常容易的——完全相同的代码可以用于在线纸牌游戏中玩家的牌桌分配，也可以用于大型编程项目中开发人员的 bug 分配，甚或用于家庭成员中的家务分配。须要再次说明的是，这类问题的目的是为了从个体中提取信息，

并将其组合起来产生出优化的结果。

本例中有 5 间宿舍，每间宿舍有两个隔间，由 10 名学生来竞争住所。每一名学生都有一个首选和一个次选。我们新建一个名为 *dorm.py* 的文件，并添加宿舍列表和人员列表，以及每个人的两项选择：

```
import random
import math

# 代表宿舍，每个宿舍有两个可用的隔间
dorms=['Zeus','Athena','Hercules','Bacchus','Pluto']

# 代表学生及其首选和次选
prefs=[('Toby', ('Bacchus', 'Hercules')),
       ('Steve', ('Zeus', 'Pluto')),
       ('Andrea', ('Athena', 'Zeus')),
       ('Sarah', ('Zeus', 'Pluto')),
       ('Dave', ('Athena', 'Bacchus')),
       ('Jeff', ('Hercules', 'Pluto')),
       ('Fred', ('Pluto', 'Athena')),
       ('Suzie', ('Bacchus', 'Hercules')),
       ('Laura', ('Bacchus', 'Hercules')),
       ('Neil', ('Hercules', 'Athena'))]
```

马上你就会发现，每个人都不可能满足各自的首选，因为 Bacchus 仅有两个隔间，而想要住进去的人却有三个。将这些人中的任何一位安置于他们的次选宿舍中，都将意味着 Hercules 中没有足够的空间留给选择它的人。

为了易于理解，我们有意将这个问题设计得很小巧，但在真实生活中，问题也许会涉及成百上千名学生在更大数量的宿舍范围内竞争更多的住所。因为这个例子仅有大约 100 000 个可能的解，所以将所有解都尝试一遍并从中找到最优解是可能的。但是当每间宿舍有 4 个隔间时，这一数字会快速增长到上万亿。

和航班问题相比，本题解在表达上更需要一点技巧。理论上，我们可以构造一个数字序列，让每个数字对应于一名学生，表示我们将学生安置在了某一间宿舍。问题在于，这种表达方式无法在题解中体现每间宿舍仅限两名学生居住的约束条件。一个全零序列代表将所有人都安置在了 Zeus 宿舍，这不是一个有效的解。

解决这一问题的一种办法是让成本函数返回一个很高的数值，用以代表无效解，但是这将使优化算法很难找到次优的解（better solutions），因为算法无法确定返回结果是否接近于其他优解（good solutions），甚或是有效的解。一般而言，我们最好不要让处理器的时钟周期浪费在无效解的搜索上。

解决这一问题的更好办法是寻找一种能让每个解都有效的题解表示法。有效解未必是优解；它仅代表恰有两名学生被安置于每间宿舍内。要达到这一目的，一种办法是设想每间宿舍都有两个“槽”，如此，在本例中共计有 10 个槽。我们将每名学生依序安置于各空槽内——第一位可置于 10 个槽中的任何一个内，第二位则可置于剩余 9 个槽中的任何一个内，依次类推。

搜索的定义域必须满足这一约束。请在 *dorm.py* 中加入如下代码行：

```
# [(0,9),(0,8),(0,7),(0,6),...,(0,0)]
domain=[(0,(len(dorms)*2)-i-1) for i in range(0,len(dorms)*2)]
```

打印题解的代码示范了槽的工作方式。该函数首先创建一个槽序列，每两个槽对应一间宿舍。然后遍历题解中的每个数字，并在槽序列中找到该数字对应的宿舍号，表示学生被安置的宿舍。此函数将学生和与之对应的宿舍打印输出，随后再将槽从序列中删除，如此，其余学生便不会再被安置于该槽中了。待最后一次迭代结束之后，槽列为空，且每名学生及其对应宿舍也都打印完毕。请将函数加入 *dorm.py* 中：

```
def printsolution(vec):
  slots=[]
  # 为每个宿舍建两个槽
  for i in range(len(dorms)): slots+=[i,i]

  # 遍历每一名学生的安置情况
  for i in range(len(vec)):
    x=int(vec[i])

    # 从剩余槽中选择
    dorm=dorms[slots[x]]
    # 输出学生及其被分配的宿舍
    print prefs[i][0],dorm
    # 删除该槽
    del slots[x]
```

可以在你的 Python 会话中导入上述文件，并试着打印一个题解如下：

```
>>> import dorm
>>> dorm.printsolution([0,0,0,0,0,0,0,0,0,0])
Toby Zeus
Steve Zeus
Andrea Athena
Sarah Athena
Dave Hercules
Jeff Hercules
Fred Bacchus
Suzie Bacchus
Laura Pluto
Neil Pluto
```

如果你想改变数值查看不同的题解，请记住每个数值必须在合理的域值范围内。在序列中，第一项的值可以介于 0 到 9 之间，第二项的值则介于 0 到 8 之间，依次类推。如果我们设置的某个数值超出了合理的域值范围，则函数将会抛出异常。由于优化函数将会保证数值在域值范围之内( 该域值通过定义域参数来指定 ),因此我们在优化期间不会遇到此类问题。

## 成本函数

### The Cost Function

成本函数的工作方式与打印函数类似。构造一个槽序列，并将用过的槽删除。成本的计算值，是通过将学生的当前宿舍安置情况与他的两项选择进行对比而得到的。如果学生当前被安置的宿舍即是其首选宿舍，则总成本加 0；如果是次选宿舍则加 1；如果不在其选择之列，则加 3：

```
def dormcost(vec):
  cost=0
  # 建立一个槽序列
  slots=[0,0,1,1,2,2,3,3,4,4]

  # 遍历每一名学生
  for i in range(len(vec)):
    x=int(vec[i])
    dorm=dorms[slots[x]]
    pref=prefs[i][1]
    # 首选成本值为 0，次选成本值为 1
    if pref[0]==dorm: cost+=0
    elif pref[1]==dorm: cost+=1
    else: cost+=3
    # 不在选择之列则成本值为 3

    # 删除选中的槽
    del slots[x]

  return cost
```

在构造成本函数时有一条法则很有用，即：尽可能让最优解的成本为零（本例中的最优解就是将每个人都安置于其首选宿舍内）。在本例中，我们已经明确不存在最优解，但是知道最优解的成本为零，却可以使我们了解到目前与最优解的差距有多少。这一法则的另一个好处在于，当优化算法找到一个最优解时，我们可以让优化算法停止搜寻更优的解。

## 执行优化函数

### Running the Optimization

有了题解的表示形式、成本函数，以及结果打印输出函数，我们就可以执行此前定义好的优化函数了。请在你的 Python 会话中输入如下内容：

```
>>> reload(dorm)
>>> s=optimization.randomoptimize(dorm.domain,dorm.dormcost)
>>> dorm.dormcost(s)
18
>>> optimization.geneticoptimize(dorm.domain,dorm.dormcost)
13
10
...
4
>>> dorm.printsolution(s)
```

```
Toby Athena
Steve Pluto
Andrea Zeus
Sarah Pluto
Dave Hercules
Jeff Hercules
Fred Bacchus
Suzie Bacchus
Laura Athena
Neil Zeus
```

同样，我们可以调整输入参数，看一看遗传优化算法是否能更快地找到一个优解。

# 网络可视化

## Network Visualization

本章的最后一个例子将向大家展示优化算法的另一种用途，这与前述的其他问题丝毫没有任何关连性。我们要讨论的是网络的可视化问题。此处的网络，意指任何彼此相连的一组事物。像 MySpace、Facebook 或 LinkedIn 这样的社会网络便是在线应用领域中的一个极好的例子。在那里，人们因互为朋友或具备特定关系而彼此相连。网站的每一位成员可以选择与他们相连的其他成员，共同构筑一个人际关系网络。将这样的网络可视化输出，以明确人们彼此间的关系结构——例如寻找联络人（那些认识许多其他朋友的人，或是联系其他私人小圈子的人）——是一件颇有意义的事情。

## 布局问题

## The Layout Problem

为了展示一大群人及其彼此间的关联，我们将网络绘制成图，绘制时会遇到一个问题，我们应该如何安置图中的每个人名（或头像）呢？以图 5-7 中的网络为例。

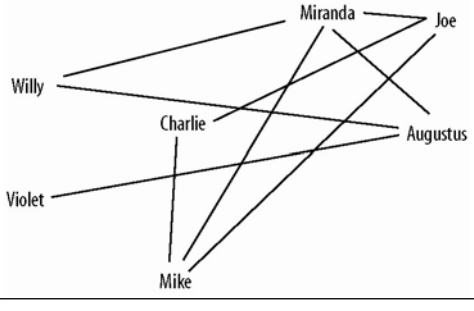


图 5-7：混乱的网络布局

在本图中，我们可以看到 Augustus 是 Willy、Violet 和 Miranda 的朋友。但是，网络的布局有点杂乱 ,而且增加更多的人会使布局非常地混乱不堪。一个更为清晰的布局如图 5-8 所示。

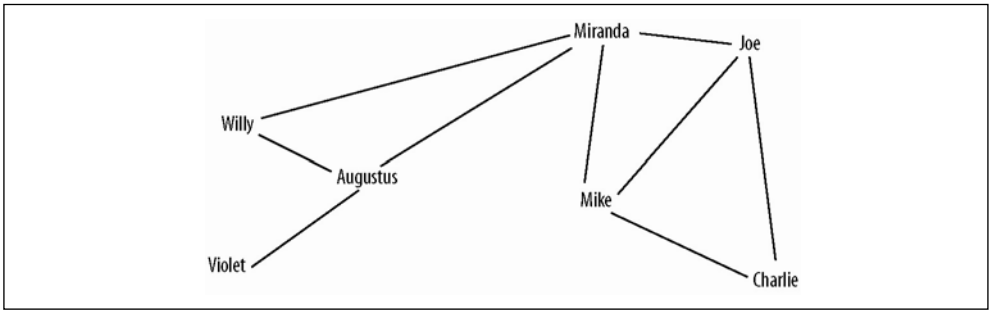



图 5-8：一个清晰的网络布局

本节我们将考虑如何运用优化算法来构建更好的而非杂乱无章的网络图。首先，我们新建一个名为 *socialnetwork.py* 的文件，并加入一些事实数据，这些数据代表着社会网络的某一个小部分：

```
import math

people=['Charlie','Augustus','Veruca','Violet','Mike','Joe','Willy','Miranda']

links=[('Augustus', 'Willy'),
       ('Mike', 'Joe'),
       ('Miranda', 'Mike'),
       ('Violet', 'Augustus'),
       ('Miranda', 'Willy'),
       ('Charlie', 'Mike'),
       ('Veruca', 'Joe'),
       ('Miranda', 'Augustus'),
       ('Willy', 'Augustus'),
       ('Joe', 'Charlie'),
       ('Veruca', 'Augustus'),
       ('Miranda', 'Joe')]
```

此处，我们的目标是要建立一个程序，令其能够读取一组有关于谁是谁的朋友的事实数据，并生成一个易于理解的网络图。要完成这项工作，通常须要借助于质点弹簧算法（mass-and-spring algorithm）。这一算法是从物理学中建模而来的：各结点彼此向对方施以推力并试图分离，而结点间的连接则试图将关联结点彼此拉近。如此一来，网络便会逐渐呈现出这样一个布局：未关联的结点被推离，而关联的结点则被彼此拉近——却又不会靠得很拢。

遗憾的是，质点弹簧算法无法避免交叉线。这使得我们很难在一个拥有大量连接的网络中观察结点的关联情况，因为追踪彼此交叉的连线是颇具难度的。不过，假如使用优化算法

来构建布局的话，那么我们只须要确定一个成本函数，并尝试令它的返回值尽可能地小。在本例中，一个值得一试的成本函数是计算彼此交叉的连线数。

## 计算交叉线

### Counting Crossed Lines

为了能够使用早先定义过的那些优化函数，我们须要将题解表示为一个数值序列。所幸的是，将这一特定问题表示成一个数值序列是非常容易的——每个结点都有 x 和 y 坐标，因此我们可以将所有结点的坐标放入一个长长的列表中：

```
sol=[120,200,250,125 ...
```

在上例中，Charlie 位于(120,200)，Augustus 位于(250,125)，凡此种种，不一而足。

随后，新的成本函数只须对彼此交叉的连线进行计数即可。有关两线交叉的公式出处，超越了本章讨论的范围，不过其基本思路就是计算线条的“分数值”（此处每条线都是“交叉”的）。如果两条线的分数值介于 0（表示线的一端）和 1（表示线的另一端）之间，则它们彼此交叉。反之，则不交叉。

该函数遍历每一对连线，并利用连线端点的当前坐标来判定它们是否交叉。如果交叉，则总分加 1。请将 crosscount 加入 *socialnetwork.py*：

```
def crosscount(v):
  # 将数字序列转换成一个 person:(x,y)的字典
  loc=dict([(people[i],(v[i*2],v[i*2+1])) for i in range(0,len(people))])
  total=0

  # 遍历每一对连线
  for i in range(len(links)):
    for j in range(i+1,len(links)):

      # 获取坐标位置
      (x1,y1),(x2,y2)=loc[links[i][0]],loc[links[i][1]]
      (x3,y3),(x4,y4)=loc[links[j][0]],loc[links[j][1]]

      den=(y4-y3)*(x2-x1)-(x4-x3)*(y2-y1)

      # 如果两线平行，则 den==0
      if den==0: continue

      # 否则，ua 与 ub 就是两条交叉线的分数值
      ua=((x4-x3)*(y1-y3)-(y4-y3)*(x1-x3))/den
      ub=((x2-x1)*(y1-y3)-(y2-y1)*(x1-x3))/den
```

```
        # 如果两条线的分数值介于 0 和 1 之间，则两线彼此交叉
        if ua>0 and ua<1 and ub>0 and ub<1:
          total+=1
    return total
```

上述搜索算法的定义域就是每组坐标的域值范围。举例而言，假设我们将网络绘制于400*400 像素的图中，则为了留出一定的页边，定义域可以稍小于该范围值。请将下列代码行加入 *socialnetwork.py* 的末尾处：

```
domain=[(10,370)]*(len(people)*2)
```

现在，我们可以尝试利用前述的优化算法，以寻找极少有连线交叉情况的题解了。请将 *socialnetwork.py* 导入你的 Python 会话中，并尝试几种优化算法：

```
>>> import socialnetwork
>>> import optimization
>>> sol=optimization.randomoptimize(socialnetwork.domain,socialnetwork.crosscount)
>>> socialnetwork.crosscount(sol)
12
>>> sol=optimization.annealingoptimize(socialnetwork.domain,
    socialnetwork.crosscount,step=50,cool=0.99)
>>> socialnetwork.crosscount(sol)
1
>>> sol
[324, 190, 241, 329, 298, 237, 117, 181, 88, 106, 56, 10, 296, 370, 11, 312]
```

利用模拟退火算法有可能会找到极少有连线交叉情况的题解，但是返回的结果却是难以理解的坐标序列。下一节中，我们将向大家展示如何编写程序来自动绘制网络。

## 绘制网络

### Drawing the Network

绘制网络须要用到第 3 章中曾经使用过的 Python Imaging Library。如果还没有安装该库，请参考附录 A，按其指示获取该库的最新版本，并将之安装到你的 Python 实例中。

绘制网络的代码非常简单易懂。全部代码要做的工作包括：建立一个 image 对象，绘制介于不同人之间的连线，并为每个人绘制相应的结点。我们将人名放到最后绘制，这样就不会被连线遮盖了。请将该函数加入 *socialnetwork.py*：

```
def drawnetwork(sol):
  # 建立 image 对象
  img=Image.new('RGB',(400,400),(255,255,255))
  draw=ImageDraw.Draw(img)

  # 建立标示位置信息的字典
  pos=dict([(people[i],(sol[i*2],sol[i*2+1])) for i in range(0,len(people))])
```

```
  # 绘制连线
  for (a,b) in links:
    draw.line((pos[a],pos[b]),fill=(255,0,0))

  # 绘制代表人的结点
  for n,p in pos.items():
    draw.text(p,n,(0,0,0))

  img.show()
```

要在你自己的 Python 会话中执行本函数，只须重新载入模块，调用该函数，并传入你的题解即可：

```
>>> reload(socialnetwork)
>>> drawnetwork(sol)
```

图 5-9 给出了一种可能的优化结果。

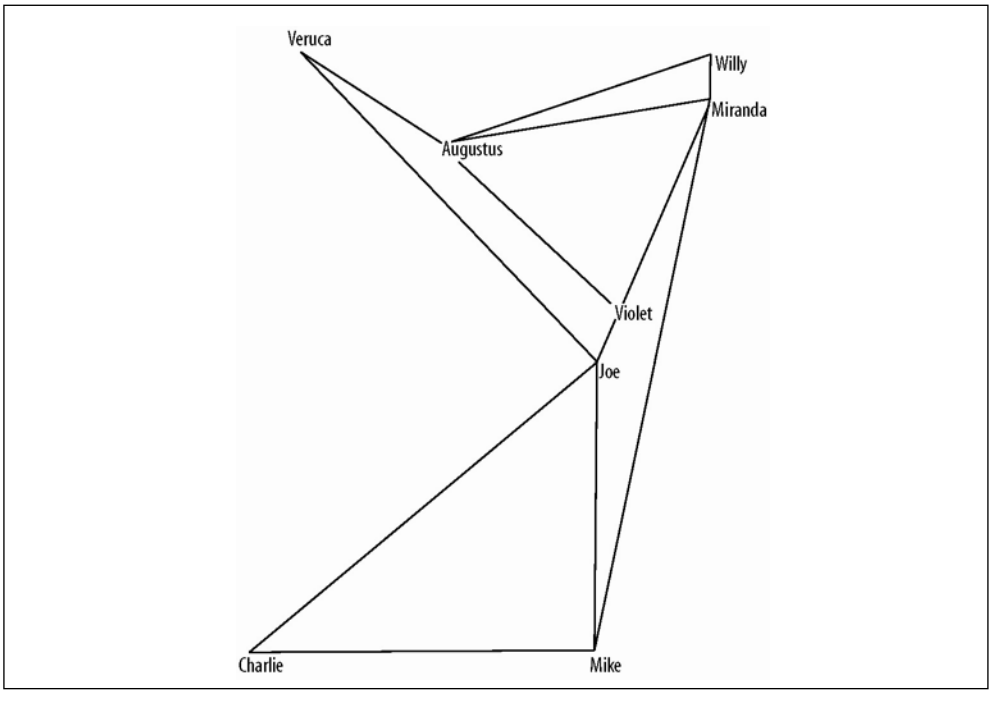



图 5-9：由无交叉连线（no-crossed-lines）优化算法形成的布局

当然，你的题解可能会有别于上述计算结果。有时题解可能会看起来非常的古怪——这是因为我们的目标只是令交叉线的数目最小化，成本函数并没有排除诸如两线夹角非常小，或两结点间距非常近这样的布局情况。就这一点而言，优化算法就像是一个忠实满足你愿

望的精灵（angles）一样。所以，清楚你到底想要什么是非常重要的。时常会有题解满足原本的“最优”条件，但却并非是我们想要的结果。

如果要对一个两结点放置太近的题解进行“判罚”（penalize），最简单的办法就是计算两结点间的距离并除以一个预期的最小距离。我们可以将如下代码添加至 `crosscount` 的末尾处（`return` 语句之前），以提供这一附加的判断。

```
for i in range(len(people)):
  for j in range(i+1,len(people)):
    # 获得两结点的位置
    (x1,y1),(x2,y2)=loc[people[i]],loc[people[j]]

    # 计算两结点的间距
    dist=math.sqrt(math.pow(x1-x2,2)+math.pow(y1-y2,2))
    # 对间距小于 50 个像素的结点进行判罚
    if dist<50:
      total+=(1.0-(dist/50.0))
```

当一对结点的彼此间距小于 50 个像素时，上述代码将产生一个比原来更高的成本值，该值与其距离的远近成比例。如果两结点恰好位置相重，则其值为 1。再次执行优化算法，看看能否形成一个分布较为开阔的布局。

## 其他可能的应用场合

### Other Possibilities

本章向大家展示了优化算法的三种截然不同的应用，但这只是众多可能的应用场合中很小的一部分。正如本章一再重申的，关键步骤在于确定题解的表示法及成本函数。如果能做到这些，那么我们就有机会利用优化算法来对问题进行求解。

关于优化，有这样一项应用也许是值得关注的：或许我们会希望对一群人进行分组，让组员的技能得以均匀分布。在一个小型的竞赛活动中，我们可能希望将参赛者进行组队，使每个队都能在体育、历史、文学，以及电视方面具备足够的知识。另一种可能的应用场合是根据人们的技能搭配情况，为项目组分派任务。优化算法可以找到任务分解的最佳方案，从而使任务列表得以在最短时间内完成。

假设有一个标注了关键字的长长的网站列表，根据用户提供的关键字来寻找一组最佳网站可能也是一件很有意义的事情。最佳网站中所包含的网站，并不需要具备大量彼此公有的关键字，而是要尽可能多地体现由用户提供的关键字。

# 练习

## Exercises

1. 组团旅行的成本函数　请以飞机上每分钟 0.50 美元的成本将总飞行时间计入成本。然后再尝试追加 20 美元的罚款，以确保任何人都能在上午 8 点之前抵达机场。

2. 退火算法的初始值　模拟退火算法的结果很大程度上取决于其初始值。请构造一个新的优化函数，用多个初始值来模拟退火，并返回最优解。

3. 遗传优化算法的结束条件　本章中的函数是以固定迭代次数来进行遗传优化的。请改变算法的结束条件，使其在经过 10 次迭代之后，任一最优解都没有任何改善时，方才结束。

4. 往返定价　此前通过 Kayak 获取航班数据的函数查找的仅是单程航班。购买往返机票的价格可能会更加便宜。请修改代码取得往返票价，并修改成本函数，令其针对某一特定往返航班进行票价查询，而不是只对单程票价进行求和运算。

5. 学生组对　假设并非要求学生列出对宿舍的偏好，而是令其表达对同住舍友的偏好。那么你将如何表达学生组对的结果呢？成本函数又将如何定义呢？

6. 连线夹角的判断　请在连接同一人的两线夹角非常小的时候，为网络布局算法的成本函数再增加一项成本。( 提示：可以使用向量的叉乘。)