Genetic Algorithms

Another set of techniques for optimization, also inspired by nature, is called *genetic algorithms*. These work by initially creating a set of random solutions known as the *population*. At each step of the optimization, the cost function for the entire population is calculated to get a ranked list of solutions. An example is shown in Table 5-1.

Table 5-1. Ranked list of solutions and costs

Solution	Cost
[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

After the solutions are ranked, a new population — known as the next *generation* — is created. First, the top solutions in the current population are added to the new population as they are. This process is called *elitism*. The rest of the new population consists of completely new solutions that are created by modifying the best solutions.

There are two ways that solutions can be modified. The simpler of these is called *mutation*, which is usually a small, simple, random change to an existing solution. In this case, a mutation can be done simply by picking one of the numbers in the solution and increasing or decreasing it. A couple of examples are shown in Figure 5-3.

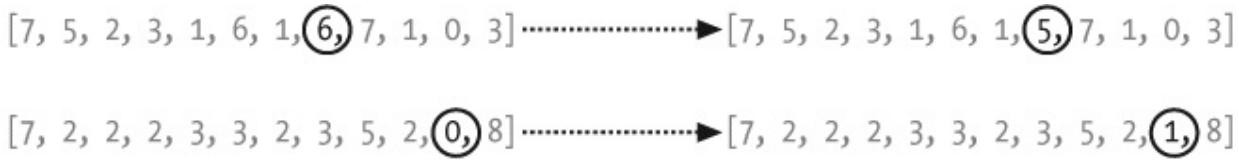

Figure 5-3. Examples of mutating a solution

The other way to modify solutions is called *crossover* or *breeding*. This

method involves taking two of the best solutions and combining them in some way. In this case, a simple way to do crossover is to take a random number of elements from one solution and the rest of the elements from another solution, as illustrated in Figure 5-4.

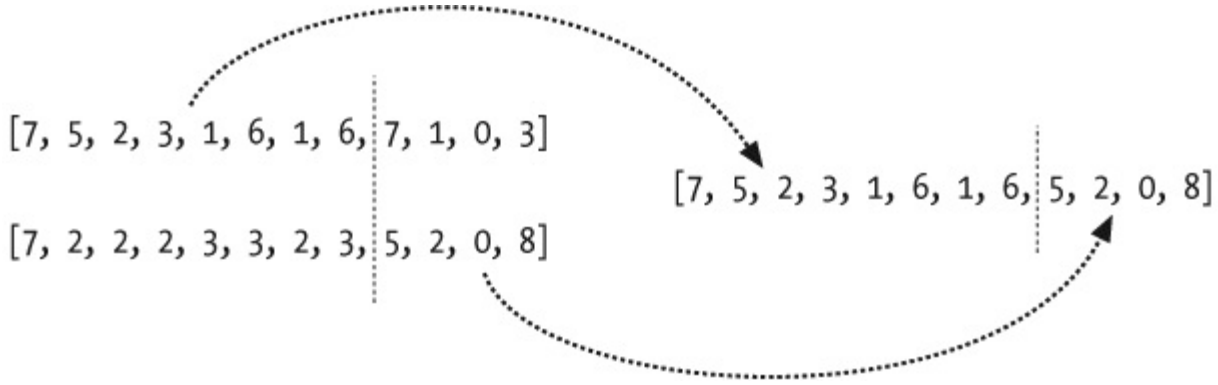

Figure 5-4. Example of crossover

A new population, usually the same size as the old one, is created by randomly mutating and breeding the best solutions. Then the process repeats — the new population is ranked and another population is created. This continues either for a fixed number of iterations or until there has been no improvement over several generations.

Add `geneticoptimize` to *optimization.py*:

```
def geneticoptimize(domain,costf,popsize=50,step=1,
                    mutprob=0.2,elite=0.2,maxiter=100):
  # Mutation Operation
  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:]

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

  # Build the initial population
  pop=[]
  for i in range(popsize):
    vec=[random.randint(domain[i][0],domain[i][1])
         for i in range(len(domain))]
    pop.append(vec)

  # How many winners from each generation?
  topelite=int(elite*popsize)
```

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

    # Start with the pure winners
    pop=ranked[0:topelite]

    # Add mutated and bred forms of the winners
    while len(pop)<popsize:
      if random.random(  )<mutprob:
        # Mutation
        c=random.randint(0,topelite)
        pop.append(mutate(ranked[c]))
      else:

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

    # Print current best score
    print scores[0][0]

  return scores[0][1]
```

This function takes several optional parameters:

`popsize`

The size of the population

`mutprob`

The probability that a new member of the population will be a mutation rather than a crossover

`elite`

The fraction of the population that are considered good solutions and are allowed to pass into the next generation

`maxiter`

The number of generations to run

Try optimizing the travel plans using the genetic algorithm in your Python session:

```
>>>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
```

In Chapter 11, you'll see an extension of genetic algorithms called *genetic programming*, where similar ideas are used to create entirely new programs.

TIP

The computer scientist John Holland is widely considered to be the father of genetic algorithms because of his 1975 book, *Adaptation in Natural and Artificial Systems* (University of Michigan Press). Yet the work goes back to biologists in the 1950s who were attempting to model evolution on computers. Since then, genetic algorithms and other optimization methods have been used for a huge variety of problems, including:

- Finding which concert hall shape gives the best acoustics
- Designing an optimal wing for a supersonic aircraft
- Suggesting the best library of chemicals to research as potential drugs
- Automatically designing a chip for voice recognition

Potential solutions to these problems can be turned into lists of numbers. This makes it easy to apply genetic algorithms or simulated annealing.

Whether a particular optimization method will work depends very much on the problem. Simulated annealing, genetic optimization, and most other optimization methods rely on the fact that, in most problems, the best solution is close to other good solutions. To see a case where optimization might not work, look at Figure 5-5.

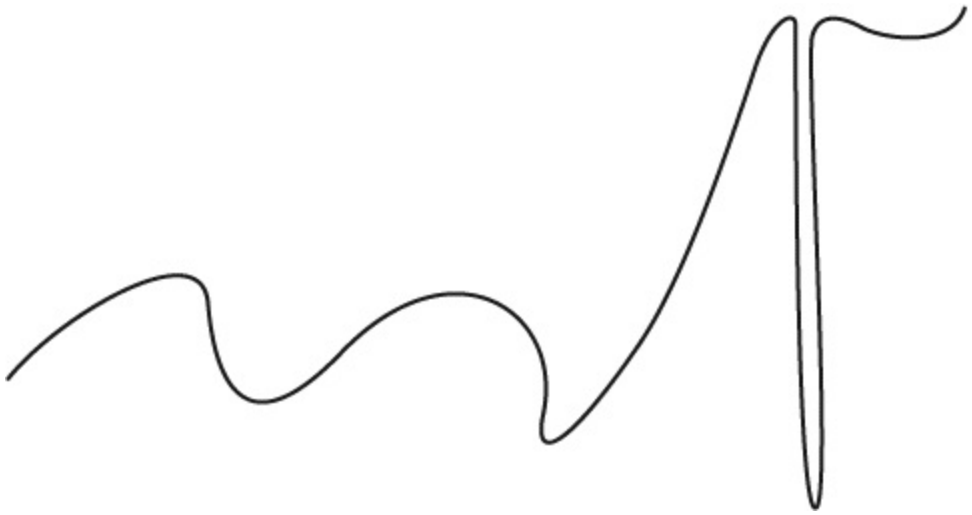

Figure 5-5. Poor problem for optimization

The cost is actually lowest at a very steep point on the far right of the figure. Any solution that is close by would probably be dismissed from consideration because of its high cost, and you would never find your way to the global minimum. Most algorithms would settle in one of the local minima on the left side of the figure.

The flight scheduling example works because moving a person from the second to the third flight of the day would probably change the overall cost by a smaller amount than moving that person to the eighth flight of the day would. If the flights were in random order, the optimization methods would work no better than a random search — in fact, there’s no optimization method that will consistently work better than a random search in that case.