Chapter 3

Optimizing Real-Valued Functions

This chapter will expand the concept of string evolver, changing the alphabet from the character set to the set of real numbers. This will lead to our first problem of interest outside of the artificial life community: maximization and minimization of real-valued functions of real variables. We will call strings of real numbers by the more standard name *arrays* of real numbers. A knowledge of calculus is helpful, and a few pertinent facts are included in Appendix B. We have already previewed this area in various homework problems, 1.12, 2.15, and in Experiment 2.8. Using real functions gives us some additional machinery. For example, we can use a much more efficient Lamarckian mutation when the function is differentiable. This mutation operator is explained in Appendix B. In this chapter, we will have our first nontrivial representation issues: comparing atomic and non-atomic representations of real numbers. (Recall that *atom* is from the Latin *a* (not) *tome* (cut), meaning uncuttable.)

Some terminology concerning optima will be useful in this chapter. An *optimum* is a minimum or maximum of a function. An optimum is said to be *local* is it is larger (respectively smaller) than all nearby points. An optimum is *global* is it is larger (respectively smaller) than every other value the function takes on over its domain space. Since minimizing f is equivalent to maximizing $-f$ we will speak in the remainder of the chapter as if we are maximizing functions and as if our optima are maxima. Following terminology in statistics, optima are also sometimes called *modes* and a function with only one optimum, e.g., $f(x) = 1 - x^2$, is said to be *unimodal*. Compare these with the definitions of mode and unimodal and local and global optimum from Chapter 2.

3.1 The Basic Real Function Optimizer

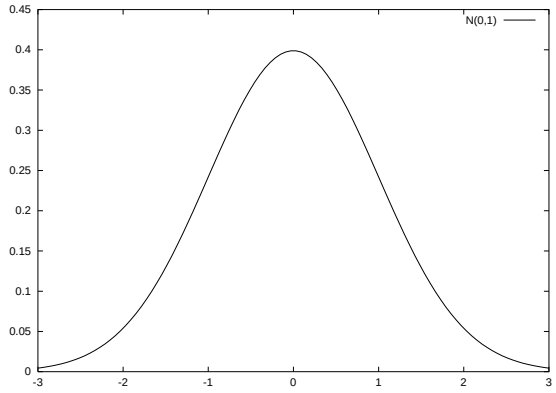
The crossover operators on strings, given in Section 2.2, carry over directly as crossover operators on arrays of real numbers. In one representation used for optimization of real-valued functions, real numbers are represented as strings of characters (representing bits), and the crossover operators are allowed to cleave real numbers "in the middle." This practice causes rather bizarre behavior in the cleaved real numbers, and so we will avoid it for the most part by forcing our crossover operators to respect real boundaries. Where in the last chapter, our notion of point mutation was to replace a character with another randomly generated character, a point mutation for a real number will consist of adding or subtracting small values to some locus of the gene.

Definition 3.1 *For a real number ϵ, we define a* **uniform real point mutation** *of a gene comprised of an array of real numbers to be addition of a uniformly distributed random number in the range $[-\epsilon, \epsilon]$ to a randomly chosen locus in the gene. The number ϵ is called the* maximum mutation size *of the mutation operator.*

A uniform real point mutation with maximum mutation size ϵ causes one of the loci in a gene to jump to a new value within ϵ of its current value. All numbers that can be reached by the mutation are equally likely. This definition of point mutation gives us an uncountable suite of mutation operators by varying the maximum mutation size continuously. Usually the problem under consideration suggests reasonable values for ϵ. Once we have a point mutation we can build from it the one, two, and k-point mutations, probabilistic mutations, and the helpful mutations described in Section 2.3. The "all" clause in the definition of Lamarckian mutation makes it impossible to use that definition with the definition of point mutation we have above but, in Appendix B, we define a derivative based Lamarckian mutation for differentiable functions.

Definition 3.2 *For a real number σ, we define a* **Gaussian real point mutation** *of a gene comprised of an array of real numbers to be addition of a normally distributed random number with mean zero and standard deviation σ to a randomly chosen locus in the gene.*

A Gaussian real point mutation shares with the uniform point mutation the property that its average is zero. Therefore, increasing the value of the number hit with the mutation is symmetric to decreasing its value. The Gaussian mutation differs in being able to go farther. Where all possible numbers are equally likely with a uniform mutation, the chance of reaching a number drops off rapidly with a Gaussian mutation. The normal distribution has infinite tails: there is a positive probability of getting *any number at all* back from a Gaussian mutation. Uniform mutation insists on local search; Gaussian mutation permits distant search, but usually performs local search. The shape of the Gaussian distribution



$$P(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

Figure 3.1: A graph of the distribution of the Gaussian distribution with mean zero and deviation 1 (Note that the tails of the distribution continue indefinitely.)

is shown in Figure 3.1. Generating Gaussian random variables with mean 0 and standard deviation 1 can be done with the formula given in Equation 3.1.

$$N(0,1) = Cos(2\pi \cdot R) \times \sqrt{-2 \cdot S}, \tag{3.1}$$

where R and S are independently generated uniform random variables in the range $[0, 1]$. In order to get Gaussian random numbers with standard deviation σ simply multiple the results by σ.

In the rest of this chapter, save for Section 3.5, we will be testing real function optimizers on problems with known answers. We will say we have "found" an answer when a member of our evolving population of candidate solutions is within a specified Euclidian distance, called a *tolerance*, of the true answer. Suppose we are optimizing for testing purposes the function $f(x) = x^2$. If we were working with a tolerance of 0.01, then any population member whose genes code for a number in the range $-0.01 \leq x \leq 0.01$ would serve as a witness that the evolving population had "solved" the problem of finding the minimum at $x = 0$.

Since models of evolution are independent of the problem being solved, we have all of the models of evolution discussed so far available. Selecting a basic real function optimizer requires that we go through the following steps. First, pick your fitness function. In the basic optimizer this function will be the function being optimized or its negative. Second, select a population size. The population will be made of n-dimensional arrays of reals where n is the number of variables in the function being optimized. Third, select a suite of variation operators. You may want a mix of mutations that include different maximum mutation sizes so that some mutations do a broader search and others do a close search. The smallest maximum mutation size in any of your mutation operators should be small enough that you are unlikely to jump entirely past a nearby optimum with one mutation. The fact

that your mutations can always be as small as the real number resolution of the computer you're working with keeps this from becoming an urgent concern. Fourth, select a model of evolution. Fifth and last, come up with a stopping condition.

Step five is a killer. Any good stopping condition that can say "Yes, this is it! We have found the true global optimum!" practically has to contain the answer to the optimization problem in order to work as advertised. When you are testing a real function optimizer on functions with known optima, optima you yourself have constructed within the function, an omniscient stopping condition is available. Normally, however, you have to make do with a second rate, substitute stopping condition. A couple of demonstrably bad but widely used stopping conditions are: (i) to stop if there has been no change in a long time or (ii) to run your real function optimizer as long as you can. Both of these stopping conditions can be marginally improved by rerunning the algorithm on different populations and seeing what sort of optima pop out. The values you get can be used to create new stopping conditions or restarting conditions. If, for example, you know of a much better optimum than the one you are in, and no progress has been made for several generations, it might well be profitable to restart evolution with a new random population. You might even share time among several evolving populations and delete the ones that appear to be going nowhere, replacing them with copies of the populations that are doing better. This is, in essence, an evolutionary algorithm whose individual creatures are evolutionary algorithm populations. This sort of thing is called a *meta-selection algorithm* and is complex enough to be a technique of last resort.

In practical or applied problems, a stopping condition is often implied by the bounds implicit in reality. If your optimizer is stuck and has found a solution that brings you in under budget, you can just accept that solution and go home for the night. An applied problem usually has obvious bounds on optima: the speed of light, the size of the federal budget, the amount of zinc in North America, or whatever. Finally, in many optimization problems there is an extensive literature containing proven or estimated bounds that you can use as stopping conditions. If you are intending to use evolutionary algorithms for optimization, then you should become familiar with the problems you are solving to the greatest extent possible and use that knowledge to make the best stopping function you can.

In his doctoral thesis Kenneth DeJong proposed five standard test functions, f_1-f_5 for evolutionary algorithms that optimize real functions. We do not use all of these functions in our experiments and the test suite is included for completeness and because of its historical importance in real function optimization with evolutionary algorithms. We generalize the test bed to the extent of making the number of variables arbitrary in some cases. Many of the functions in the test bed have optima which are very special points, e.g., the origin. It is considered good practice to shift the optima to points that may not have special status in the representation used.

$$f_1(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i^2 \tag{3.2}$$

with $-5.12 \leq x_i \leq 5.12$. This function is to be minimized.

$$f_2(x, y) = 100(x^2 - y^2)^2 + (1 - x)^2 \tag{3.3}$$

with $-2.048 \leq x, y \leq 2.048$. This function is to be minimized.

$$f_3(x_1, \ldots, x_n) = \sum_{i=1}^{n} [x_i] \tag{3.4}$$

with $-5.12 \leq x_i \leq 5.12$ where $[x]$ is the greatest integer in x. This function may be minimized or maximized.

$$f_4(x_1, \ldots, x_n) = \sum_{i=1}^{n} i \cdot x_i^4 + \text{Gauss}(0,1) \tag{3.5}$$

with $-1.28 \leq x_i \leq 1.28$ and where Gauss(0,1) is a Gaussian random variable with a mean of zero and a standard deviation of 1. Recall that the formula for Gaussian random numbers are given in Equation 3.1. The Gaussian random variable is added each time the function f_4 is called. This function is to be minimized.

$$f_5 = 0.002 + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{n} (x_i - a_{i,j})^6}, \tag{3.6}$$

with $-65.536 \leq x_i \leq 65.536$ and where the $a_{i,j}$ are coordinates of a set of points within the function's domain generated at random and fixed before optimization takes place. This function is to be maximized.

Let us look qualitatively at these functions. The function, f_1, has a single mode at the origin and is the simplest imaginable real function that has an optimum. The function, f_2, is still not hard, but has two areas that look like optima locally, with only one being a true optimum. The function, f_3, is unimodal, but, where it is not constant, it is discontinuous. It, thus, serves as a good example of a function where the techniques of calculus are useless. The function, f_4, is also unimodal with a mode at the origin, but it is flatter near its optimum than f_1 and has random noise added to it. The function, f_5, simply has a large number of extremely tall, narrow optima with differing heights at the positions $(a_{i,j} : i = 1 \ldots n)$. It could be used in a destruction test for mutation operators that made too large jumps.

DeJong's function test bed is a standard for testing evolutionary computation systems. Other functions can be used as building blocks to construct our own test functions. The

Fake Bell Curve, Equation 2.1, is one. Another is the *Sombrero Function*

$$\cos\left(\sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}\right). \tag{3.7}$$

To see where this function got its name look at the two-dimensional version on a square of side length 4π centered at the origin.

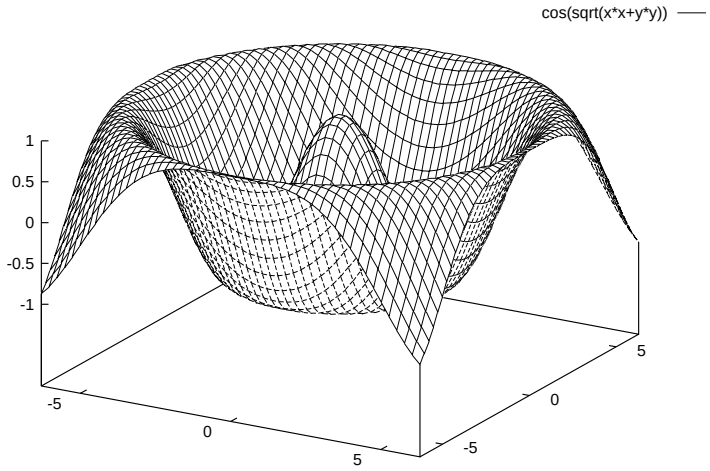


Figure 3.2: The Sombrero Function in 2-D, $-2\pi \leq x, y \leq 2\pi$

Experiment 3.1 *Write or obtain software for a real function optimizer for use on functions 1-5 given below. Use tournament selection with tournament size 4, two point crossover, uniform real single point mutation with* $\epsilon = 0.2$*, used on each new creature. Stop after 100 generations on functions 1-3, stop within a Euclidian distance of 0.001 of the true optimum on functions 4-5. Use a population size of 120 individuals. For functions 1-3 do 30 runs and give the solutions found. See if they cluster in any way. For functions 4 and 5, find the mean and standard deviation of time-to-solution, using a tolerance of* 0.001*, averaged over 30 runs. Be sure your definition of two point crossover does not choke in 2 dimensions; it should act like one point crossover in this case.*
1) Minimize:

$$x^4 + 2x^2y^2 + y^4 - 6x^3 - 10x^2y - 6xy^2 - 10y^3 + 37y^2 + 37x^2 - 12x - 20y + 70, \ -10 \leq x, y \leq 10$$

.

2) Maximize

$$\frac{\cos\left(5 \cdot \sqrt{x^2+y^2}\right)}{x^2+y^2-6x-2y+11},\ 0 \leq x, y \leq 5$$

.

3) Maximize and Minimize:

$$\frac{3xy-2}{4x^2-4xy+y^2+4x-2y+2},\ -5 \leq x, y \leq 5$$

.

4) DeJong function f_1, Equation 3.2, in 10 dimensions.

5) Fake bell curve, Equation 2.1, in 4 dimensions, with $-2 \leq x_i \leq 2$.

In Experiment 3.1, functions 1-3 have optima at fairly weird points. The functions avoid "working out even" and the optima are even hard to find by staring at a graph of the functions, unless you play games with vertical scale. Functions 4-5 commit the sin of having optima at special points - the origin - and are the functions you should probably use when debugging your code.

Experiment 3.2 *Redo functions 4 and 5 of Experiment 3.1, replacing single point mutation with (i) a probabilistic mutation operator that yields the same overall mutation rate as the one point mutation used in Experiment 3.1, (ii) two point mutation applied to each new creature, (iii) a probabilistic mutation that yields the same overall mutation rate as two point mutation, and (iv) Gaussian mutation with standard deviation* 0.1*. In your write up, compare the performance by drawing confidence intervals of one standard deviation about the mean solution times and checking for overlap.*

Experiment 3.3 *Redo Experiment 3.2, using single point mutation and also replacing the single point mutation with the Lamarckian mutation operator from Appendix B in 10% of the mutations. Compare the performance with the algorithm used in Experiment 3.2.*

Experiment 3.4 *Redo Experiment 3.2, using only the mutation operator you found to work best (this may be different for different functions). Rewrite the evolutionary algorithm to use one or two point crossover or uniform crossover. Compare the performance of the three different types of crossover, measuring time-to-solution in both crossover events and real numbers generated.*

In the next experiment, we will explore a representational issue: how does representing points in the domain space of a function with real numbers compare to a discrete representation of character strings coding for bits?

Experiment 3.5 *Redo Experiment 3.2 with two point crossover, but this time radically modify your data structure be strings of 0s and 1s, twelve per real variable. Interpret these as real numbers by taking the characters in blocks of 12 as binary integers and then mapping those integers uniformly into the specified range of the function. These integers are in the range* $0 \leq x \leq 4095$ *so to map them onto a real* y *in the interval* (r, s) *set* $y = (s - r) \cdot x/4095 + r$*. Let the mutation operator be single point mutation. Compare the performance of this evolutionary algorithm with that of Experiment 3.2. Discuss the connection, if any, between this experiment and the Royal Road function with block size 12.*

We conclude by reminding the reader of a standard and possibly useful formula.

Definition 3.3 *The* **Euclidian distance** *between two points* $p = (p_1, p_2, \ldots, p_n)$ *and* $q = (q_1, q_2, \ldots, q_n)$ *in* $\mathbb{R}^n$ *is given by the formula*

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2}.$$

Problems

Problem 3.1 Essay. *Explain in detail why the unmodified Lamarckian mutation operator from Chapter 2 cannot be reasonably applied with the definition of point mutation given in this section.*

Problem 3.2 *Suppose we are optimizing some real-valued function in 4 dimensions with an evolutionary algorithm. If we are using single point mutation with maximum mutation size* ϵ *then what is the expected distance that 1, 2, 3, 4, or 5 single point mutations will move us? You answer should be in terms of* ϵ*.*

Problem 3.3 *Do Problem 3.2 experimentally for* $\epsilon = 0.2$*. In other words, generate* $k = 1, 2, 3, 4,$ *or* 5 *single point mutations of the the gene* $(0, 0, 0, 0)$ *and check the average resulting difference for a large number of trials (e.g., thousands). Use the results to check your theoretical predictions.*

Problem 3.4 *A* neighborhood of radius r *of a point* p *in* $\mathbb{R}^n$ *is the set of all points distance* r *of* p*. Give an example of a continuous, nonconstant function from* $\mathbb{R}^2$ *to* $\mathbb{R}$ *and a point* p *so that for all choices of small* r *neighborhoods of* p *contain a point* q *other than* p *so that* $f(p) = f(q)$ *and for no* x *in the neighborhood is* $f(x) > f(p)$*.*

Problem 3.5 *This problem requires you to compute functional gradients as described in Appendix B. For each or the following functions and points, compute a unit vector in the direction of maximum increase at the specified point.*

(i) $f(x) = \cos(\frac{x}{x^2+1})$ *at* $x = 2$*.*

(ii) $f(x,y) = \frac{xy}{x^2+y^2+1}$ *at* $(1,2)$.

(iii) $f(x,y,z) = \frac{\cos(x)\sin(y)}{(z)^2+1}$ *at* $(\pi/3, \pi/6, 1)$.

Problem 3.6 *This problem requires you to compute functional gradients as described in Appendix B. Compute and display the gradient vectors within the square* $-5 \leq x, y \leq 5$ *of the function*

$$f(x,y) = \cos\left(\sqrt{x^2+y^2}\right).$$

You should compute the gradient at 25 points within the square in a regular $5x5$ *grid.*

Problem 3.7 *Give a function with an infinite number of local optima with distinct values on the interval* $-1 \leq x \leq 1$. *Can such a function be made to have a global optimum? Can such a function be made not to have a global optimum? For both questions given an example if the answer is yes and a proof if it is no.*

Problem 3.8 *The Fake Bell Curve, Equation 2.1, produces values in the range* $0 \leq f(x,y) \leq 1$. *Assume we are optimizing this curve. If we look at the change between maximum fitness of a pair of parents and maximum fitness of the children resulting from two point crossover of those parents, give the range of changes that are possible. In other words, what possible fitness difference can result from best parent to best child as a result of crossover? Assume this is the type of crossover that treats creatures as strings of indivisible reals.*

Problem 3.9 *Do the results change any in Problem 3.8 if we switch the data representation to that used in Experiment 3.5?*

3.2 Fitness Landscapes

Evolutionary algorithms operate on populations of structures. These populations of structures are not all the same (if they were there would be no point in having them). Over algorithmic time the population changes as better structures are generated and selected. The space in which the population lives is called the *fitness landscape*. The fitness landscape is a metaphor that helps us to understand the behavior of evolutionary algorithms. Let us come up with a formal definition sufficiently general to permit its use in almost any evolutionary computation system.

Definition 3.4 *Let* G *be the space of all the data structures that could appear as members of the population in some evolutionary algorithm. Then, the* **fitness landscape** *of that evolutionary algorithm is the graph of the fitness function over the space given by* G.

As with many of the definitions in this text, Definition 3.4 cheats. It refers to the space of all data structures upon which the evolutionary algorithm in question might work. To generate an organized graph, there must be some sort of relationship among the data structures. In this chapter, this isn't a problem, because all the data structures are points in $\mathbb{R}^n$. We are used to doing graphs over $\mathbb{R}^n$. When our evolutionary computation shifts to discrete structures, as in genetic programming, the structure of the domain space of the fitness landscape becomes far more complex.

For the type of evolutionary algorithms described in Section 3.1, the graph of the fitness function is the fitness landscape. In the next couple of sections of this chapter, we will attempt to enhance the performance of our evolutionary algorithms by modifying the fitness landscape or making the landscape dynamic. We will now examine the notion of fitness landscapes.

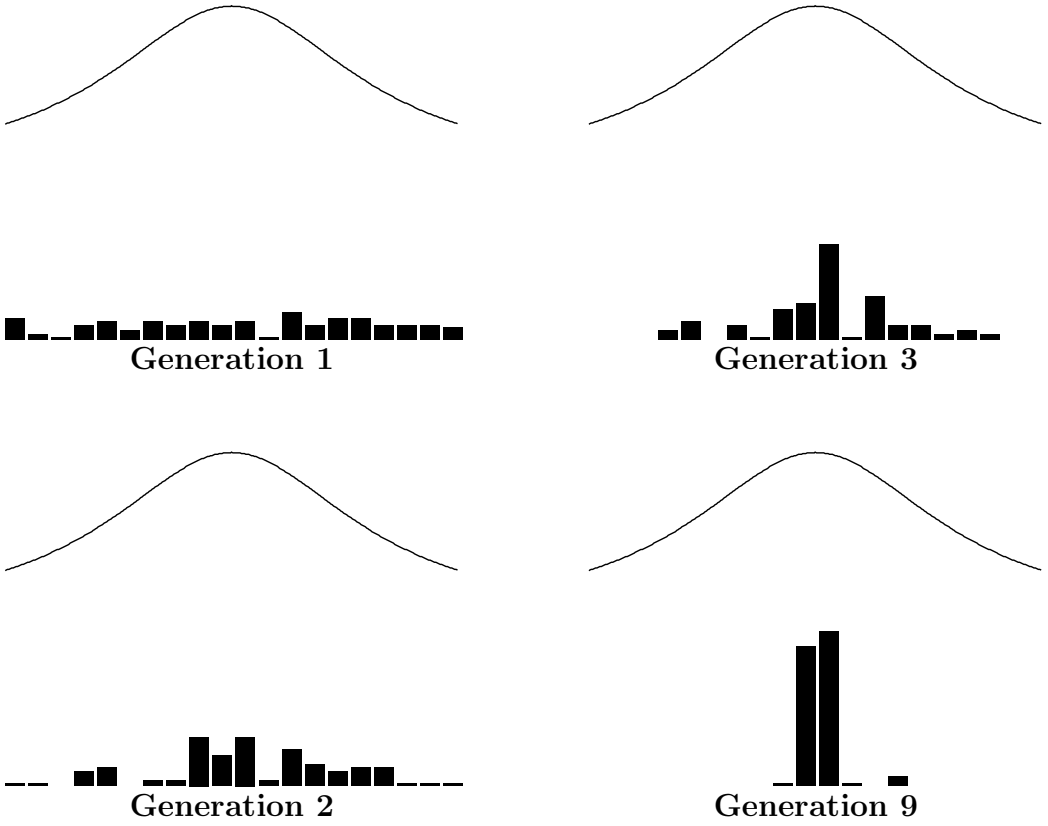


Figure 3.3: Graph of the Fake Bell Curve with one variable together with a histogram of the distribution of the population in the range [-3,3] at various times during evolution (The evolutionary algorithm which generated these pictures used roulette selection on a population of 100 data structures with uniform single point mutation of size 0.03 and no crossover. The algorithm is elitist, saving its best two genes.)

Examine Figure 3.3. The Fake Bell Curve in one dimension is extremely simple to optimize and, as we see, the population rapidly moves from a disordered random state to a pile-up beneath the optimum. Since the evolutionary algorithm running under Figure 3.3 had a relatively small mutation size and no crossover, this pile-up is probably mostly due to reproduction and selection. Since the Fake Bell Curve has a single mode, nothing can go wrong in optimizing it with an evolutionary algorithm. Selection and elitism ensure good genes remain, a "fitness ratchet," and genes near the best genes are generated at random. This means that the population will rapidly pile up near the unique optimum. There is no way to trap the population away from the optimum - all roads are uphill toward the unique maximum. Let's look at a less friendly function.

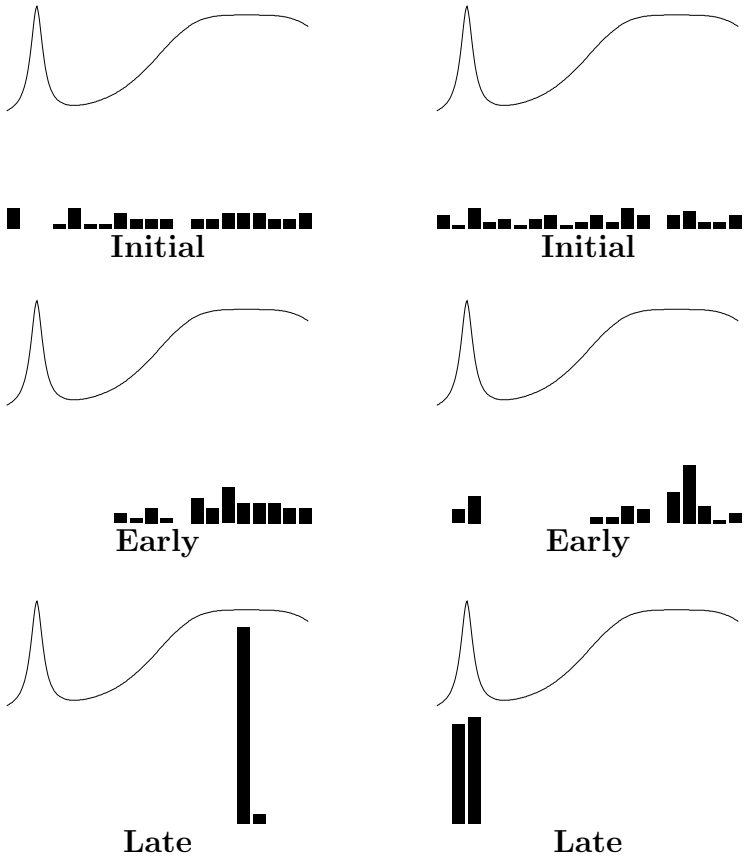


Figure 3.4: Graph of $f(x) = \frac{3.1}{1+(40x-4)^2} + \frac{3.0}{1+(3x-2.4)^4}$ in the same style as Figure 3.3 (The results of running two distinct initial populations are shown.)

In Figure 3.4, we see that when there is more than one optima the evolutionary algorithm can find different optima in different "runs." The difference between the left and right hand

diagrams are caused by the choice of initial population. The function

$$f(x) = \frac{3.1}{1 + (40x - 4)^2} + \frac{3.0}{1 + (3x - 2.4)^4} \tag{3.8}$$

has a pair of maxima. The left hand optimum is slightly higher, while the right hand optimum is much wider and hence easier to find. An interesting question is the probability of finding each of the optimum.

Experiment 3.6 *Take the code from Experiment 3.1 and modify it to use roulette selection and random replacement with an elite of size 2, to run without crossover (on only one variable) and to use uniform mutation with size 0.01. Use this code with a population size of 50 to optimize Function 3.8 on the interval* $[0, 1]$ *and report how many times the algorithm "found" each of the two optima. In your write up, be sure to give a clear definition of what it means for the population to have "found" the optima.*

This experiment gives us a way of measuring the way the "converged" (final) population of an evolutionary algorithm samples between two optima. It is clear that if we have a large number of distinct optima floating around, the evolutionary algorithm will have some sort of probability distribution (see Appendix A) on the optima of the function. This means that modifying the way we sample to produce the population may help. Let's try it.

Experiment 3.7 *Repeat Experiment 3.6 save that before evaluating the fitness of a point* x*, replace* x *with* $g(x)$ *where*

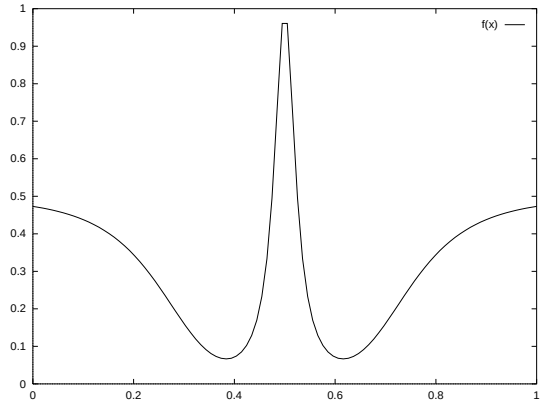
$$g(x) = \frac{x}{6 - 5x}.$$

Again: do everything as before, except that the fitness of a population member x *is* $f(g(x))$ *instead of* $f(x)$*. For each run, report* x*,* $g(x)$ *and* $f(g(x))$ *for the best population member. The* $g(x)$ *values are the ones used to judge which of the two maxima the population has converged to. In your write up, state the effect of the modification on the chance of locating the left hand optimum.*

We will return to the issue of fitness landscapes many times in the subsequent text. The idea of fitness landscapes is helpful for speculating on the behavior of evolutionary algorithms, but should never be taken as conclusive unless experiments or mathematical proof back the intuition generated. As the dimension of a problem increases, the quality of human intuition degrades. If we cease to consider functions of real variables, as we will in a majority of subsequent chapters, the intuition generated by the nice smooth graphs presented in this section will be even less helpful.

Problems

Problem 3.10 *Prove that the function $g(x)$ in Experiment 3.7 leaves population members that start in the range $[0, 1]$ in that range.*



$f(x)=\frac{2}{1+(40x-20)^2}-\frac{1}{1+(4x-2)^4}$

Problem 3.11 *Examine the function above. This function is* deceptive *in the sense that, unlike the Fake Bell Curve, there are many points from which uphill is away from the global maximum. Compute the probability an algorithm that picks a point uniformly at random in the interval $[0, 1]$ and then heads uphill will find the true optima of this function. Hint: this involves computing the position of the two minima, which you may do numerically if you wish.*

Problem 3.12 *Find an explicit formula for some $f(x, y)$ that generalizes the function given in Problem 3.11 to two dimensions. Having done this, compute the probability of successful hill climbing for that function for points chosen uniformly at random in the unit square. Will the problem get worse as the dimension increases?*

Problem 3.13 *Explain, analytically, what the function $g(x)$ in Experiment 3.7 does. Hint: start by graphing it.*

Problem 3.14 *Assuming that everything worked as intended, Experiment 3.7 modified the evolutionary algorithm to find the narrower but taller optimum more often. The function $g(x)$ is, in some sense, "tailored" to Function 3.8. Examine, for positive α, the functions*

$$g_\alpha(x) = \frac{x}{(\alpha + 1) - \alpha x}, \tag{3.9}$$

and,

$$h_\alpha(x) = \frac{(\alpha + 1)x}{1 + \alpha x}. \tag{3.10}$$

Suppose we are going to run an evolutionary algorithm, like the one in Experiment 3.8, a large number of times. Using the functions $g_\alpha(x)$ and $h_\alpha(x)$, come up with a scheme for running a different sampling of the space each time. Explain what roles the functions play. Assume that we have no idea where the optima are in the functions we are trying to optimize. You may want to give your answer as pseudo-code.

Problem 3.15 *Prove the functions $g_\alpha(x)$ and $h_\alpha(x)$ from Problem 3.14 have the following properties on the interval $[0, 1]$.*

(i) $g_\alpha(h_\alpha(x)) = h_\alpha(g_\alpha(x)) = x$.

(ii) The functions are monotone for all choices of α. (A function $q(x)$ is *monotone* if $a < b$ implies that $q(a) < q(b)$.)

(iii) Both functions take on every value in the range $[0, 1]$ exactly once each.

Problem 3.16 *Generalize the scheme you worked out in Problem 3.14 to multivariate functions. Suggestion: if we view $g_\alpha(x)$ and $h_\alpha(x)$ as* distortions*, it may be good to have different distortions for each dimension.*

Problem 3.17 Essay. *Speculate as to the effect of changing the population size on Experiment 3.6. Advanced students should actually redo the experiment with different populations sizes to support their speculation.*

Problem 3.18 Essay. *Speculate as to the effect of changing the mutation size on Experiment 3.6. Advanced students should actually redo the experiment with different populations sizes to support their speculation.*

Problem 3.19 Essay. *In Section 2.5, the Royal Road function is described. What is the fitness landscape for the evolutionary algorithm described in Experiment 2.10?*

3.3 Niche Specialization

In this section we will tinker with the fitness function of our real function optimizer to try and enhance performance. The idea of *niche specialization* for use in evolutionary algorithms was proposed by David Goldberg[14]. It is inspired by a biological notion with the same name, discussed in Section 1.2. The basic idea is simple; reduce or divide the fitness of a member of an evolving population by a function of the number of other essentially similar members of the population. In order to do niche specialization one needs to create a similarity measure. In real function optimization at least two obvious similarity measures are available. The population we operate on in our evolutionary algorithm for optimizing real functions is just

a collection of points in $\mathbb{R}^n$ and so proximity in $\mathbb{R}^n$ is one possible similarity measure. Call this *domain niche specialization*. Let r be the *similarity radius* and define a new fitness measure as follows. Where before we used the value of the function f being optimized as a fitness function, we will now take the value of the function divided by a penalty based on the number of members of the population nearby. Let m be the number of members of the population at distance r or less from a particular population member v. Let $q(m)$ be the *penalty function* and set

$$Fitness(v) := f(v)/q(m). \tag{3.11}$$

The effect of this will be to take optima and make them less attractive as more creatures find them. Since we are going to use it quite a lot the function $q(m)$ should be inexpensive to compute but also needs to avoid throwing creatures out of optima before they can explore them and find their true depth. One very easy to compute function is $q(m) = m$, but this function may well be far too harsh a penalty to the fitness. One might try a simple modification of the identity function like

$$q(m) = \begin{cases} 1 & m \leq 4 \\ m/4 & m > 4 \end{cases} . \tag{3.12}$$

The next experiment tests domain niche specialization on a function with three modes.

Experiment 3.8 *Recall that $\mathcal{B}_n$ is the Fake Bell Curve, Equation 2.1. Use the following function with three modes*

$$f(x, y, z) = \mathcal{B}_3(x, y, z) + 2 \cdot \mathcal{B}_3(x - 2, y - 2, z - 2) + 3 \cdot \mathcal{B}_3(x - 4, y - 4, z - 4)$$

and then modify the software from Experiment 3.1 to test niche specialization on $f(x, y, z)$ as follows. Use an initial population of 60 creatures all of the form $(0, 0, 0)$. Test the time for a creature to get within distance 0.05 of $(4, 4, 4)$ in real time *and in generations. Compute the mean and standard deviation of these times over 50 runs in each of 3 sets of trials, as follows. The first set of trials should be the basic real function optimizer. The second should be one with domain niche specialization with similarity radius 0.05 and $q(m) = m$. The third should be as the second save that $q(m) = \sqrt{m}$. Recall that $m \geq 1$ because the creature is close to itself.*

Since the computation of m and $q(m)$ are potentially quite expensive, the measurement of real time taken is very important, a more objective measure of efficiency than the generations or crossovers to solution. Notice that in Experiment 3.8 we place the initial population in a very bad place, inside a local optimum and with another local optimum between it and the unique global optimum. This is very different from having an initial population uniformly placed throughout the domain space.

Experiment 3.9 *Redo Experiment 3.8 with a random initial population placed uniformly on the domain $-1 \leq x, y, z \leq 5$. Emphasize the differences in performance resulting from initial population placement in your write up.*

The similarity measure used in domain niche specialization seems to assume that the optima are zero-dimensional, that is to say that all directions away from a local optimum result in points with lower fitness, as measured by the function being optimized. If we had functions that attained their optimum on some non-point region, they might prove a challenge for domain niche specialization. Instead of forcing the creatures to move away from an optimum and find new optima the niche specialization might well force them into different parts of the "same" local optimum. In Problem 3.4, we asked you to construct a function that had this property and in fact Equation 3.7, the Sombrero Function, is just such a function. All but one of its local optima are $(n-1)$-dimensional spheres centered on the origin. A problem with the Sombrero Function is that all the local optima are global optima but we can get around this by combining the Sombrero Function with the Fake Bell Curve. Examine the following pair of equations.

$$f(x_1, x_2, \ldots, x_n) = \frac{\cos\left(\sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}\right)}{x_1^2 + x_2^2 + \cdots x_n^2 + 1}. \tag{3.13}$$

$$f(x_1, x_2, \ldots, x_n) = \frac{\cos\left(\sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}\right)}{\left(4 - \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}\right)^2 + 1}. \tag{3.14}$$

The first, Equation 3.13 is a Sombrero Function multiplied by a Fake Bell Curve so that there is a single global optimum at the origin and sphere shaped local optima at each radius $2\pi k$ from the origin. The second, Equation 3.14 is a Sombrero Function multiplied by a Fake Bell Curve that has been modified to have a single spherical global optimum at a radius of 4 from the origin, placing the global optimum of the function in a spherical locus at a distance of nearly 4 from the origin. Part of the graphs of these functions is shown in Figure 3.5.

Experiment 3.10 *Redo Experiment 3.9 on the Equations 3.13 and 3.14. If you can, do the experiment in two dimensions and dynamically plot the location of each population member. Do the creatures spread out along the spherical local optimum? Given that we have crossover available, is that a bad thing? In any case, check the mean and standard deviation of time-to-solution to some part of the global optimum in 2, 3, and 4 dimensions. Use a tolerance of 0.05 for stopping. As testing tolerance is harder with non-point optima, carefully document how you test the stopping condition.*

The second sort of similarity measure we want to examine in this section is based on the position a creature codes for in the function's range space. We retain the similarity radius r

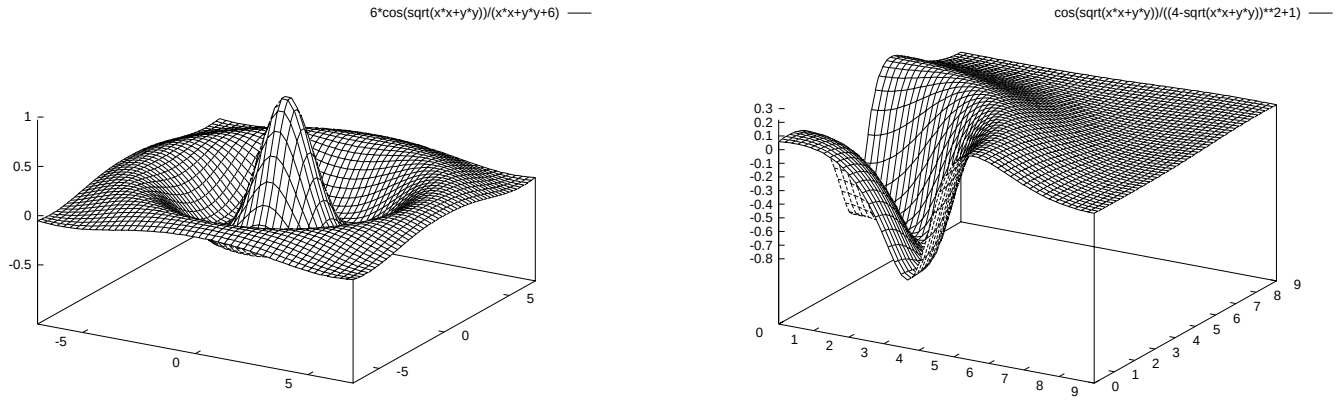


Figure 3.5: Graphs of Equations 3.13 and 3.14

from the preceding discussion as well as the penalty function $q(m)$ but change the method of computing m, the number of essentially similar creatures. If f is the function we are optimizing then let m for a creature y be the number of creatures x so that $|f(x)-f(y)| \leq r$. In other words we are just computing the number of creatures that have found roughly the same functional value. We term niche specialization on this new similarity measure, *range niche specialization*. Range niche specialization avoids some of the potential pitfalls of domain niche specialization. It could well compensate for the funny spherical local optima that come from the Sombrero Function. Imagine a function was jumping around quite a lot in a very small region. Range niche specialization wouldn't tend to drive creatures out of such a region before it was explored. On the other hand, if there is a local optimum with a steep slope leading into it, then range niche specialization might create a population living up and down the sides of the optimum.

Experiment 3.11 *Redo Experiment 3.10 with range niche specialization. Does it help, hurt, or make very little difference?*

Since the two types of niche specialization we have discussed have different strengths and weaknesses and both also differ from the basic algorithm it might be possible to use the various types of niche specialization intermittently. Given that they have nontrivial computational costs, using them intermittently is a computationally attractive option.

Experiment 3.12 *Pick a function that has been difficult for both sorts of niche specialization and modify your optimizing software as follows. Run the basic real function optimizer for 20 generations and then one or the other sort of niche specialization, for 10 generations, and repeat. Also try alternating the types of niche specialization. Do you observe*

any improvement? Remember to measure time-to-solution with both a "stop watch" and in generations. As the directions given for this experiment are fuzzy and imprecise, you should give a detailed write up of exactly what you did and why in your write up.

Range niche specialization may be of more use in situations where the functions being optimized are not continuous. In the next section we will see some examples of functions that are constant where they are not discontinuous. These functions may be a more fruitful area for use of niche specialization.

Problems

Problem 3.20 *Suppose we are optimizing the Fake Bell Curve in two dimensions with an 8 member population. If the creatures' genes are as shown in the following table, compute the modified fitness of each creature for domain niche specialization with similarity radius* $r = 0.1$ *and penalty function* $q(m) = \sqrt{m}$*. Do you think the use of niche specialization will help in this instance?*

x	y
0.042	*0.043*
0.047	*0.048*
0.051	*0.066*
0.121	*0.136*
0.077	*0.081*
0.166	*0.135*
0.042	*0.055*
0.211	*0.056*

Problem 3.21 *Do Problem 3.20 over but for range niche specialization with similarity radius* $r = 0.01$*.*

Problem 3.22 *In Experiment 3.8 it was asserted that the function being optimized,*

$$f(x, y, z) = \mathcal{B}_3(x, y, z) + 2 \cdot \mathcal{B}_3(x - 2, y - 2, z - 2) + 3 \cdot \mathcal{B}_3(x - 4, y - 4, z - 4),$$

had 3 modes. In Problem 2.25 we saw that if Fake Bell Curves are put too close together the modes blend. Derive from $f(x, y, z)$ *a single variable function that will have the same number of modes as* f *and graph it to verify* f *has 3 modes.*

Problem 3.23 Essay. *Construct a function that you think will be more easily optimized by a basic real function optimizer than by that same optimizer modified by domain niche specialization. Discuss how you construct the function and give your reasons for thinking that domain niche specialization might run afoul of your function. Consult your instructor as to the degree of precision he demands in specifying a function. Advanced students should support their conclusions experimentally.*

Problem 3.24 Essay. *Do problem 3.23 for range niche specialization.*

Problem 3.25 *Carefully graph*

$$f(x,y) = \frac{C}{\left(d - \sqrt{x^2+y^2}\right)^2 + 1}$$

for $C = 1$ and $d = 3$ and discuss its relationship to Equation 3.14 and the Fake Bell Curve, Equation 2.1. This function is called the Crater *Function.*

Problem 3.26 Essay. *Assume that we are using the stopping condition that says "stop if there has been no change in the gene of the most fit creature for 20 generations." The Crater Function, Problem 3.25, creates an annoying optimum that takes on its maximum value at an infinite number of points. Since all these points are true global optima, this creates a problem for our stopping criterion. Discuss what would happen to the optimization process if we added a Fake Bell Curve or other bounded, easily-computed curve to the Crater Function. Assume that the curve added has a very small maximum compared to the maximum of the curve actually being optimized. In particular would adding this other "bias" function ease the problem afflicting the stopping criterion? What if the bias function was instead composed of a fixed set of random noise of small amplitude? Remember to discuss not only the effects on the population structure but the computational costs.*

Problem 3.27 Essay. *Suppose we are optimizing real functions with either domain or range niche specialization. Try to characterize the set of continuous functions that you think would be more likely to be optimized more efficiently under each of these two specialization techniques.*

Problem 3.28 *Explain why the following are desirable properties of a penalty function $q(m)$, $m \varepsilon \{1, 2, 3, \ldots\}$.*

(i) $q(1) = 1$,
(ii) $q(m) \geq 1$ for all m, and
(iii) $q(m+1) \geq q(m)$ for all m.

Problem 3.29 Essay. *One potential problem with niche specialization is that it may drive parts of a population out of a niche before it is explored. Discuss how the choice of the penalty function can be used to combat this problem without removing all the benefits of niche specialization, e.g. by taking $q(m) = 1$. Give at least three penalty functions and discuss their advantages and disadvantages both in terms of the population structures they encourage and their computational cost. The* unpenalized local population size *is the size of a population that can exist within a single optimum smaller than the similarity radius without any fitness decrease. For each penalty function you discuss, compute the unpenalized local population size. How could this size be zero and why would that be bad?*

3.4 Path Length, an Extended Example

In this section, we will work with minimizing the length or cost of a path (like a Garden Tour in which you are given a list of addresses that you are to visit in order). This is a rich class of minimization problems with some interesting properties. An interesting thing about this class of problems is that there are two different "natural" fitness functions that produce different fitness landscapes, but which both have a global optimum which solves the problem. We note that an algorithm called *dynamic programming*, which appears in the robotics literature, can be used to solve the type of problems treated in the section far more efficiently than an evolutionary algorithm: we are using path length optimization to illuminate some of the features of evolutionary computation.

A *path* with n points is a sequence $\mathcal{P} = \{(x_0, x_1), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ of points to be traversed in order.

The *length* of a path is the sum:

$$Len(\mathcal{P}) = \sum_{k=0}^{n-2} \sqrt{(x_k - x_{k+1})^2 + (y_k - y_{k+1})^2} \tag{3.15}$$

Our paths will live in the unit square:

$$\mathcal{U} = \{(x, y) : 0 \leq x, y \leq 1\} \tag{3.16}$$

For the nonce we will anchor our paths at $(0, 0)$ and $(1, 1)$ by insisting that $(x_0, y_0) = (0, 0)$ and $(x_{n-1}, y_{n-1}) = (1, 1)$.

An optimal solution for minimizing the length of a path is to place all of the points in the path in order on the diagonal. There are a large number of optimal solutions, since moving a point anywhere between its adjacent points on the diagonal does not change the length of the path. This means that if we were to use Equation 3.15 as a fitness function on a population of paths there would be a nontrivial flat space at the bottom (remember, we are minimizing) of the fitness landscape. When there is only one point, the path length is a function of the one moving point (illustrated in Figure 3.6). Note that the flat space is a line corresponding to placing the movable point on the line segment joining $(0, 0)$ with $(1, 1)$.

For those familiar with the notion of dimension, the flat space, while much larger than the point optima we have seen thus far, is a lower dimensional subset of the fitness landscape. In this case, the fitness landscape is a two-dimensional surface while the optimal solutions all lie on a one-dimensional line segment.

Experiment 3.13 *Write or obtain software for a real function optimizer evolutionary algorithm that can optimize the length of a path of the sort described above. The function to be optimized is Equation 3.15 with the independent variables being the x and y positions of*

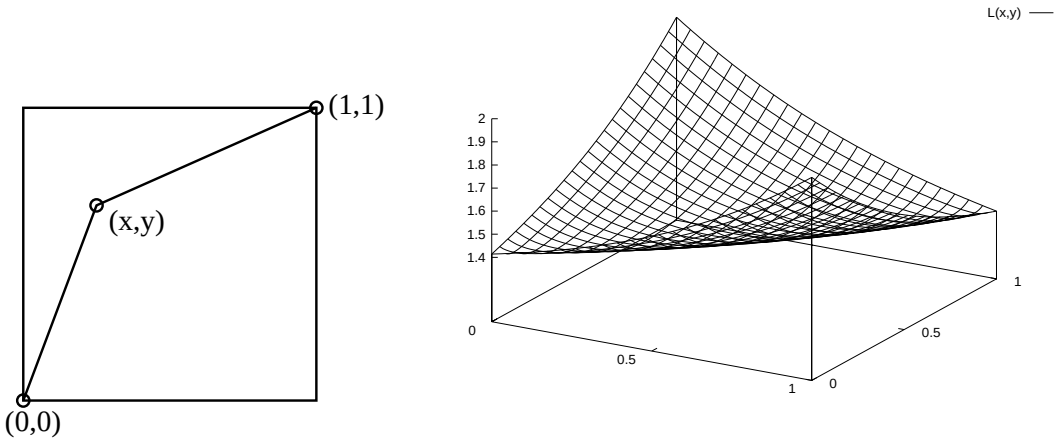


Figure 3.6: A path with one non-anchored point and the graph of the length $L(x, y)$ of that path as a function of the unanchored point (Note the trough-shaped set of minima along the diagonal from $(0, 0)$ to $(1, 1)$.)

the points other than the anchor points. Make the number of points $n \geq 3$ a parameter that you can change easily.

Your algorithm should operate on a population of 100 paths. Use roulette selection and random replacement with an elite of 2 paths. Use two point crossover and uniform two point mutation with a size of 0.1. Run the algorithm 50 times for $n = 5$ points (the 5 includes the 2 anchors). Save the best (shortest) path from each run and plot the best, worst, and 3 others of these "best of run" paths. Run until the path is within 5% of the true minimum length or for 1000 generations, whichever is shortest.

Now repeat the experiment, but instead use a mutation operator that mutates both coordinates of one point. Plot the same collection of paths and, in your write up, comment on the effect the new mutation operator had.

The point of Experiment 3.13 is to compare two mutation operators, one of which "knows something" about the problem. The gene used in Experiment 3.13 has more obvious structure than the ones we've used to far. It is organized into pairs of point. We said nothing in the experiment about how to organize the independent variables. There are 720 possible orders for them, but, in some sense, only a few natural orders. We could group the x and y coordinates into their own blocks or we could group the pairs of variables giving the position of each point together. The next experiment compares uniform and Gaussian mutations.

Experiment 3.14 *Modify the software from Experiment 3.13 to use Gaussian mutations in place of uniform ones. Compute a standard deviation for the Gaussian mutation that is the*

same as that of the uniform mutation used before. In your lab write up, compare uniform and Gaussian mutations for this problem.

So far we have tinkered with the action of mutation on points and with the shape of mutation. The next experiment suggests another potential avenue for improvement, but you'll have to hunt around for a way to make it work.

Experiment 3.15 *Modify the software from Experiment 3.13 to let the size of the mutations decrease as a function of the number of generations. Repeat the experiment with at least two different schemes for decreasing the mutation size as time passes. In your write up, document if this mutation size shrinking helped and speculate as to why or why not.*

We are now ready to look at the second fitness function mentioned at the beginning of the section. In Equation 3.15 we compute the length of the line segments making up the path $\mathcal{P}$. The square root function, which appears numerous times in this computation, is a monotone function and so removing it will still permit us to minimize path length in some fashion. (Think about why this is desirable.)

$$Len2(\mathcal{P}) = \sum_{k=0}^{n-2} (x_k - x_{k+1})^2 + (y_k - y_{k+1})^2 \tag{3.17}$$

Experiment 3.16 *Modify the software from Experiment 3.13 to use Equation 3.17 as a fitness function and re-perform the experiment using the mutation operator acting on both coordinates of a point. Compare both the time required to get within 5% of optimum and the character of the "best of run" paths.*

Having two different fitness functions for the same problem permits us to try and experiment in avoiding traps in the fitness landscape. As we will see in the problems, the fitness landscape for either function does contain traps, in the form of tortuous (mutational) paths rather than direct paths to the optima. If we view the population as moving across the fitness landscape, then one function may have a trap where the other does not. Thus, alternating between the fitness functions may permit more rapid convergence that the use of either fitness function.

Experiment 3.17 *Modify the software from Experiment 3.13 to use either Equation 3.15 or Equation 3.17 as a fitness function and re-perform the experiment using the mutation operator acting on both coordinates of a point. Try both alternating the use of the two fitness functions every other generation and every fifth generation. In your write up, compare the time required for each version of the algorithm to get within 5% of optimum.*

Problems

Problem 3.30 *In Figure 3.6, there is a graph of Equation 3.15 for the case of a path with one mobile point. Copy this graph and make a graph for both Equation 3.17 and for the difference of Equation 3.15 and Equation 3.17. Comment on the difference between their minima.*

Problem 3.31 *Prove that all optima of Equation 3.17 are also optima of Equation 3.15, and give an example that shows the reverse is not true.*

Problem 3.32 *Construct an example of a path with 5 total points (including the 2 anchored points) and a mutation of both coordinates of a point so that (i) the mutation makes the fitness of the path worse, and (ii) the mutation moves the point to a place where it would be in an optimal solution.*

Problem 3.33 *Construct an example of a non-optimal path and a mutation of both coordinates of a point that does not change the fitness of the path.*

Problem 3.34 Essay. *Reread Problem 3.15. Does the use of the functions $g_\alpha(x)$ and $h_\alpha(x)$ remove traps in the fitness landscape, as we hope the alternation of fitness functions will do in Experiment 3.17 or does it do something else? Explain.*

Problem 3.35 Essay. *Suppose that instead of alternating fitness functions, as in Experiment 3.17 we instead added a small random number to the fitness evaluation of a path. First of all, would this help us out of traps, and second what other benefits or problems might this cause? Give the design of an experiment to test this hypothesis.*

3.5 Optimizing a Discrete-Valued Function: Crossing Numbers

The third function in DeJong's test suite, Equation 3.4, has the property that it is discontinuous everywhere it is not constant. When looking at the definition of this function, one may be inspired to ask "do functions like that ever come up in practice?" The answer is a resounding "yes!" and in this section we will define and optimize a class of such functions.

Our motivation for this class of functions will be circuit layout. Circuit layout is the configuration of the components of an electric circuit on a printed circuit board. Efficiency will be measured by the number of jumpers needed. A printed circuit board is a nonconducting board with the wiring diagram of a circuit printed on the board in copper. The parts are soldered into holes punched inside the copper coated regions. When laying out the circuit

board, it may be that two connections must cross one another on the two-dimensional surface of the board without any electrical connection. When this happens we need a jumper. Where the crossing happens, one of the two connections in the copper to be printed on the board is broken, and an insulated wire is soldered in to bridge the gap. It turns out that some circuit layouts require far fewer bridges than others, so locating such good layouts is desirable.

Since representing actual circuit diagrams in the computer would be a little tricky, we will formalize circuit layouts as drawings of *combinatorial graphs*. The elementary theory of such graphs is discussed in Appendix C. Briefly, a graph is a collection of points (called *vertices*) and lines joining some pairs of points (called *edges*). A *drawing* of a graph is simply a placement of the vertices of the graph into the plane together with a representation of the edges as simple curves joining their endpoints. A drawing is said to be *rectilinear* if all the edges are line segments. The *crossing number* of a drawing of a graph is the number of pairs of edges that intersect (other than at their endpoints). The *crossing number* of a graph is the minimum crossing number attained by any drawing. The *rectilinear crossing number* of a graph is the minimum crossing number of any rectilinear drawing of that graph. It is known that for some graphs the rectilinear crossing number is strictly larger than the crossing number. A theorem of graph theory says that any graph with a crossing number of zero also has a rectilinear crossing number of zero. Graphs with a crossing number of zero are said to be *planar*.

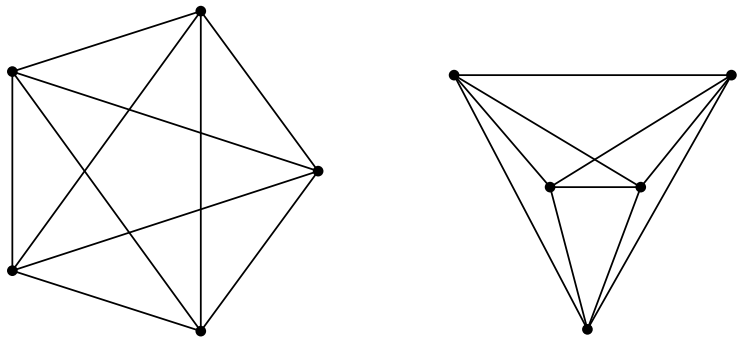


Figure 3.7: The complete graph K_5 drawn with five crossings and with one crossing

The *complete graph on n vertices*, denoted K_n, is the graph with all possible edges. A small amount of work will convince the reader that K_n is planar only when $n \leq 4$. In Figure 3.7, we show a standard presentation of K_5 and a rectilinear drawing of K_5 that has crossing number one.

At the time of this writing, the crossing number and rectilinear crossing number of K_n are only known for $n \leq 9$. The known values for rectilinear crossing numbers of K_n are given

in Figure 3.8.

n	2	3	4	5	6	7	8	9
Crossing Number	0	0	0	1	3	9	19	36

Figure 3.8: Known rectilinear crossing numbers for K_n

The first step in designing software to approximate the rectilinear crossing number is to figure out when two lines cross. The specific problem is, given the endpoints of two line segments, how do you decide if those line segments cross other than at their endpoints? First, if two lines emanate from the same point then they do not contribute to the crossing number and need not be compared. Such checks should be done by your algorithm before any question of lines intersecting is treated. Suppose that we have two line segments with endpoints $(x_1, y_1), (x_2, y_2)$ and $(r_1, s_1), (r_2, s_2)$ respectively, with $x_1 \leq x_2$ and $r_1 \leq r_2$. Then, using simple algebra to compute the slopes m_1 and m_2 of the lines respectively, we find the x-coordinate of the intersection of the two lines is

$$x_i = \frac{m_1 x_1 - y_1 - m_2 r_1 + s_1}{m_1 - m_2} \tag{3.18}$$

and so we can check if $x_1 \leq x_i \leq x_2$ and $r_1 \leq x_i \leq r_2$.

There are some problems with this test; a Boolean predicate that computes the intersection of lines would want to check for $m_1 = m_2$ or $m_i = \infty, i = 1, 2$ and deal with them appropriately. It is also possible for things to get very confusing if 3 vertices are colinear. In the event that 3 vertices are colinear, treat the structure as a single intersection. The task of avoiding all these pitfalls is left for you in Problem 3.43. In addition, numerical precision issues bedevil anyone who wishes to compute if two line segments intersect in their interior. These issues are beyond the scope of this text and are the business of the field of *computational geometry*.

Assuming we can compute when two line segments intersect in their interior we are ready to modify any of the real function optimizers that we've done so far to estimate crossing numbers. The function we are optimizing will have $2n$ variables for an n vertex graph, where these variables are the x and y-coordinates of the vertices for the drawing of the graph. In the case of the complete graph, K_n, we will be looking at all possible pairs of line segments. In general, we will need an *adjacency matrix* for the graph as defined in Appendix C. The fitness function to minimize is: for each pair of edges in the graph which intersect but do not share a vertex, add one to the fitness function.

Experiment 3.18 *Write or obtain software for a real function optimizer that estimates the rectilinear crossing numbers of complete graphs. Test this optimizer on K_4, K_5, and K_6. Use tournament selection with tournament size 4, single point mutation with mutation size*

$\epsilon = 0.1$ *applied to each new creature, and two point crossover. Force your starting drawing to lie in the unit square but allow mutation to take the vertices outside the unit square if it wishes. Compute the mean and standard deviation of time-to-solution over 30 runs for each graph. Turn in the visually most pleasing drawings you obtain for* K_4, K_5, *and* K_6.

Experiment 3.19 *Modify Experiment 3.18 to use Gaussian point mutation with the same variance as in Experiment 3.18 and compare with the results from uniform mutation.*

Experiment 3.20 *Take the software from Experiment 3.18 and modify it to use domain and range niche specialization. Taking the known answer for* K_6, *compare the raw algorithm with both domain and range niche specialization. The comparison of time-to-solution should be made by comparing mean times with a one standard deviation error bar. Use a similarity radius of* 0.05 *for the domain niche specialization. Range niche specialization will consider creatures the same if they have the same fitness. For both sorts of niche specialization use the penalty function* $q(m)$ *given in Equation 3.12.*

Notice that in Experiment 3.20 the range niche specialization uses a similarity radius of zero. This is sensible because we are optimizing an integer-valued function. Zero isn't the only possible sensible value - but for real-valued functions it might never happen in practice. One corollary of the fitness function being integer-valued with real domain is that a great deal of information must be getting lost. In other words, there must be many, many genes that map to the same fitness value.

Experiment 3.21 *Modify the basic program from Experiment 3.18 so that the algorithm incorporates a (possibly hard-coded) adjacency matrix for a graph* G *and modify the algorithm so that the algorithm estimates the crossing number for* G. *Try your optimizer on 12 vertices connected in a cycle and on the graph with the following adjacency matrix.*

$$\begin{bmatrix}
0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0
\end{bmatrix}$$

Problems

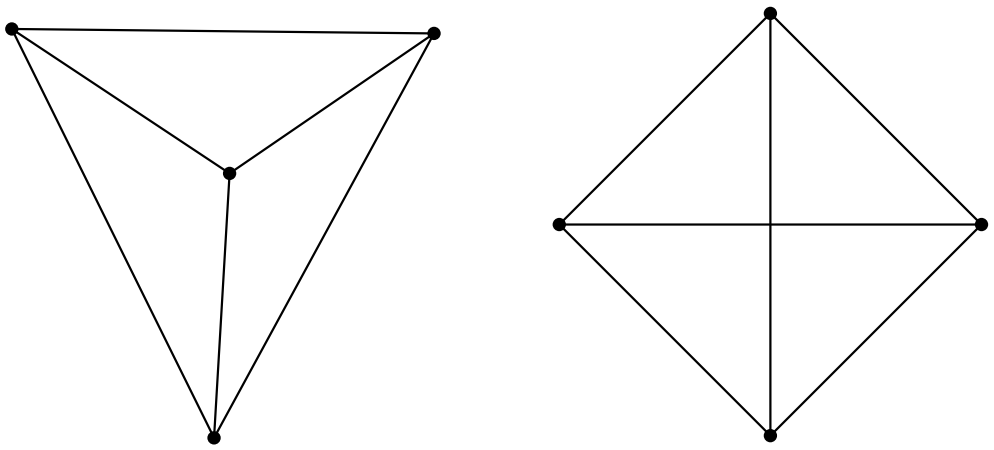
Problem 3.36 *Verify the formula given in Equation 3.18 showing your work clearly. Obtain a similar formula for the y-coordinate of the intersection.*

Problem 3.37 *Do the evolutionary algorithms discussed in this section place upper or lower bounds on crossing numbers? Justify your answer in a sentence or two.*

Problem 3.38 *What is the maximum crossing number possible for a rectilinear drawing of K_n? Advanced students should offer a mathematical proof of their answer.*

Problem 3.39 Essay. *Would you expect domain or range niche specialization to be more effective at improving the performance of an evolutionary algorithm that estimated rectilinear crossing number? Justify your answer with logic and with experimental evidence if it is available. You may wish to examine problem 3.40 before commencing.*

Problem 3.40 *Shown below are two drawings of K_4. Suppose we were going to add a vertex somewhere and connect it to all the other vertices so as to get a rectilinear drawing of K_5. First, give a proof that the crossing number of the new drawing depends only on in which region of the drawing the added vertex is placed. Second, redraw these pictures and place the resulting crossing number in each of the regions.*



Problem 3.41 *Problem 3.40 suggest a novel mutation operator for use in computing crossing numbers. Discuss it briefly and outline an algorithm that would implement such a mutation operator.*

Problem 3.42 *There are an infinite number of possible drawings of a graph in the plane. (Given a drawing, just move a vertex of the drawing any distance in any direction to get*

another drawing.) A little thought shows this isn't a useful definition of the word "different" when attempting to compute crossing number. Following problem 3.40, come up with an alternate definition of when two drawings are the same or different and explain why your definition is useful. You may want to glance at a book on topology to get some of your terms. Consider the role of vertices, edges, and intersections of edges in your definition of "different." Is it easy to compute when two graphs are different under your definition?

Problem 3.43 *Give the code for a function that accepts two line segments and returns true if they intersect in their interior. Intersection at the endpoints is returned as false. Make sure your code deals well with parallel lines.*

Problem 3.44 *Write a function such that, given the number n of vertices in a matrix, two real arrays x and y that specify the vertex coordinates, and a matrix A with $a_{i,j} = 1$ (meaning that vertices i and j are joined by an edge), the function returns the rectilinear crossing number of a graph. It should use the code from problem 3.43.*

Problem 3.45 Essay. *In a practical implementation of an evolutionary algorithm to estimate rectilinear crossing number, there will be problems with almost parallel and almost vertical lines. Discuss hacks to avoid having numerical precision problems. You may wish to consider modifying the fitness function to penalize certain slopes or pairs of slopes, moving vertices periodically to avoid these sorts of problems, or moving vertices exactly when a vertex is apparently causing such problems. Be sure to discuss the computational cost of such changes as well as their efficiency in preventing numerical precision problems. Would putting the vertices of the graph onto a discrete grid help?*

Problem 3.46 *In this section, we develop evolutionary algorithms to estimate rectilinear crossing numbers of graphs. Working with the true crossing number would require that we somehow manipulate fully general curves in the plane, which is very hard. Suppose that instead of representing edges as line segments, we represent them as multiple line segments. This would bring the rectilinear crossing number closer to the true crossing number. Find a graphical construction that transforms a graph G into a graph G' by drawing the edges of G using k consecutive line segments, so that the rectilinear crossing number of G' is an estimate of the crossing number of G.*

Problem 3.47 *Suppose we have a drawing of a graph in which one vertex lies* on *a line segment joining two others. In the formal graph theory that computes crossing numbers, this is viewed as a pathological case, repaired by making a tiny change in the position of the vertex causing the problem. In this chapter, this is counted as a crossing in order to get on with things. Our motivating example was a circuit board. Argue from physical reality that a vertex in the middle of an edge should not be counted as a crossing at all. Give an example.*

Problem 3.48 *If you have software that computes the crossing numbers of drawings experimentally, estimate and graph the crossing number of randomly drawn k-gons for $k = 4, 5, \ldots, 12$. Also estimate the probability of getting a correct solution in the initial population for the 12-cycle half of experiment 3.21.*