Chapter 2

Designing Simple Evolutionary Algorithms

The purpose of this chapter is to show you how an evolutionary algorithm works and to teach you how to design your own. We start simple, by evolving binary character strings. Evolutionary algorithms are a synthesis of several techniques: genetic algorithms, evolutionary programming, evolutionary strategies, and genetic programming. In this chapter, there is a bias towards genetic algorithms [14], because they were designed around the manipulation of binary character strings. The terminology used in this book comes from many sources; arbitrary choices were necessary when several terms exist for the same concept.

Create an initial population.
Evaluate the fitness of the population.
Repeat
    Select pairs from the population to be parents, with a fitness bias.
    Copy the parents to make children.
    Perform crossover on the children (optional).
    Mutate the resulting children (probabilistic).
    Place the children in the population.
    Evaluate the fitness of the children.
Until Done.

Figure 2.1: A simple evolutionary algorithm

Figure 2.1 is an outline for a simple evolutionary algorithm. It's more complex than it seems at first glance. There are five important decisions that factor into the design of the algorithm.

What data structure will you use? In the string evolver and real function optimizer in Section 1.2, for example, the data structures were a string and an array of real numbers, respectively. This data structure is often termed the *gene* of the evolutionary algorithm. You must also decide how many genes will be in the evolving population.

What fitness function will you use? A fitness function maps the genes onto some ordered set, such as the integers or the real numbers. For the string evolver, the fitness function had its range in the natural numbers; the fitness of a given string was the number of positions at which it agreed with a reference string. For the real function optimizer, the fitness function was simply the function being optimized (when maximizing) or its negative (when minimizing).

What crossover and mutation operators will you use? Crossover operators map pairs of genes onto pairs of genes; they simulate sexual reproduction. Mutation operators make small changes in a gene. Taken together, these are called *variation operators*.

How will you select parents from the population, and how will you insert children into the population? The only requirement is that the selection method be biased toward "better" organisms. There are many different ways to do this.

What termination condition will end your algorithm? This could be a fixed number of trials or when a solution is found.

Our prototype evolutionary algorithm will be the string evolver (see Definition 1.2 and Problem 1.11). Our data structure will be a string of characters, and our fitness function will be the number of agreements with a fixed reference string. We will experiment with different variation operators and different ways of picking parents and inserting children.

2.1   Models of Evolution

Definition 2.1 *The method of picking parents and the method of inserting children back into the population, taken together, are called the* **model of evolution** *used by an evolutionary algorithm.*

The model of evolution used in Problem 1.11 is called *single tournament selection*. In single tournament selection, the population is shuffled randomly and divided into small groups, usually of size 4. The two most fit individuals in each small group are chosen to be parents. These parent strings are crossed over and the results possibly mutated to provide two children that replace the two least fit members of the small group.

Single tournament selection has two advantages. First, for small groups of size n, the best $n-2$ creatures in the population are guaranteed to survive. This ensures that the maximum

fitness of a group (with a deterministic fitness function) cannot decline as evolution proceeds. Second, no matter how fit a creature is compared to the rest of the population, it can have at most one child in each generation. This prevents the premature loss of diversity in a population that can occur when a single parent has a large number of children, a perennial problem in evolutionary algorithms of all sorts. When a creature that is relatively fit in the initial population dominates the early evolved population, it can prevent the discovery of better creatures by leading the population into a local optimum.

Definition 2.2 *A* **global optimum** *is a point in the fitness space whose value exceeds that of any other value (or is exceeded by every other value, if we are minimizing). A* **local optimum** *is a point in the fitness space that has the property that no chain of mutations starting at that point can go up without first going down.*

Making an analogy to a mountain range, the global optimum might be thought of as the top of the highest mountain, while the local optima are the peaks of every mountain or foothill in the range. Even rocks will have associated local optima at their high points. Notice, the global optimum is one of the local optima. Also, notice there may be more than one global optimum, if two mountains tie for highest.

When the members of a population with the highest fitness are guaranteed to survive in an evolutionary algorithm, that algorithm is said to exhibit *elitism*. Those members of the population guaranteed to survive are called the *elite*. Elitism guarantees that a population with a fixed fitness function cannot slip back to a smaller maximum fitness in later generations, but it also causes the current elite to be more likely to have more children in the future causing their genes to dominate the population. Such domination can impair search of the space of genes, because the current elite may not contain the genes needed for the best possible creatures. A good compromise is to have a small elite. Single tournament selection has an elite of size 2. Half the population survives, but only two creatures, the two most fit, must survive. Other creatures survive only if they have the good luck to be put in a group with creatures less fit than they.

In single tournament selection, the selection of parents and the method for inserting children are wedded to one another by the picking of the small groups. This need not be the case; in fact, it is usually not the case. There are several other methods of selecting parents.

In *double tournament selection*, with tournament size n, you pick a group of n creatures and take the single most fit one as a parent, repeating the process with a new group of n creatures to get a second parent. Double tournament selection may also be done *with replacement* (the *same* parent can be picked twice) or *without replacement* (the same parent cannot be picked twice, i.e., the first parent is excluded during the selection of the second parent).

Roulette wheel selection, also called called roulette selection, chooses parents in direct proportion to their fitness. If creature i has fitness f_i then the probability of being picked as a parent is $\frac{f_i}{F}$, where F is the sum of the fitness values of the entire population.

Rank selection works in a fashion similar to roulette wheel selection except that the creatures are ordered by fitness and then selected by their rank instead of their fitness. If creature i has rank f_i, then the probability of being picked as a parent is $\frac{f_i}{F}$, where F is the sum of the ranks of the entire population. Note: the *least* fit creature is given a rank of 1 so as to give it the smallest chance of being picked.

In Figure 2.2, we compare the probabilities for rank and roulette selection. If there is a strong fitness gradient, then roulette wheel selection gives a stronger fitness bias than rank selection and, hence, tends to take the population to a nearly uniform type faster. The utility of faster fixation depends on the problem under consideration.

Creature #	Fitness	Rank	P(chosen) Roulette	P(chosen) Rank
1	2.1	1	0.099	0.048
2	3.6	5	0.169	0.238
3	7.1	6	0.333	0.286
4	2.4	2	0.113	0.095
5	3.5	4	0.164	0.190
6	2.6	3	0.122	0.143

Figure 2.2: Differing probabilities for roulette and rank selection

A model of evolution also needs a child insertion method. There are several such methods. One is to place the children in the population at random, replacing anyone. This is called *random replacement*. If we select creatures to be replaced with a probability inversely proportional to their fitness, we are using *roulette wheel replacement* (also called roulette replacement). If we rank the creatures in the opposite order used in rank selection and then choose those to be replaced with probability proportional to their rank, we are using *rank replacement*. In another method, termed *absolute fitness replacement*, we replace the least fit members of the population with the children. Another possible method is to have children replace their parents only if they are more fit. In this method, called *locally elite replacement*, the two parents and their two children are examined, and the two most fit are put into the population in the slots occupied by the parents. In *random elite replacement*, each child is compared to a randomly selected member of the population and replaces it only if it is at least as good.

With all of the selection and replacement techniques described above, you must decide how many pairs of parents to select in each generation of your evolutionary algorithm. At one extreme, you select enough pairs of parents to replace your entire population; this is called a *generational* evolutionary algorithm. At the other extreme, a *steady state* evolutionary algorithm, you count each act of selecting parents and placing (or failing to place) the

children in the population as a "generation." Such single mating "generations" are usually called *mating events*.

Generational evolutionary algorithms were first to appear in the literature and are considered "standard." Steady state evolutionary algorithms are described very well by Reynolds [30] and were discovered independently by Syswerda [34] and Whitley [38].

Experiment 2.1 *Write or obtain software for a string evolver (defined in Section 1.2). For each of the listed models of evolution, do 100 trials. Use 20-character strings of printable ASCII characters and a 60-member population. To stay consistent with single tournament selection in number of crossover events, implement all other models of evolution so that they replace exactly half the population. This updating of half the population will constitute a generation. For this experiment, use the type of crossover used in the first part of Figure 1.3 and Problem 1.11, in which the children are copies of the parents with their gene loci swapped after a randomly generated crossover point. For mutation, change a single character in each new creature at random.*

 (i) *Single tournament selection with small groups of size 4*

 (ii) *Roulette selection and locally elite replacement*

 (iii) *Roulette selection and random replacement*

 (iv) *Roulette selection and absolute fitness replacement*

 (v) *Rank selection and locally elite replacement*

 (vi) *Rank selection and random replacement*

(vii) *Rank selection and absolute fitness replacement*

Write a few paragraphs explaining the results. Include the mean and standard deviation of the solution times (measured in generations) for each model of evolution. (A population is considered to have arrived at a "solution" when it contains one string that matches the reference string.) Compare your results with those of other students. Pay special attention to trials done by other students with identical models of evolution that give substantially different results.

Experiment 2.2 *Use the version of the code from Experiment 2.1 with roulette selection and random replacement. Compute the mean and standard deviation of time-to-solution of 100 trials in each of 5 identical populations in which you replace 1/5, 1/3, 1/2, 2/3, and 4/5 of the population in each generation. Measure time in generations and in number of crossovers; discuss which measure of time is more nearly a fair comparison of the different models of evolution.*

Experiment 2.3 *Starting with the code from Experiment 2.1, build a steady state evolutionary algorithm. For each of the following models of evolution, do 20 different runs. Give the mean and standard deviation of the number of mating events until a maximum fitness creature is located. Cut off the algorithm at 1,000,000 mating events, if no maximum fitness creature is located. Assume the double tournament selection is with replacement.*

(i) *Single tournament selection with tournament size 4.*

(ii) *Single tournament selection with tournament size 6.*

(iii) *Double tournament selection with tournament size 2.*

(iv) *Double tournament selection with tournament size 3.*

Problems

Problem 2.1 *Assume that we are running an evolutionary algorithm on a population of 12 creatures, numbered 1-12, with fitness values of 1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, and 34. Compute the expected number of children each of the 12 creatures will have for the following parent selection methods: (i) roulette selection, (ii) rank selection, and (iii) single tournament selection with tournament size 4. (The definition of* expected value *may be found in Appendix A.) Assume that both parents can be the same individual in the roulette and rank cases.*

Problem 2.2 *Repeat Problem 2.1 (i) and (ii), but assume that the parents must be distinct.*

Problem 2.3 *Compute the numbers that would appear in an additional column of Figure 2.2 for P(chosen) using single tournament selection with small groups of size 3.*

Problem 2.4 *Compute the numbers that would appear in an additional column of Figure 2.2 for P(chosen) using double tournament selection with small groups of size 4 and with replacement.*

Problem 2.5 *First, explain why the method of selecting parents, when separate from the method of placing children in the population, cannot have any effect on whether a model of evolution is elitist or not. Then, classify the following methods of placing children in the population as elitist or non-elitist. If it is possible for a method to be elitist or not depending on some other factor, e.g., fraction of population replaced, then say what that factor is and explain when the method in question is or is not elitist.*
(i) *random replacement,*
(ii) *absolute fitness replacement,*
(iii) *roulette wheel replacement,*

(iv) *rank replacement,*
(v) *locally elite replacement,*
(vi) *random elite replacement.*

Problem 2.6 Essay. *Aside from the fact that we already know the answer before we run the evolutionary algorithm, the problem being solved by a string evolver is very simple in the sense that all the positions in the creature's gene are independent. In other words, the degree to which a change at a particular location in the gene is helpful, unhelpful, or detrimental depends in no way on the value of the gene in other locations. Given that this is so, which of the possible models of evolution that you could build from the various parent selection and child placement methods, including single tournament selection, would you expect to work best and worst? Advanced students should support their conclusions with experimental data.*

Problem 2.7 *Give a sketch or outline of an evolutionary algorithm and a problem that together have the property that fitness in one genetic loci* can *be bought at the expense of fitness in another genetic loci.*

Problem 2.8 *Invent a model of evolution not described in this section that you think will be more efficient than any of those given for the string evolver problem. Advanced students should offer experimental evidence that their method beats both the models* single tournament selection *and* roulette selection with random replacement.

Problem 2.9 Essay. *Describe, as best you can, the model of evolution used by rabbits in their reproduction. One important difference between rabbits and a string evolver is that most evolutionary algorithms have a constant population whereas rabbit populations fluctuate somewhat. Ignore this difference by assuming a population of rabbits in which births and deaths are roughly equal per unit time.*

Problem 2.10 Essay. *Repeat Problem 2.9 for honey bees instead of rabbits. Warning: this is a hard problem.*

Problem 2.11 *Suppose that we modify the model of evolution "single tournament selection with group size 4" on a population of size 4n as follows. Instead of selecting the small groups at random we select them in rotation as shown in the following table of population indices.*

Generation	Group 1	Group 2	$\cdots$	Group n
1	0123	4567	$\cdots$	(4n-4)(4n-3)(4n-2)(4n-1)
2	(4n-1)012	3456	$\cdots$	(4n-5)(4n-4)(4n-3)(4n-2)
3	(4n-2)(4n-1)01	2345	$\cdots$	(4n-6)(4n-5)(4n-4)(4n-3)
4	(4n-3)(4n-2)(4n-1)0	1234	$\cdots$	(4n-7)(4n-6)(4n-5)(4n-4)
		etc.		

Call this modification cyclic single tournament selection. *One of the qualities that makes single tournament selection desirable is that it can retard the rate at which the currently best gene spreads through the population. Would cyclic single tournament selection increase or decrease the rate of spread of a gene with relatively high fitness? Justify your answer.*

Problem 2.12 *Explain why double tournament selection of tournament size 2 without re-placement and locally elite replacement is* not *the same as single tournament selection with tournament size 4. Give an example in which a set of 4 creatures is processed differently by these two models of evolution.*

Problem 2.13 *For double tournament selection with tournament size n with replacement and without replacement, compute the expected number of mating events the best gene participates in if we do one mating event for $n = 2, 3$, or 4 in a population of size 8.*

2.2 Types of Crossover

Definition 2.3 *A* **crossover operator** *for a set of genes G is a map*

$$Cross : G \times G \to G \times G$$

or

$$Cross : G \times G \to G.$$

The points making up the pairs in the domain space of the crossover operator are termed **parents** *while the points either in or making up the pairs in the range space are termed* **children**. *The children are expected to preserve some part of the parent's structure.*

In later chapters, we will study all sorts of exotic crossover operators. They will be needed because the data structures being operated on will be more complex than strings or arrays. Even for strings, there are a number of different types of crossover. The crossover used in Experiment 2.1 is called *single point crossover*. To crossover two parents, randomly generate a locus, called the *crossover point*, and then copy the loci in the genes from the parents to the child so that the information for each child comes from a different parent before and after the crossover point.

There is a problem with single point crossover. Loci near one another in the representation used in the evolutionary algorithm are kept together with a much higher probability than those that are farther apart. If we are evolving strings of length 20 to match a string composed entirely of the character "A", then a creature with an "A" in positions 2 and 19 must almost be cloned during crossover in order to pass both good loci along. A simple way of reducing this problem is to have *multiple point crossover*. In *two point crossover*, as shown in Figure 2.3, two random loci are generated and then the loci in the children are copied from one parent before and after the crossover points and from the other parent in between the crossover points. This idea generalizes in many ways. One could, for example, generate a random number of crossover points for each crossover or specify fixed fractions of usage for different sorts of crossover.

Parent 1	aaaaaaaaaaaaaaaaaaaa
Parent 2	bbbbbbbbbbbbbbbbbbbb
Child 1	aaaabbbbbbbbbaaaaaaa
Child 2	bbbbaaaaaaaaabbbbbbb

Figure 2.3: Two point crossover

Experiment 2.4 *Modify the version of the code from Experiment 2.1 that does roulette selection with random elite replacement to work with different sorts of crossover. Run it as*

steady state algorithm for 100 trials. Use 20-character strings and a 60-member population. Measuring time in number of crossovers done, compare the mean and standard deviation of time-to-solution for the following crossover operators:

(i) *one point,*

(ii) *two point,*

(iii) *half-and-half one and two point.*

When writing up your experiment, consult with others who have done the experiment and compare your trials to theirs.

Another kind of crossover which is computationally expensive but eliminates the problem of representational bias is *uniform crossover*. This crossover operator flips a coin for each locus in the gene to decide which parent contributes its genetic value to which child. It is computationally expensive because of the large number of random numbers needed, though clever programming can reduce the cost.

This raises an issue that is critical to research in the area of artificial life. It is easy to come up with new wrinkles for use in an evolutionary algorithm; it is hard to assess their performance. If uniform crossover reduces the average number of generations, or even crossovers, to solution in an evolutionary algorithm, it may still be slower because of the additional time needed to generate the extra random numbers. Keeping this in mind, try the next experiment.

Experiment 2.5 *Repeat Experiment 2.4 with the following crossover operators:*

(i) *one point,*

(ii) *two point,*

(iii) *uniform crossover.*

In addition to measuring time in crossovers, also measure it in terms of random numbers generated and, if possible, true time by the clock. Discuss the degree to which the measures of time agree or fail to agree and frame and defend a hypothesis as to the worth of uniform crossover in this situation.

In some experiments, different crossover operators are better during different phases of the evolution. A technique to use in these situations is *adaptive crossover*. In adaptive crossover, each creature has its gene augmented by a *crossover template*, a string of 0s and 1s with one position for each item in the original data structure. When two parents are chosen, the crossover template from the first parent chosen is used to do the crossover. In positions

where the template has a 0, items go from first parent to the first child and the second parent to the second child. In positions where the template has a 1, items go from the first parent to the second child and from the second parent to the first child. The parental crossover templates are themselves crossed over and mutated with their own distinct crossover and mutation operators to obtain the children's crossover templates. The templates thus co-evolve with the creatures and seek out crossover operators that are currently useful. This can allow evolution to focus crossover activity in regions where it can help the most. The crossover templates that evolve during a successful run of an evolutionary algorithm may contain nontrivial useful information about the structure of the problem.

Example 2.1 *Suppose we are designing an evolutionary algorithm whose gene consists of 6 real numbers. A crossover template would then be a string of 6 0s or 1s, and crossover would work like this:*

	Gene						Template
Parent 1	1.2	3.4	5.6	4.5	7.9	6.8	010101
Parent 2	4.7	2.3	1.6	3.2	6.4	7.7	011100
Child 1	1.2	2.3	5.6	3.2	7.9	7.7	010100
Child 2	4.7	3.4	1.6	4.5	6.4	6.8	011101

The crossover operator used on the crossover templates is single point crossover (after position 3).

Adaptive crossover can suffer from a common problem called a *two time-scale problem*. The amount of time needed to efficiently find those fit genes that are easy to locate with a given crossover template can be a great deal less than that needed to *find* the crossover template in the first place. For some problems this will not be the case, for some it will, and intuition backed by preliminary data is the best tool currently known for telling which problems might benefit from adaptive crossover. If a problem must be solved over and over for different parameters, then saving crossover templates between runs of the evolutionary algorithm may help. In this case, the crossover templates are being used to find good representations, relative to the crossover operator, for the problem in general while solving specific cases.

Experiment 2.6 *Repeat Experiment 2.4 with the following crossover operators:*

  (i) one point,

  (ii) two point,

  (iii) adaptive crossover.

For the variation operators for the crossover templates, use one point crossover together with a mutation operator that flips a single bit 50% of the time. When comparing solution times attempt to compensate for the additional computational cost of adaptive crossover. Using real time-to-solution would be one good way to do this.

The last crossover operator we wish to mention is *null crossover*. In null crossover there is no crossover; the children are copies of the parents. Null crossover is often used as part of a mix of crossover operators or when debugging an algorithm. We conclude with a definition that will become important when we return to studying genetic programming.

Definition 2.4 *A crossover operator is called* **conservative***, if the crossover of identical parents produces children identical to those parents.*

Problems

Problem 2.14 *Assume we are working with a string evolver. If the reference string is*

$$11111111111111111111$$

then what is the expected fitness of the children of

$$11111111000000000000$$

and

$$00000000000011111111$$

under:

  (i) *one point crossover,*

 (ii) *two point crossover,*

(iii) *uniform crossover.*

Problem 2.15 *Assume we are maximizing the real function $f(x, y) = \frac{1}{x^2+y^2+1}$ with the technique described in Problem 1.12. Find a pair of parents (x_1, y_1), (x_2, y_2) such that neither parent has a fitness of more than 0.1 but one of their potential crossovers has fitness of at least 0.9. Crossover in this case would consist simply of taking the x coordinate from one parent and the y coordinate from the other. Fitness of a gene (a, b) is $f(a, b)$.*

Problem 2.16 *Usually we require that a crossover operator be conservative. Give a non-conservative crossover operator for use in the string evolver which you think will improve performance, and show why the lack of conservation might help.*

Problem 2.17 **Essay.** *If we take the point of view that evolution finds pieces of a solution and then puts them together, please explain why conservative crossover operators might be a good thing.*

Problem 2.18 *Suppose that we keep track of which pairs of parents have high or low fitness children by simply tracking the average fitness of all children produced by each pair of parents. We use these numbers to bias the selection of a second parent after the first is selected with a pure fitness bias. If this technique is used in a string evolver, will there be a two time-scale problem? Explain what two separate process are going on in the course of justifying your answer. Hint: what is the average number of children a given member of the population has?*

Problem 2.19 *Prove that, for the string evolver problem, all of the conservative crossover operators given in this section conserve fitness in the following sense: if we have a crossover operator take parents (p_1, p_2) to children (c_1, c_2), then the sum of the fitness of the children equals the sum of the fitness of the parents.*

Problem 2.20 *Read Problem 2.19. Find a problem that does not have the conservation property described. Prove your answer is correct.*

Problem 2.21 Essay. *In the definition of the term "crossover operator" there were two possibilities - producing one or two children. If we transform a crossover operator that produces two children into an operator that produces one by throwing out the least fit child, then do we disrupt the conservation property described in Problem 2.19? Do you think this would improve the average performance of a string evolver or harm it?*

Problem 2.22 *Suppose we are running a string evolver with a 20-character reference string, a crossover operator producing two children, and no mutation operator. What condition must be true for the original population for there to be any hope of eventual solution? Does the condition that allows eventual solution ensure it? Prove your answers to both these questions. Estimate theoretically or experimentally the population size required to give a 95% chance of satisfying this condition.*

2.3   Mutation

Definition 2.5 *A **mutation operator** on a population of genes G is a function*

$$Mute : G \to G$$

that takes a gene to another similar but different gene.

Crossover mixes and matches disparate creatures; it facilitates a broad search of the space of data structures accessible to a given evolutionary algorithm. Mutation, on the other hand, makes small changes in individual creatures. It facilitates a local search and also a gradual introduction of new ideas into the population. The string evolvers we have studied

use a single type of mutation, changing the value of the string at a single position. Such a mutation is called a *point mutation.* More complex data structures might have a number of distinct types of minimal changes which could serve as point mutations. Once you have a point mutation, you can use it in a number of ways to build different mutation operators.

Definition 2.6 *A* **single point mutation** *of a gene consists of generating a random position within the gene and applying a point mutation at that position.*

Definition 2.7 *A* **multiple point mutation** *consists of generating some fixed number of positions in the gene and doing a point mutation at each of them.*

Definition 2.8 *A* **probabilistic mutation with rate** α *operates by going through the entire gene and performing a point mutation with probability α at each position.*

Definition 2.9 *A* **Lamarckian mutation of depth k** *is performed by looking at all possible combinations of k or fewer point mutations and using the one that results in the best fitness value.*

Definition 2.10 *A* **null mutation** *is one that does not change anything.*

Any mutation operator may be made *helpful*, if the fitnesses of the gene before and after mutation are compared and the better result is saved. (Lamarckian mutation is already helpful since "no mutations" is included in "k or fewer point mutations.")

Any mutation operator may be applied with some probability, as was done in several of the experiments in this chapter so far. The following experiment illustrates the use of mutations.

Experiment 2.7 *Modify the standard string evolver software used in Experiment 2.1 as follows. Use roulette wheel selection and random elite replacement. Use two point crossover and put in an option to either use or fail to use single point mutation in a given run of the evolutionary algorithm. When used, the single point mutation should be applied to every new creature. Compute the average time-to-solution, cutting off the algorithm at generation 3000, if it has not found a solution yet. Report the number of runs that fail and the mean solution time of those that do find a solution. Explain the differences the mutation operator created.*

Definition 2.11 *A* **mode** *of a function is informally defined as a high point in the function's graph. Formally, a* **point mode** *is a point p in the domain of f such that there is a region R, also in the domain of f, about that point for which, for each $x \neq p \in R$, it is the case that $f(x) < f(p)$. Another type of mode is a contiguous region of points all at the same height in the graph f, such that all points around the border of that region are lower than the points in the region. Figure 2.4 shows a function with two modes.*
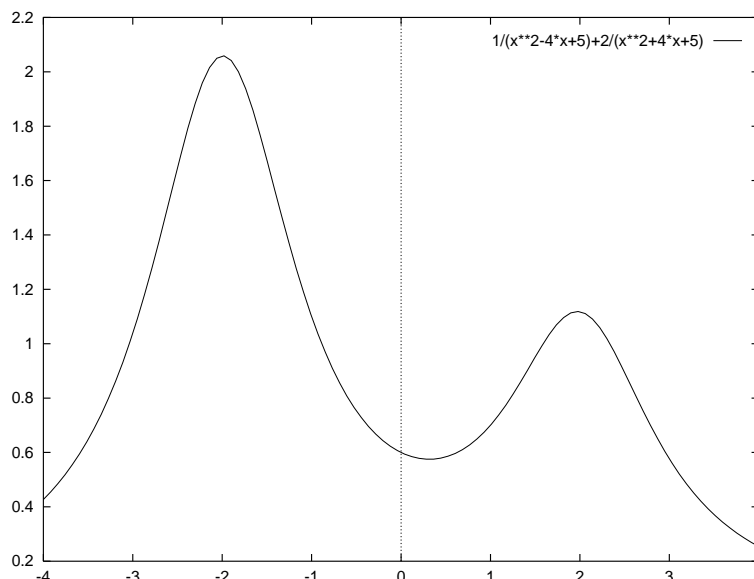
Figure 2.4: A function with two modes

The string evolver problem is what is called a *unimodal* problem; that is to say, there is one solution and an uphill path from any place in the gene space of the problem to the solution. For any given string other than the reference string, there are single character changes that improve the fitness.

Note: single character changes (no matter if they help, hurt, or fail to change fitness) induce a notion of distance between strings. Formally, the distance between any two strings is the smallest number of one character changes needed to transform one into the other. This distance, called *Hamming distance* or *Hamming metric*, makes precise the notion of similar in Definition 2.5. Mutation operators on any problem induce a notion of distance, but rarely one as nice as Hamming distance.

When designing an evolutionary algorithm, you need to select a set of mutation operators and then decide how often each one will be used. The probability that a given mutation operator will be used on a given creature is called the *rate* or *mutation rate* for that operator. The expected number of point mutations to be made in a new creature is called the *overall mutation rate* of the evolutionary algorithm. For helpful and Lamarckian mutation operators, computation of the overall mutation rate is usually infeasible - it depends on the composition of the population.

To explore the effects of changing mutation rates, we will shift from string evolvers to function optimizers. It is folklore in the evolutionary algorithms community that an overall mutation rate equal to the reciprocal of the gene length of a creature works best for moving between nearby optima of nearly equal height when doing function optimization. Experiment
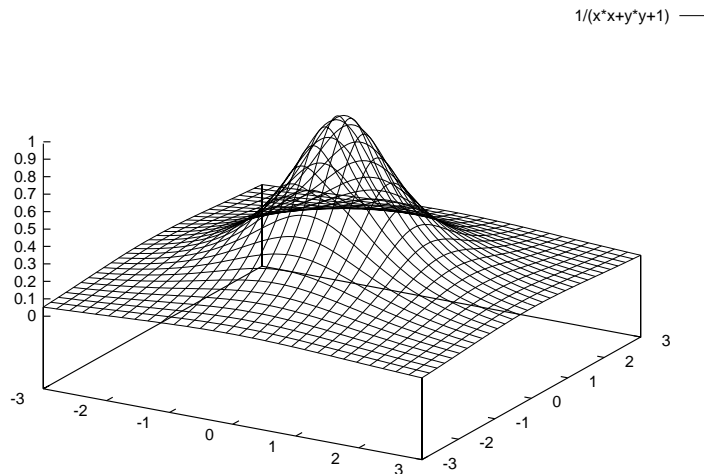
1/(x*x+y*y+1) ——

Figure 2.5: Fake bell curve $f(x, y) = \frac{1}{1+x^2+y^2}$ in two dimensions

2.8 will test this notion.

The *Fake Bell Curve in n dimensions* is given by the function

$$\mathcal{B}_n(x_1, x_2, \ldots, x_n) = \frac{1}{1 + \sum_{i=1}^{n} x_i^2}. \tag{2.1}$$

This function has a single mode at the origin of n-dimensional Euclidean space, as shown for $n = 2$ in Figure 2.5 (it is unimodal). By shifting and scaling this function we can create all sorts of test problems placing optima where we wish, though some care is needed as shown in Problem 2.25. Figure 2.4 was created in exactly this fashion.

Experiment 2.8 *Write or obtain software for a function optimizer, with no crossover operator, that uses probabilistic mutation with rate α. Use rank selection with random replacement. Use strings of real numbers of length n, where n is one of 4, 6, 8, and 10. Use overall mutation rates of r/n where r is one of 0.8, 0.9, 1, 1.1, 1.2. (Obtain the proper overall mutation rate by computing the α that yields the correct overall mutation rate: $\alpha = r/n^2$.) Run the algorithm to optimize $f_n = \mathcal{B}_n(x_1, x_2, \ldots, x_n) + \mathcal{B}_n(x_1 - 2, x_2 - 2, \ldots, x_n - 2)$. Use a population of 200 creatures all initialized to $(2, 2, \ldots, 2)$. Do 100 runs. Compute the average time for a creature to appear that is less than 0.001 in absolute value in every locus.*

In Figures 2.5 and 2.4 we give examples of functions with one and two modes, respectively. The clarity of these examples relies on the smooth, continuous nature of the real numbers - both these examples are graphs of a continuous function from a real space to a real space.

Our fundamental example, the string evolver, does not admit nice graphs. A string evolver operating on strings of length 20 would require a 20-dimensional graph to display the full detail of the fitness function. In spite of this, the string evolver fitness function is quite simple.

Problems

Problem 2.23 *Suppose we modify a string evolver so that there are two reference strings and a string's fitness is taken to be the number of positions in which it agrees with either of the reference strings. If the strings are length l over an alphabet with k characters, then how many strings in the space exhibit maximum fitness? Hint: your answer will involve the number* q *of characters on which the two reference strings agree.*

Problem 2.24 *Is the fitness function given in Problem 2.23 unimodal? Prove your answer and describe any point or non-point modes.*

Problem 2.25 *Examine the Fake Bell Curve, Equation 2.1, in 1 dimension, $f(x) = \frac{1}{1+x^2}$. If we want a function with two maxima, then we can take $f(x) + f(x - c)$ but only if c is big enough. Give the values of c for which $g(x) = f(x) + f(x - c)$ has one maximum, and those for which it has two.*

Problem 2.26 Essay. *Explain why it is difficult to compute the overall mutation rate for a Lamarckian or helpful mutation. Give examples.*

Problem 2.27 *Construct a continuous, differentiable (these terms are defined in calculus books) function f(x,y) so that the function has three local maxima with the property that the line segment P (in x-y space) from the origin through the position of the highest maximum intersects the line segment Q joining the other two maxima with the length of P at least twice the length of Q. Hint: multiply, don't add.*

Problem 2.28 *Suppose that we modify a string evolver to have two reference strings, but, unlike Problem 2.23, take the fitness function to be the maximum of the number of positions in which a given string matches one or the other of the reference strings. This fitness function can be unimodal or it can have more than one mode. Explain under what conditions the function is uni- or multi-modal.*

Problem 2.29 *Suppose that we are looking at a string evolver on strings of length 4 with underlying alphabet $\{0, 1\}$. What is the largest number of reference strings we could have, as in Problem 2.28, and have as many modes as strings?*

2.4   Population Size

Definition 2.12 *The* **population size** *of an evolutionary algorithm is the number of data structures in the evolving population.*

In biology it is known that small populations are likely to die out for lack of sufficient genetic diversity to meet environmental changes or because all members of the population share some defective gene. As we saw in Problem 2.22, analogous effects are possible even in simple evolutionary optimizers like the string evolver. On the other hand, a random initial population is usually jammed with average creatures. In the course of finding the reference string, we burn away a lot of randomness at some computational cost. There is thus a tension between the need for sufficient diversity to ensure solution and the need to avoid processing a population so large that it slows time-to-solution. Let's experiment with the string evolver to attempt to locate the sweet region and break-even point for increasing population size.

Experiment 2.9 *Modify the standard string evolver, operating on 20-character strings, as follows. Use roulette wheel selection, random elite replacement, and one point mutation applied with probability one. Use a steady state evolutionary algorithm and change the underlying alphabet to be $\{0, 1\}$. Put into the code the ability to change the population size, and gather the following data. Measure the time-to-solution in crossover events, averaged over 100 runs, for populations of size 20, 40, 60, 80, 100, 110, and 120. Approximate the best size and do a couple of additional runs near where you suspect the best size is. Graph the results as part of your write up.*

Problems

Problem 2.30 Essay. *Larger populations, having higher initial diversity, should present less need to preserve diversity. Would you expect larger populations to be of more value in preserving diversity in a unimodal or polymodal problem as compared to diversity preservation techniques like single tournament selection?*

Problem 2.31 *Give a model of evolution that can process a large population more efficiently, for the string evolver problem, than any of the ones given in this chapter. Hint: concentrate on small subsets of the population without completely ignoring anyone.*

Problem 2.32 Essay. *There is no requirement in the theory of evolutionary algorithms that we have one population. In fact, when we do 100 experimental runs, we are using 100 different populations. Give a specification, like those in the text, for an experiment that will test how many small populations 600 creatures should be divided into for an arbitrary problem. It should explore reasonably between the extremes of running one population of 600 creatures and 600 populations of one creature each.*

2.5 A Nontrivial String Evolver

An unfortunate feature of the string evolver is that it solves a trivial problem. It is possible to build very difficult string evolution problems by modifying the way in which fitness is computed. The standard example of this is the *Royal Road* function (defined by John Holland) which is defined over the alphabet $\{0, 1\}$. This function assumes a reference string of length 64, but blocks of 8 adjacent characters in positions 1-8, 9-16, ..., 57-64 are given special status. For each such block made entirely of 1's, the string's fitness is incremented by 8. Blocks with only some 1's give no fitness. This function is quite difficult to optimize and is a good test function for evolutionary optimization systems of difficult unimodal problems. The length of 64 and block size of 8 are traditional, but varying these numbers yields many possibly interesting test problems.

Definition 2.13 *Define the* **Royal Road function of length l and block size b**, *where b divides l, to be a fitness function for strings where fitness is assessed by dividing the string into $\frac{l}{b}$ pieces of length b and then giving a fitness of b for each piece on which a string in an evolving population exactly matches the reference string.*

Experiment 2.10 *Take the software you used for Experiment 2.4 and modify it to work on the Royal Road function with reference string "all ones" and alphabet $\{0, 1\}$ with $l = 16$ and $b = 1, 2, 4, 8$. Report the mean and deviation time-to-solution over 100 runs for a population of 120 creatures, cutting off an unsuccessful run at $10,000$ generations (do not include the cutoff runs in the mean and deviation computations). If you have a fast enough computer, obtain higher quality data by increasing the cutoff limit. Use two point crossover and single point mutation (with probability one). In addition to reporting and explaining your results, explain why cutoff is probably needed and is a bad thing. What is the rough dependence of time-to-solution on b?*

Experiment 2.11 *Modify the software from Experiment 2.10 so that it uses probabilistic mutation with rate α. For $l = 16$ and $b = 4$ make a conjecture about the optimum value for α and test this conjecture by finding average time-to-solution over 100 runs for 80%, 90%, 100%, 110%, and 120% of your conjectured α. Feel free to revise your conjecture and rerun the experiment.*

Problems

Problem 2.33 *Compute the probability of even one creature having nonzero fitness in the original population of n genes in a string evolver on the alphabet $\{0, 1\}$ when the fitness function is the Royal Road function of length l and block size b for the following values.*
    (i) $n = 60$, $l = 36$, $b = 6$,
    (ii) $n = 32$, $l = 49$, $b = 7$,

(iii) n = 120, l = 64, b = 8,
(iv) n = 20, l = 120, b = 10.

Problem 2.34 Essay. *Suppose we are running a string evolver with the classical Royal Road fitness function (l = 64, b = 8). Which of the mutation operators in this section would you expect to be most helpful and why? Clearly Lamarckian mutation with a depth of 8 would guarantee a solution, but it is computationally very expensive. Keeping this example in mind, factor computational cost into your discussion.*

Problem 2.35 Essay. *Single tournament selection does not perform well relative to roulette selection with random elite replacement on the basic string evolver. If possible, experimentally verify this. In any case, conjecture why this is so and say if you would expect this to also be so with the classical Royal Road fitness function (l = 64, b = 8). Support your argument with experimental data if it is available.*

Problem 2.36 *Read Problem 2.34. How many sets of point mutations must be checked in a single Lamarckian mutation of depth 8?*

Problem 2.37 *Consider a string evolver over the alphabet $\{0,1\}$ using a Royal Road fitness function with l = 4 and a population of 2 creatures. The evolver proceeds by copying a single point mutation of the best creature onto the worst creature in each generation. Estimate mathematically or experimentally the time-to-solution for b = 1, 2, 4 if the reference string is 1111 and the population is initialized to be all 0000. AppendixA on probability theory may be helpful.*

Problem 2.38 *Is the classical Royal Road fitness function unimodal or not?*

2.6   A Polymodal String Evolver

In this chapter so far we have experimented with a number of evolutionary algorithms that are working on unimodal fitness functions. In addition we've worked, in Experiment 2.8, with a constructively bimodal fitness function. In this section we will work with a highly polymodal fitness function. This polymodal fitness function is one used to locate *self-avoiding walks* that cover a finite grid.

Definition 2.14 *A **grid** is a collection of squares, called **cells**, laid out in a rectangle (like graph paper).*

Definition 2.15 *A **walk** is a sequence of moves on a grid between cells that share a side. If no cell is visited twice then the walk is **self-avoiding**. If every cell is visited then the walk is **optimal**.*

From any cell in a grid then there are four possible moves for a walk - up, down, right, and left. We will thus code walks as strings over the alphabet **{U, D, L, R}**, which will be interpreted as the successive moves of a walk. Some examples of walks are given in Figure 2.6.

UUURRRDDLULDDRR *RRUULDLUURRRDRURDDDLULD*
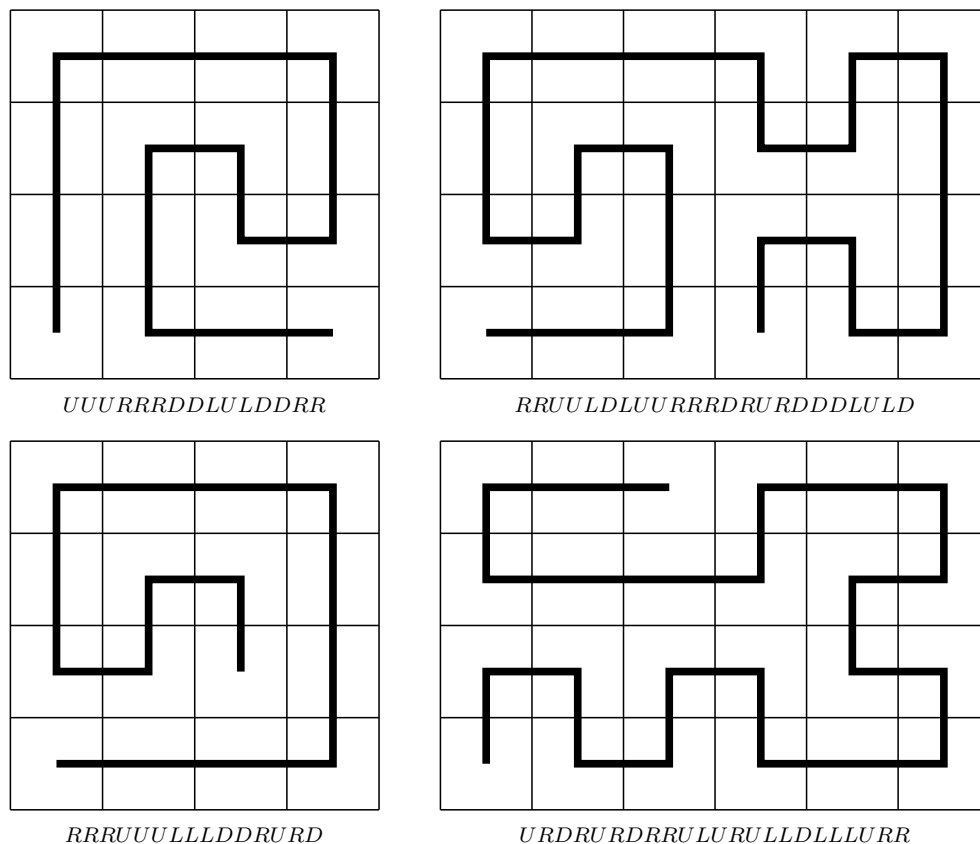
RRRUUULLLDDRURD *URDRURDRRULURULLDLLLURR*

Figure 2.6: Optimal self-avoiding walks on 4x4 and 4x6 grids that visit each cell (The walks are traced as paths starting in the lower left cell and shown in string form beneath the grids with U=up, D=down, R-right, and L=left.)

To evolve self-avoiding walks that cover a grid, we will permit the walks to fail to be self-avoiding, but we will write the fitness function so that the best score can be obtained only by a self-avoiding walk. Definition 2.16 gives such a function. If we think of self-avoiding walks as *admissible* configurations and walks that fail to avoid themselves as *inadmissible*, then we are permitting our evolutionary algorithm to search an entire space while looking for islands of admissibility. When a space is almost entirely inadmissible, attempting to search only the admissible parts of it is impractical. It is thus an interesting question, treated in the problems, what fraction of the space is admissible.

Definition 2.16 *The* coverage fitness *of a random walk of length $NM - 1$ on an $N \times M$ grid is computed as follows. Begin in the lower left cell of the grid, marking it as visited. For each of the moves in the random walk, make the move (if it stays on the grid) or ignore the move (if it attempts to move off the grid). Mark each cell reached during the walk as visited. The fitness function returns the number of cells visited.*

Notice that this fitness function requires that the walk have exactly one fewer move than there are cells, so each move must hit a new cell. The examples given in Figure 2.6 have this property.

Experiment 2.12 *Modify the basic string evolver software to work on a population of n strings with two-point crossover and k point mutation. Use size 4 tournament selection applied to the entire population. Make sure that changing n and k is easy. Run 400 populations each using the coverage fitness on 15-character strings over the alphabet* **{U, D, R, L}** *on a 4×4 grid for $n = 200, 400$ and $k = 1, 2, 3$. This is 2400 runs and will take a while on even a fast computer. Stop each individual run when a solution is found (this is a success) or when the run hits 1000 generations. Tabulate the number of successes and the fraction of successes. Discuss whether there is a clearly superior mutation operator and discuss the merits of the two population sizes (recalling that the larger one is twice as much work per generation).*
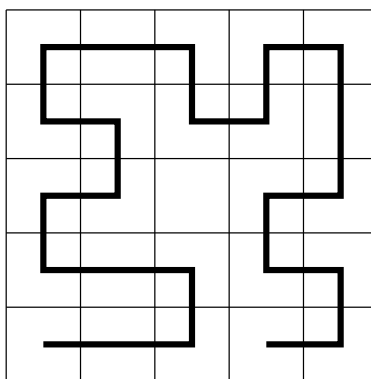

Figure 2.7: A slightly sub-optimal walk

If the code used for Experiment 2.12 does a running trace of the best fitness, then it is easy to see that the search "gets stuck" sometimes. If you save time-to-solution for the runs that terminate in less than 1000 generations, you will also observe that solution is often rapid, much faster than 1000 generations. This suggests that not only are there many global optima (Figure 2.6 shows a pair of global optima for each of two different grid sizes) but there are probably a host of local optima. Look at the walk shown in Figure 2.7. It has a

coverage fitness of 24 when the optimal is 25. It is also several point mutations from any optimal gene. Thus, this walk forms an example of a local optimum.
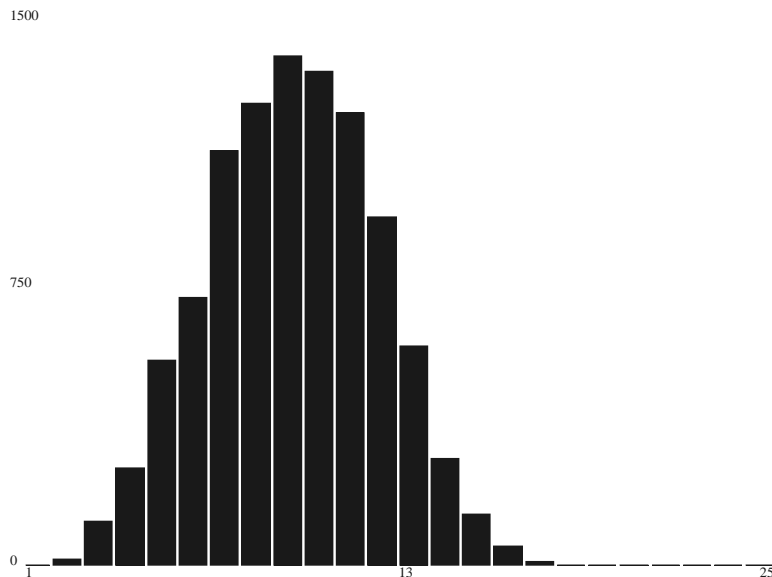

Figure 2.8: A histogram of the covering fitness of 10,000 strings of 24 moves on a 5 grid (The most common fitness was 10, attained by 1374 of the strings. The largest fitness obtained was 20.)

As we will see in the problems, each optimal self-avoiding walk has a unique encoding, but local optima have a number of distinct codings that in fact grows with their degree of suboptimality. As we attempt to approach an optimum, the fragility of our genetic representation of the walk grows. More and more of the loci are such that changing them materially decreases fitness. Let's take a look at how fitnesses are distributed in a random sample of strings coding for walks. Figure 2.8 shows how the fitnesses of 10,000 genes generated uniformly at random are distributed. Given that our evolutionary algorithms can find solutions to problems of this type, clearly the evolutionary algorithm is superior to mere random sampling. Our next experiment is intended to give us a tool for documenting the presence of a rich collection of local optima in the coverage fitness function.

Definition 2.17 *A **stochastic hill climber** is an algorithm that repeatedly modifies an initial configuration, saving the new configuration only if it is better (or no worse).*

Experiment 2.13 *Write or obtain software for a stochastic hill climber that requires new results be better for length 24 walks on a 5×5 grid starting in the lower left cell. Use single point mutation to perform modifications. Run the hill climber for 1000 steps each time you run it, and run it until you get 5 walks of fitness 20 or more. Make pictures of the walks, pooling results with anyone else who has performed the experiment.*

RU LUU D RR D UU L RRR D D LU RUU LL   *RRR L L L LU L R UUUUU R D R D D RU RR*

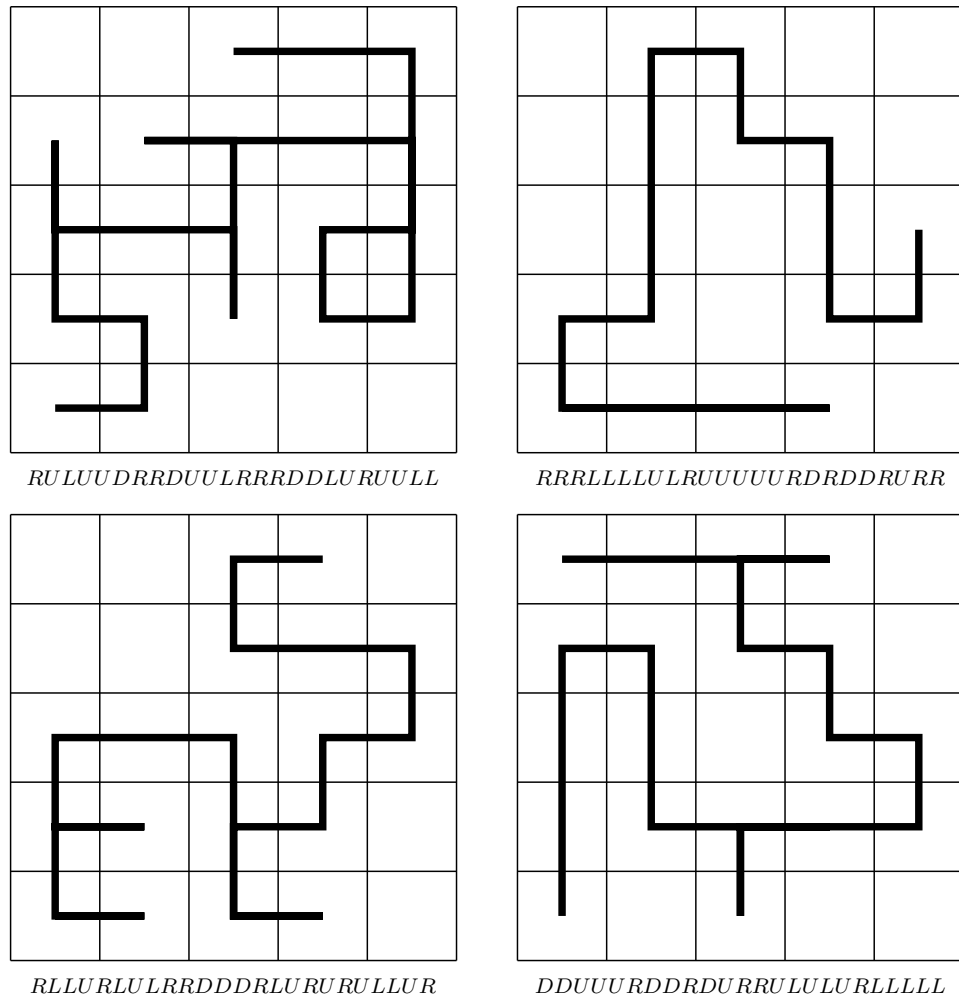*R LLU R LU L R R D D D R LU RU RU LLU R*   *D D UUU R D D R D U R RU LU LU R LLLLL*

Figure 2.9: Examples of the output of a stochastic hill climber

Figure 2.9 shows four walks generated by a stochastic hill climber. The coverage fitnesses of these walks are 20, 16, 18, and 19, respectively. All four of the walks start in the lower left cell and all four fail to self-avoid. Finally, all four arose fairly early in the 1,000 step stochastic hill climb. If these qualities turn out to be typical of the walks arrived at in Experiment 2.13 via the stochastic hill climber, then it seems that a stochastic hill climber is not the best tool for exploring this fitness landscape. In the interest of fairness, let us extend the reach of our exploration of stochastic hill climber behavior with an additional experiment.

Experiment 2.14 *Modify the stochastic hill climber from Experiment 2.13 to use two point mutation. In addition to this change, perform 10,000 rather than 1000 mutations. (This is*

probably more than necessary, but it should be computationally manageable.) Run both the old and new hill climbers 100 times and compare histograms of the resulting fitnesses.

The stochastic hill climbers in Experiments 2.13 and 2.14 require that new results be better, so they will only make a move if it leads uphill. Taking the mutated string only if it was *no worse* may tend to let the search move more, simply because "sideways" moves are permitted. Let's see what we can learn about the effect of these sideways moves.

Experiment 2.15 *Modify the stochastic hill climbers from Experiments 2.13 and 2.14 so that they accept mutated strings that are no worse. Repeat Experiment 2.14 with the modified hill climbers. Compare the results.*

A stochastic hill climber can be viewed as repeated application of a helpful mutation operator to a single member population. After having done all this work on stochastic hill climbing, it might be interesting to see how it works within the evolutionary algorithm.

Experiment 2.16 *Modify the software from Experiment 2.12 to use helpful mutation operators part of the time. Rerun the experiment for $n = 200$ and $k = 1, 2$ with 50% and 100% helpful mutation. Compare with the corresponding runs from Experiment 2.12. Summarize and attempt to explain the effects.*

At this point we conclude this phase of our exploration of polymodal fitness functions. We will revisit this fitness function in Chapter 13, where a technique for structurally enhancing evolutionary algorithms at low computational cost is explored.

Problems

Problem 2.39 *For a 3×3 grid and walks of length 8 moves, give examples of:*

(i) *An optimal self-avoiding walk other than UURRDDLU (which is given later in this section as part of a problem).*

(ii) *A non-self-avoiding walk.*

(iii) *A self-avoiding non-optimal walk.*

Notice that you will have to waste moves at the edge of the grid (which are not moves at all) in order to achieve some of the answers. Be sure to reread Definition 2.16 before doing this problem.

Problem 2.40 *Give an example of a self-avoiding walk that cannot be extended to an optimal self-avoiding walk. You may pick your grid size.*

Problem 2.41 *Make a diagram, structured as a tree, showing all self-avoiding walks on a 3×3 grid that start in the lower left cell, excluding those that waste moves off the edge of the grid. These walks will vary in length from 1 to 8. This is easy as a coding problem and a little time consuming by hand. While there are only 8 optimal self-avoiding walks there are quite a few self-avoiding walks.*

Problem 2.42 *Prove that the coverage fitness function given in Definition 2.16 awards the maximum possible fitness only to optimal self-avoiding walks.*

Problem 2.43 *Give an exact formula for the number of optimal self-avoiding walks on a $1 \times n$ and on a $2 \times n$ grid as a function of n. Assume the walks start in the lower left cell.*

Problem 2.44 *Draw all possible optimal self-avoiding walks on a 3×3 grid and a 3×4 grid. Start in the lower left cell.*

Problem 2.45 *Give an exact formula for the number of optimal self-avoiding walks on a $3 \times n$ grid and as a function of n. Assume the walks start in the lower left cell. (This is a very difficult problem.)*

UURRDDLU
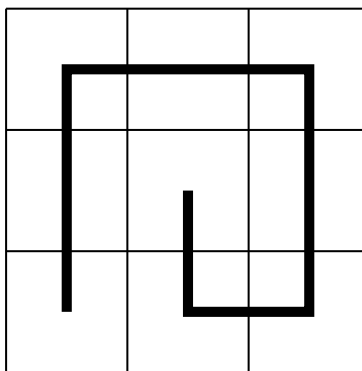
Problem 2.46 *Review the discussion of admissible and inadmissible walks in the text. For the length 8 walk given above, how many of the one point mutants of the walk are admissible? Warning: there are 3^8 one point mutants of this walk - you need either code or cleverness to do this problem.*

Problem 2.47 *Suppose that instead of wasting moves that move off the grid, we wrap the grid at the edges. Does this make the problem harder or easier to solver via evolutionary computation?*

Problem 2.48 *Prove that all single point mutations of a string specifying an optimal self-avoiding walk are themselves non-optimal.*

Problem 2.49 *Find a walk with a coverage fitness one less than the maximum on a 3×3 grid and then enumerate as many strings as you can that code for it (at least 2).*

Problem 2.50 *On a 5×5 grid make an optimal self-avoiding walk and find a point mutation such that the fitness decrease caused by the point mutation is as large as possible.*

Problem 2.51 *Construct a string for the walk shown in Figure 2.7 that ends in a downward move off the grid (there is only one such string). Now find the smallest sequence of point mutations you can that makes the string code for an optimal self-avoiding walk.*

Problem 2.52 *If you performed Experiment 2.13, then modify the software to record when the hill climber, in the course of performing the stochastic hill climb, found its best answer. Give the mean, standard deviation, and maximum and minimum times to get stuck for 1000 attempts.*

Problem 2.53 *Read the description of Experiments 2.13 and 2.14. Explain why a stochastic hill climber operating on two point mutation might need more trials per hill climbing attempt than one using one point mutation.*

Problem 2.54 *Given that we start in the lower left cell of a grid, prove there are never more than three choices of a way for a walk to leave a given grid cell in a self-avoiding fashion.*

Problem 2.55 *Based on the results of Problem 2.54, give a scheme for coding walks starting in the lower left cell of a grid with a trinary alphabet. Find strings that results in the walks pictured in Figure 2.6.*

Problem 2.56 *Prove that the fraction of genes that encode optimal self-avoiding walks is less than $\left(\frac{3}{4}\right)^{NM-1}$ on an $N \times M$ grid.*

Problem 2.57 Essay. *Based on the sort of re-encoding needed to answer Problem 2.55 (use your own if you did the problem), try to argue for or against the proposition that the re-encoding will make the space easier or harder to search with an evolutionary algorithm. Be sure to address not only the size of the search space but also the ability of the algorithm to get caught. If you are feeling gung-ho, support your argument with experimental evidence.*

2.7 The Many Lives of Roulette Selection

In Section 2.1, we mentioned roulette selection as one of the selection algorithms that can be used to build a model of evolution. It turns out that the basic roulette selection code, given in Figure 2.10, can be used for several tasks in evolutionary computation. The most

```
//Returns an integer in the range 0 to (n-1) with probability of i
//proportional to f[i]/Sum(f[i]).

int RouletteSelect(double *f;int n){  //f holds positive fitness
values
                                      //n is the number of entries in f

int i; double ttl,dart;

   ttl=0;
   for(i=0;i<n;i++)ttl+=f[i];  //compute the total fitness
   dart=ttl*random01();        //generate randomly 0<=dart<=(total fitness)
   i=-1;
   do {
      dart-=f[++i];  //subtract successive fitnesses;
   } while(dart>=0); //the one that takes you negative is
                     //where the dart landed

   return(i);  //tell the poor user what the decision is
}
```

Figure 2.10: Roulette Selection Code

basic is to perform roulette selection, but there are others. Let us trace through the roulette selection code and make sure we understand it first.

The routine takes, as arguments, an array of positive fitness values f and an integer argument n that specifies the number of entries in the fitness array. It returns an integer, the index of the fitness selected. The probability a given index i will be selected is in proportion to the fraction of the total fitness in f at $f[i]$. Why? The routine first totals f, placing the resulting total in the variable ttl. It then multiplies this total by a random number uniformly distributed in the interval $[0, 1]$ to select a position in the range $[0, \text{Total Fitness}]$, which is placed in the variable $dart$. (The variable name is a metaphor for throwing a dart at a virtual dart board that is divided into areas that are proportional to the fitnesses in f.) We then use iterated subtraction, of successive fitness values from the dart, to find out where the dart landed. If the dart is in the range $[0, f[0])$, then subtracting $f[0]$ from the dart will drive the total negative. If the dart is in the range $[f[0], f[0] + f[1])$, then the iterated subtraction will go negative once we have subtracted both $f[0]$ and $f[1]$. This pattern continues with

the effect that the probability that the iterated subtraction will drive the dart negative at index i is exactly $\frac{f[i]}{ttl}$. We thus return the value of i at which the iterated subtraction does drive the dart negative.

Now that we have code for roulette selection, let's figure out what else we can do with it. It is often desirable to select in direct proportion to a function of the fitness. If, for instance, we have fitness values in the range $0 < x < 1$ but we want some minimal chance of every gene being selected, then we might use $x + 0.1$ as the "fitness" for selection. This could be coded by simply pre-processing the fitness array f before handing it off to the *RouletteSelect* routine. In general, if we want to select in proportion to $g(fitness)$, then we need only apply the function $g(x)$ to each entry of f before using it as the "fitness" array passed to *RouletteSelect*. It is, however, important for correct functioning of both evolution and the selection code that $g(x)$ be a monotone function, i.e., $a < b \rightarrow g(a) < g(b)$.

The other major selection method in Section 2.1 was *rank selection*. Here we gave the most fit of n creatures rank n, the next most fit rank $n-1$, etc., and then selected creatures to be parents in proportion to their rank. Rank is thus nothing more than a monotone function of fitness. This means the roulette selection code is also rank selection code as long as we pass an array of ranks. If we compute the ranks in reverse fashion, with the most fit creature's rank at 1, then the roulette selection code may be used to do the selection needed for rank replacement. Roulette replacement is also achieved by a simple modification of f. Let us now consider an application of roulette selection to the computational details of mutation.

The Poisson Distributions and Efficient Uniform Mutation

When we place a probability distribution on a finite set, we get a list of probabilities, each associated with one member of the finite set. Typically a programming language comes equipped with a routine that generates random integers in the range $0, 1, \ldots, n-1$ and with another routine that generates random floating point numbers in the range $(0, 1)$. So long as a uniform distribution on an interval is all that is required, an affine transformation $g(x) = ax + b$ can transform these basic random numbers into the integer or floating point distribution required. Computing nonuniform distributions can require a good deal of mathematical muscle. In Chapter 3 we will learn to transform uniform 0-1 random numbers into normal (also called *Gaussian*) random numbers. Here we will adapt roulette selection to nonuniform distributions on finite sets and then give an application for efficiently performing probabilistic mutation.

By now the alert reader will have noticed that if we know a probability distribution on a finite set, then the roulette selection routine can generate probabilities according to that distribution if we simply hand it that list of probabilities in place of the fitness array. If, for example, we pass $f = \{0.5, 0.25, 0.25\}$ to the routine in Figure 2.10 then it will return 0 with probability 0.5, 1 with probability 0.25, and 2 with probability 0.25. In the course

of designing simulations and search software in later chapters, it will be useful to be able to select random numbers according to any distribution we wish, but at present we want to concentrate on a particular distribution, the Poisson distribution.

In Appendix A, the *binomial distribution* is discussed at some length. When we are doing n experiments, each of which can either succeed or fail, the binomial distribution lets us compute the probability of k of the experiments succeeding. The canonical example of this kind of experiment is flipping a coin with "heads" being taken as a success. Now imagine we were to flip 3000 (very odd) coins but that the chance of getting a head was only one in 1,500. Then, on average, we would expect to get 2 heads, *but* if we wanted to compute explicitly the chance of getting 0 heads, 1 head, etc., numbers like 3000! (three-thousand factorial) come into the process and our lives become a trifle difficult. This sort of situation, a very large number of experiments with a small chance of success, comes up fairly often. A statistician examining data on how many Prussian cavalry officers were kicked to death by their horses (a situation with many experiments and few "successes") discovered a short cut.

As long as we have a very large number of experiments with a low probability of success then the *Poisson distribution*, Equation 2.2, gives the probability of k successes with great accuracy.

$$P(\text{k successes}) = \frac{e^{-m} \cdot m^k}{k!} \tag{2.2}$$

The parameter m requires some explanation. It is the average number of successes. For n experiments with probability α of success we have $m = n\alpha$. In Figure 2.11 we give an example of the initial part of a Poisson distribution, both listed and plotted. How does this help us with probabilistic mutation?

When we perform a probabilistic mutation with rate α on a string with n characters, we generate a separate random number for each character in the string. If the length of the string is small, this is not too expensive. If the string has 100 characters in it, this can be a very substantial computational expense. Avoiding this expense is our object. Typically we keep the expected number of mutations, $m = n\alpha$, quite small by keeping the string length times the rate of the probabilistic mutation operator small. This means that the Poisson distribution can be used to generate the number of mutations r and then we can perform an r-point mutation.

There is one small wrinkle. As stated in Equation 2.2, the Poisson distribution gives a positive probability to each integer. This means that if we fill an n-element array with the Poisson probabilities of $0, 1, \ldots, n-1$, the array will not quite sum to 1 and will hence not quite be a probability distribution. Looking at Figure 2.11, we see that the value of the Poisson distribution drops off quite quickly. This means that if we ignore the missing terms after $n-1$ and send the not-quite-probability distribution to the routine *RouletteSelect*(f, n), we will get something very close to the right numbers of mutations. So close, in fact, that

```
P(0)=0.135335
P(1)=0.270671
P(2)=0.270671
P(3)=0.180447
P(4)=0.0902235
P(5)=0.0360894
P(6)=0.0120298
P(7)=0.00343709
P(8)=0.000859272
P(9)=0.000190949
P(10)=3.81899e-05
P(11)=6.94361e-06
     ...
```
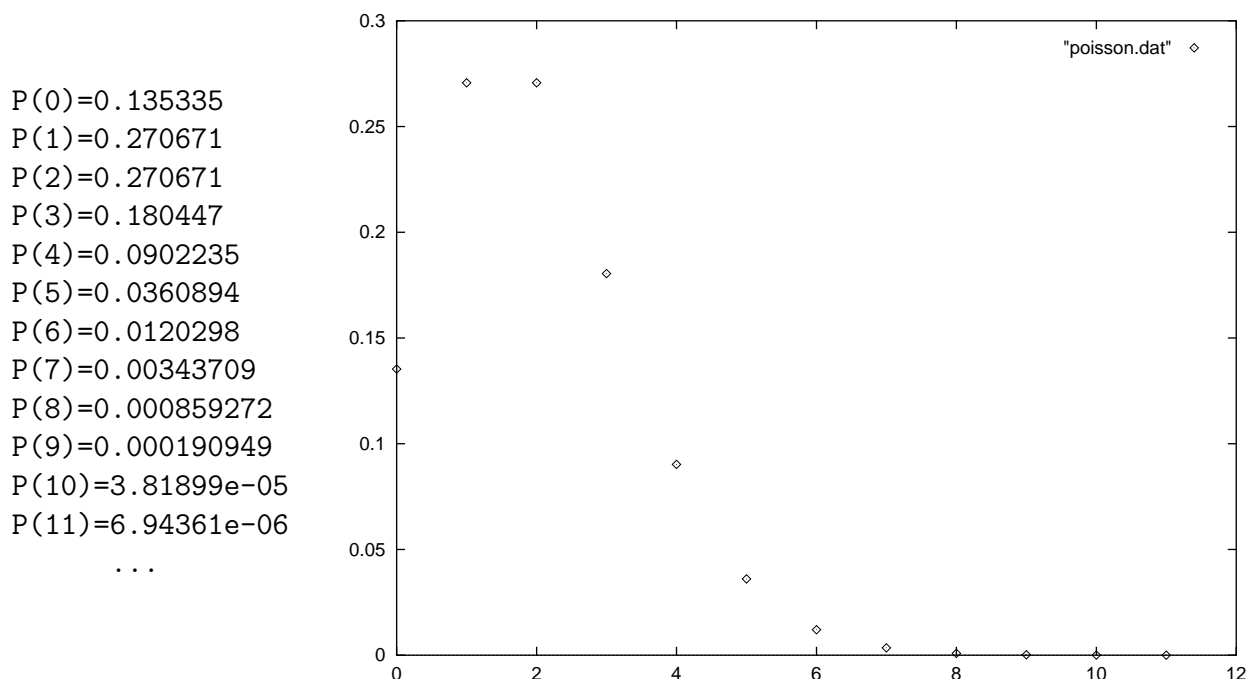
Figure 2.11: A listing and plot of the Poisson distribution with a mean of $m = 2$

it should not make any real difference in the behavior of the mutation operator. In the problems, we will examine the question of when it is worth using a Poisson distribution to simulate probabilistic mutation.

Problems

Problem 2.58 *The code given in Figure 2.10 is claimed to require that f be an array of positive fitness values. Explain why this is true and explain what will happen if (i) some zero fitness values are included or (ii) negative fitness values creep in.*

Problem 2.59 *The code given in Figure 2.10 returns an integer value without explicitly checking that it is in the range $0, 1, \ldots, n-1$. Prove that, if all the fitness values in f are positive, it will return an integer in this range.*

Problem 2.60 *Modify the RouletteSelect(f, n) routine to work with an array of integral fitness values. Other than changing the variable types, are any changes required? Why or why not?*

Problem 2.61 *If C is not your programming language of choice, translate the routine given in Figure 2.10 to your favored language.*

Problem 2.62 *Explicitly explain, including the code to modify the entries of f, how to use the RouletteSelect(f,n) code in Figure 2.10 for roulette replacement. This, recall, selects creatures to be replaced by new creatures with probability inversely proportional to their fitness.*

Problem 2.63 *Give the specialization of Equation 2.2 to a mean of $m = 1$ and compute for which k the probability of k successes drops to no more than 10^{-6}.*

Problem 2.64 *Suppose we wish to perform probabilistic mutation on a 100-character string with rate $\alpha = 0.03$. Give the Poisson distribution of the number of mutations and give the code to implement efficient probabilistic mutation as outlined in the text. Be sure to design the code to compute the partial Poisson distribution only once.*

Problem 2.65 *For an n-character string gene being modified by probabilistic mutation with rate α, compute the number of random numbers (other than those required to compute point mutations) needed to perform efficient probabilistic mutation. Compare this to the number needed to perform probabilistic mutation in the usual fashion. From these computations derive a criterion, in terms of n and α, for when to use the efficient version of probabilistic mutation instead of the standard one.*

Problem 2.66 *Suppose we are using an evolutionary algorithm to search for highly fit strings that fit a particular criterion. Suppose also that all good strings, according to this criterion, have roughly the same fraction of each character but have them arranged in different orders. If we know a few highly fit strings and want to locate more, give a way to apply RouletteSelect(f,n) to generate initial populations that will have above average fitness and so will let us sample the collection of highly fit strings more efficiently.*

Problem 2.67 *Suppose we have an evolutionary algorithm that uses a collection of several different mutation operators. For each, we can keep track of the number of times it is used and the number of times it enhances the fitness of the gene it operates on. From this we can get, by dividing these two numbers, an estimate of the probability each mutation operator has of improving a given gene. Clearly, using the most useful mutation operators more often would be good. Give a method for using RouletteSelect(f,n) to probabilistically select mutation operators according to their estimated usefulness.*

Problem 2.68 Essay. *Read Problem 2.67. Suppose we have a system for estimating the usefulness of several mutation operators. In Problem 2.67, this estimate is the ratio of applications of a mutation operator that enhanced fitness to the total number of applications of that mutation operator. It is likely that the mutation operators that help the most with an initial, almost random, population will be different from those that help the most with a converged population. Suggest and justify a method for estimating the* recent *usefulness of*

each mutation operator, such as would enhance performance when used with the system described in Problem 2.67. Discuss the computational complexity of maintaining these moving estimates and try to keep the computational cost of your technique low.