6. An Evolutionary Approach

The works of nature must all be accounted good.

Cicero, *De Senectute*

In the previous three chapters we discussed various classic problem-solving methods, including dynamic programming, branch and bound, and local search algorithms, as well as some modern heuristic methods like simulated annealing and tabu search. Some of these techniques were seen to be deterministic. Essentially you "turn the crank" and out pops the answer. For these methods, given a search space and an evaluation function, some would always return the same solution (e.g., dynamic programming), while others could generate different solutions based on the initial configuration or starting point (e.g., a greedy algorithm or the hill-climbing technique). Still other methods were probabilistic, incorporating random variation into the search for optimal solutions. These methods (e.g., simulated annealing) could return different final solutions even when given the same initial configuration. No two trials with these algorithms could be expected to take exactly the same course. Each trial is much like a person's fingerprint: although there are broad similarities across fingerprints, no two are exactly alike.

There's an interesting observation to make with all of these techniques: each relies on a single solution as the basis for future exploration with each iteration. They either process complete solutions in their entirety, or they construct the final solution from smaller building blocks. Greedy algorithms, for example, successively build solutions according to maximizing locally available improvements. Dynamic programming solves many smaller subproblems before arriving at the final complete solution. Branch and bound methods organize the search space into several subspaces, then search and eliminate some of these in a systematic manner. In contrast, local search methods, simulated annealing, and tabu search process complete solutions, and you could obtain a potential answer (although quite likely a suboptimal one) by stopping any of these methods at any particular iteration. They always have a *single* "current best" solution stored that they try to improve in the next step. Despite these differences, every one of these algorithms works on or constructs a single solution at a time.

Speaking quite generally, the approach of keeping the best solution found so far and attempting to improve it is intuitively sound, remarkably simple, and often quite efficient. We can use (1) deterministic rules — e.g., hill-climbing uses the rule: if an examined neighbor is better, proceed to that neighbor and continue searching from there; otherwise, continue searching in the current neighborhood; (2) probabilistic rules — e.g., simulated annealing uses the rule: if an

Z. Michalewicz et al., *How to Solve It: Modern Heuristics*

examined neighbor is better, accept this as the new current position; otherwise, either probabilistically accept this new weaker position anyway or continue to search in the current neighborhood; or even (3) the history of the search up to the current time — e.g., tabu search uses the rule: take the best available neighbor, which need not be better than the current solution, but which isn't listed in memory as a restricted or "tabu" move.

Now let's consider a revolutionary idea, one that doesn't appear in any of the methods that we discussed previously. Let's abandon the idea of processing only a single solution. What would happen if we instead maintained several solutions simultaneously? That is, what would happen if our search algorithm worked with a *population* of solutions?

At first blush, it might seem that this idea doesn't provide us with anything really new. Certainly, we can process several solutions in parallel if a parallel computer is available! If this were the case, we could implement a parallel search method, like a parallel simulated annealing or tabu search, where each processor would maintain a single solution and each processor would execute the same algorithm in parallel. It would seem that all there is to be gained here is just the speed of computation: instead of running an algorithm k times to increase the probability of arriving at the global optimum, a k-processor computer would complete this task in a single run, and therefore uses much less real time.

But there's an additional component that can make population-based algorithms essentially different from other problem-solving methods: the concept of competition between solutions in a population. That is, let's simulate the evolutionary process of competition and selection and let the candidate solutions in our population fight for room in future generations! Moreover, we can use random variation to search for new solutions in a manner similar to natural evolution.

In case you haven't heard this analogy before, let's digress with some poetic license to a somewhat whimsical example of rabbits and foxes. In any population of rabbits,[1] some are faster and smarter than others. These faster, smarter rabbits are less likely to be eaten by foxes, and therefore more of them survive to do what rabbits do best: make more rabbits! This surviving population of rabbits starts breeding. Breeding mixes the rabbits' genetic material, and the behaviors of the smarter and faster rabbits are heritable. Some slow rabbits breed with fast rabbits, some fast with fast, some smart ones with not-so-smart ones, and so on. On top of that, nature throws in a "wild hare" every once in a while because all of the genes in each rabbit have a chance of mutating along the way. The resulting baby rabbits aren't copies of their parents, but are instead random perturbations of these parents. Many of these babies will grow up to express the behaviors that help them compete even better with foxes, and other rabbits! Thus, over many generations, the rabbits may be expected to become smarter and faster than those that survived the initial generation. And don't forget that the foxes are evolving at the same time, putting added pressure on

[1] An individual rabbit represents a solution for a problem. It represents a single point of the search space.

the rabbits. The interaction of the two populations would seem to push both of them to their physical limits. Foxes are forced to get better at finding a meal and rabbits get better at avoiding being someone else's lunch!

Let's re-tell the above story in terms of search spaces, evaluation functions, and potential solutions. Suppose we start with a population of initial solutions, perhaps generated by sampling randomly from the search space $\mathcal{S}$. Sometimes, to emphasize a link with genetics, these solutions are called *chromosomes*, but there is no need to think of biological terms.[2] The algorithms that we will explore are just that — algorithms — mathematical procedures based on analogies to natural evolutionary processes, and there is no need for our so-called chromosomes to behave in any way like nature's chromosomes. We may simply describe the candidate solutions as vectors, or whatever representation and data structure we select.

The evaluation function can be used to determine the relative merit of each of our initial solutions. This isn't always straightforward because multiple objectives need to be combined in a useful way. For example, which rabbit is better, the one that is fast and dumb, or slow but smart? Natural selection makes this decision by using a quality control method that might be called "trial by foxes." The proof of which rabbit is better able to meet the demands imposed by the environment rests in which ones can survive encounters with the foxes, or evade them all together. But whereas nature doesn't have a mathematical function to judge the quality of an individual's behavior, we often have to determine such a function. At least, the chosen evaluation function must be capable of differentiating between two individuals, i.e., it has to be able to rank one solution ahead of another. Those solutions that are better, as determined by the evaluation function, are favored to become parents for the next generation of offspring.

How should we generate these offspring? Well, actually we've seen the basic concept in most other algorithms: new solutions are generated probabilistically in the neighborhood of old solutions. When applying an evolutionary algorithm, however, we don't have to rely only on the neighborhoods of each individual solution. We can also examine the neighborhoods of pairs of solutions. That is, we can use more than one parent solution to generate a new candidate solution. One way we can do this is by taking parts of two parents and putting them together to form an offspring. For example, we might take the first half of one parent together with the second half of another. The first half of one parent might be viewed as an "idea," and similarly so with the second half of the second parent. Sometimes this recombination can be very useful: You put cookie dough together with ice cream and you get a delicious treat. But sometimes things don't work out so well: You put the first half of Shakespeare's *Hamlet* together with the second half of *King Lear* and you still get Shakespeare but it just doesn't make any sense any more. Another way of using two solutions to generate a new possibility occurs when facing continuous optimization problems. In these cases we can blend parameters of both parents, essentially performing

[2]In fact, the use of biological terms may offer the illusion that there is more of direct connection to natural evolution than can be justified in a particular example.

a weighted average component by component. Again, sometimes this can be advantageous, and others times it can be a waste of time. The appropriateness of every search operator depends on the problem at hand.

Since our evolutionary algorithms only exist in a computer, we don't have to rely on real biological constraints. For example, in the vast majority of sexual organisms (not including bacteria), mating occurs only between pairs of individuals, but our evolutionary searches can rely on "mating" or "blending" more than two parents, perhaps using *all* of the individuals in the population of each generation to determine the probabilities for selecting each new candidate solution.

With each generation, the individuals compete — either only among themselves or also against their parents — for inclusion in the next iteration. It's very often the case that after a series of generations, we observe a succession of improvements in the quality of the tested solution and a convergence toward the neighborhood of a nearly optimum solution.

Without any doubt, this is an appealing idea! Why should we labor to solve a problem by calculating difficult mathematical expressions or developing complicated computer programs to address approximate models of a problem if we can simulate the essential and basic process of evolution and discover nearly optimal solutions using models of much greater fidelity, particularly when the difficult part of the search essentially comes "for free." Things that are free are usually too good to be true, so it is appropriate to ask some probing questions. What kind of effort is expected from a user of an evolutionary algorithm? In other words, how much work does it take to implement these concepts in an algorithm? Is this cost effective? And can we *really* solve real-world problems using such an evolutionary approach? We'll answer these questions in the remainder of the book.

In the following sections we revisit the three problems introduced in chapter 1: the satisfiability problem (SAT), the traveling salesman problem (TSP), and a nonlinear programming problem (NLP), and we demonstrate some possibilities for constructing evolutionary algorithms to address each one.

6.1 Evolutionary approach for the SAT

Suppose that we wanted to use an evolutionary algorithm to generate solutions for an SAT of 100 variables. Let's say that the problem is essentially like that of chapter 1, section 1.1, where we have a compound Boolean statement in the form

$$F(\mathbf{x}) = (x_{17} \vee \overline{x}_{37} \vee x_{73}) \wedge (\overline{x}_{11} \vee \overline{x}_{56}) \wedge \ldots \wedge (x_2 \vee x_{43} \vee \overline{x}_{77} \vee \overline{x}_{89} \vee \overline{x}_{97}),$$

and we need to find a vector of truth assignments for all 100 variables x_i, $i = 1, \ldots, 100$, such that $F(\mathbf{x}) = 1$ (TRUE). Since we are dealing with variables that can only take on two states, TRUE or FALSE, a natural representation to choose is a binary string (vector) of length 100. Each component in the string

signifies the truth assignment for the variable which corresponds to that location in the string. For example, the string $\langle 1010...10 \rangle$ would assign TRUE to all odd variables, $x_1, x_3, \ldots, x_{99}$, and FALSE to all even variables, $x_2, x_4, \ldots, x_{100}$, under the coding of 1 being TRUE and 0 being FALSE. Any binary string of 100 variables would constitute a possible solution to the problem. We could use $F(\mathbf{x})$ to check any such vector and determine whether it satisfied the necessary conditions to make $F(\mathbf{x}) = 1$.

We need an initial population to start this approach, so our next consideration is the population size. How many individuals do we want to have? Let's say that we want 30 parent vectors. One way to determine this collection would be to have a 50–50 chance of setting each bit in each vector to be either 1 or 0. This would, hopefully, give us a diverse set of parents. Typically, this diversity is desirable, although in some cases we might want to initialize all of the available parents to some best-known solution and proceed from there. For the sake of our example, however, let's say that we use this randomized procedure to determine the initial population. Note immediately, then, that every trial of our evolutionary algorithm will have a potentially different initial population based on the random sampling that we generate.

Suppose that we have the initial randomized population. What we need to do next is evaluate each vector to see if we have discovered the optimum solution and if not, we need to know the quality of the solutions we have at hand; that is, we need an evaluation function. This presents a difficulty. It is straightforward to use $F(\mathbf{x})$ to check if any of our initial vectors are perfect solutions, but let's suppose that none of these vectors turns out to be perfect; i.e., for each randomly selected initial vector, $\mathbf{x}$, $F(\mathbf{x}) = 0$. Now what? This doesn't provide any useful information for determining how close a particular vector might be to the perfect solution. If we use $F(\mathbf{x})$ as the evaluation function, and if the perfect solution were $\langle 11...11 \rangle$ and two solutions in the initial population were $\langle 00...00 \rangle$ and $\langle 11...10 \rangle$, these latter solutions would both receive an evaluation of 0, but the first vector has 100 incorrectly set variables while the second is only one bit away from perfection! Clearly $F(\mathbf{x})$ does not provide the information we need to guide a search for successively improved strings.

This is a case where the objective is not suggestive of a suitable evaluation function. The objective of making the compound Boolean statement $F(\mathbf{x}) = 1$ is insufficient. What *is* sufficient? At first, we might be tempted to simply compare the number of bits that differ between each candidate solution and the perfect solution and use this as an error function. The more bits that are incorrectly specified, the worse the error, and then we could seek to minimize the error. The trouble with this is that it presumes we already know the perfect vector in order to make the comparison, but if we knew the perfect vector, we wouldn't need to search for a solution! Instead, what we might do is consider each part of the compound Boolean expression $F(\mathbf{x})$ that is separated by an "and" symbol ($\wedge$), e.g., $(x_{17} \vee \overline{x}_{37} \vee x_{73})$, and count up how many of these separate subexpressions are TRUE. We could then use the number of TRUE subexpressions as a evaluation function, with the goal of maximizing the number of TRUE subexpressions,

because, given $F(\mathbf{x})$ as it appears above, if all of the subexpressions are TRUE then it follows that $F(\mathbf{x}) = 1$, just as we desire. Do we know that this evaluation function will lead us to the perfect solution? Unfortunately, we don't. All we can say before some experimentation is that it appears to be a reasonable choice for the evaluation function.

Let's presume that we use this function to judge the quality of our 30 initial vectors, and let's further presume that there are 20 subexpressions that comprise $F(\mathbf{x})$; therefore, the best score we can get is 20 and the worst is 0. Suppose that the best vector in the population after this initial scoring has a score of 10, the worst has a score of 1, and the average has a score somewhere between 4 and 5. What we might do next is think about how we want to generate the individuals to survive as parents for the subsequent generation. That is, we need to apply a selection function.

One method for doing this assigns a probability of being selected to each individual in proportion to their relative fitness. That is, a solution with a score of 10 is 10 times more likely to be chosen than a solution with a score of 1. This *proportional selection* is also sometimes called *roulette wheel selection* because a common method for accomplishing this procedure is to think of a roulette wheel being spun once for each available slot in the next population, where each solution has a slice of the roulette wheel allocated in proportion to their fitness score (see figure 6.1). If we wanted to keep the population at a size of 30, we would allocate 30 slices to the wheel, one for each existing individual, and we would then spin the wheel 30 times, once for each available slot in the next population. Note, then, that we might obtain multiple copies of some solutions that happened to be chosen more than once in the 30 spins, and some solutions very likely wouldn't be selected at all. In fact, even the best solution in the current population might not be selected, just by random chance. The roulette wheel merely biases the selection toward better solutions, but just like spinning the roulette wheel in a casino, there are no guarantees!

Suppose that we have spun the wheel 30 times and have chosen our 30 candidates for the next generation. Perhaps the best solution is now represented three times, and the worst solution failed to be selected even once. It has now disappeared from consideration. Regardless of the resulting distribution of our solutions, we haven't generated anything new, we have only modified the distribution of our previous solutions. Next we need to use random variation to search for new solutions.

One possibility for doing this would be to randomly include the chance that some of our new 30 candidates will recombine. For each individual we might have a probability of, say, 0.9 that it will not pass unchanged into the next generation, but instead will be broken apart and reattached with another candidate solution chosen completely at random from the remaining 29 other solutions. This two-parent operator would essentially be a cut-and-splice procedure designed, hopefully, to ensure that successive components in candidate strings that tend to generate better solutions have a chance of staying together. Whether or not this would be effective partly depends on the degree of independence of the com-

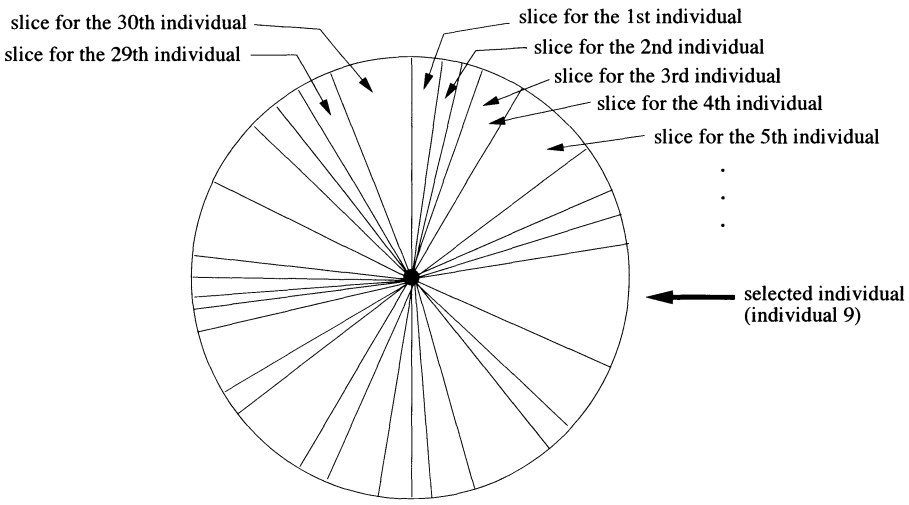

Fig. 6.1. A roulette wheel with 30 slices. The size of each slice corresponds to the fitness of the appropriate individual. From the graphic here, individual 9 had the highest fitness.

ponents. If the components of the candidate vectors aren't independent in light of the evaluation function, this recombination operator might end up metaphorically cutting and splicing *Hamlet* and *King Lear*. Another potential difficulty is that this recombination operator can only generate new solutions for evaluation if there is sufficient diversity in the current population. If we are unlucky and the roulette wheel happened to generate a homogeneous population, we'd be completely stuck, even if our current best solution wasn't a local optimum! As some insurance, we might also include another operator that flips a single bit in a chosen string at random, and then apply this with a low probability. In this way, we'd ensure that we always have the potential to introduce novel variations.

These five steps of specifying (1) the representation, (2) the initial population, (3) the evaluation function, (4) the selection procedure, and (5) the random variation operators define the essential aspects of one possible evolutionary approach to the SAT. By iterating over successive generations of evaluation, selection, and variation, our hope is that the population will discover increasingly appropriate vectors that satisfy more and more of the subexpressions of $F(\mathbf{x})$, and ultimately satisfy the condition that $F(\mathbf{x}) = 1$.

6.2 Evolutionary approach for the TSP

Suppose that we wanted to use an evolutionary algorithm to generate solutions for a TSP of 100 cities. To make things easier, let's say that it's possible to

travel from any city to any other city, and that the distances between the cities are fixed, with the objective of minimizing the distance that the salesman has to travel while visiting each city once and only once, and then returning to his home base.

Our first consideration is again the representation of a possible solution. We have several possible choices but the natural one that comes immediately to mind is to use an ordered list of the cities to be visited. For example, the vector

[17, 1, 43, 4, 6, 79, ..., 100, 2]

would mean that the salesman starts at city 17, then goes to city 1, then on to cities 43, 4, 6, 79, and so on through all of the cities until they pass through city 100, and lastly city 2. They still have to get home to city 17, but we don't need any special modification to our representation to handle that. Once we move the salesman to the last city in the list, it's implied that there is one more city to travel to, namely, the first city in the list. It's easy to see that any permutation of this list of cities generates a new candidate solution that meets the constraints of visiting each city one time and returning home.

To begin, we have to initialize the population, so let's again figure that we have 30 candidate tours. We can generate 30 random permutations of the integers [1...100] to serve as the starting population. If we had some hints about good solutions, maybe from some salesmen who have traveled these routes before, we could include these, but let's say for the sake of example that we don't have any such prior information.

Once we have generated our 30 candidate tours, we have to evaluate their quality. In this case, the objective of minimizing the total distance of the tour suggests using the evaluation function that assigns a score to each tour that is equivalent to its total length. The longer the tour, the worse the score, and our goal then is to find the tour that minimizes the evaluation function. This part of the evolutionary approach appears much more straightforward for the TSP than for the SAT.

Let's say that we use the total distance to assess our 30 initial tours. We are now left with the tasks of applying selection and random variation, but we don't have to apply them in this order. As an alternative to the example for the SAT, suppose that we decide to generate new solutions first, evaluate them, and then apply selection.

Our first decision then concerns how to generate new solutions from the existing candidate tours. It's always nice to take advantage of knowledge about the problem we face [309], and this is one case where we can do this. Our evaluation function is based on distances in Euclidean space, so consider this question: Which is longer, the sum of the lengths of the diagonals of quadrilateral, or the sum of the lengths of either pair of opposite sides? Figure 6.2 illustrates the case. By Euclidean geometry, the sum of the diagonals (e and f) are always longer than the sum of the opposing sides (a and c or b and d), and by consequence this means that tours that cross their own path are longer than those

that don't. So perhaps we can devise a variation operator that would remove instances where a parent tour crosses its own path.

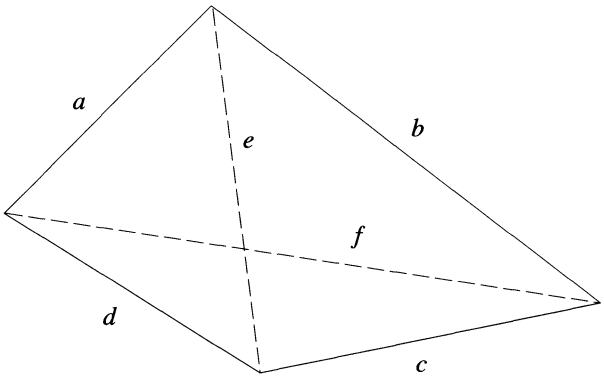

Fig. 6.2. A quadrilateral. Note that $a + c < e + f$ and $b + d < e + f$.

Indeed, this is very easily done for the representation we have chosen. For any ordered list of cities, if we pick two cities along the list and reverse all of the cities between the starting and the ending point, we have the ability to repair a tour that crosses its own path. Of course, if we apply this operator to solutions that don't cross their own paths, then we may introduce the same problems! But we can't know beforehand which solutions will be crossing over on themselves and which won't, so one reasonable approach is to simply pick two cities on the list at random and generate the new solution. If the variation isn't successful, we'll rely on selection to eliminate it from further consideration.

Like the example of the SAT before, we might think about using two or more tours to generate a new solution, but this would require some careful consideration. It's plain that we couldn't just take the first part of one solution and splice on the second part of another. In that case we would very likely generate a solution that had multiple instances of some cities and no instances of others (figure 6.3). It would violate the constraints of the problem that demand we visit every city once and only once. We could in turn simply eliminate these sorts of solutions, essentially giving them an error score of infinity, but they would be generated so frequently that we'd spend most of our time in vain, discovering infeasible solutions and then rejecting them. Under our chosen representation, we'd have to be fairly inventive to construct variation operators that would rely on two parents and generate feasible solutions.

For example, we might take two parents and begin at the first location in each tour by randomly selecting the first city in either the first parent or the second parent to be the first city in the offspring's tour list. Then we could proceed to the second position in both parents, again choose randomly from either parent, and continue on down the list. We'd have to ensure at each step that we didn't copy a city that was already copied in a previous step. To do

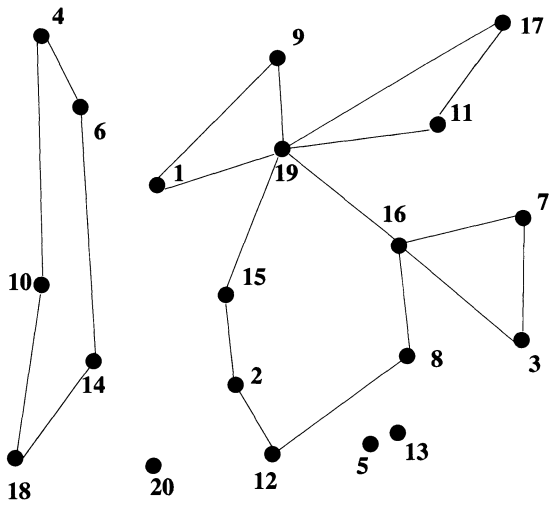

Fig. 6.3. An illegal solution for the TSP. The depicted routing is infeasible because it fails to meet the constraints of visiting each city once and only once while completing a circuit.

that, we could instead simply choose to copy from the other parent, or in the case where the cities in both parents at a particular location had already been placed in the list, we could simply choose a new city at random such that we didn't violate the constraints of visiting each city once and only once.

Suppose that instead of generating a number of offspring that is equal to the number of parents, we decide to generate three offspring per parent. We can choose any number we believe to be appropriate. For each of the 30 parents, three times in succession, we pick two cities at random in the parent, flip the list of cities between these points, and enter this new solution in the population. When we are through, we have 90 offspring tours.

We now have a choice: we can put the 90 offspring in competition with the 30 parents and select new parents from among all $30 + 90 = 120$ solutions, or we can discard the parents and just focus on the offspring. Although discarding the parents might not seem prudent, there are some supposed justifications that can be offered for this choice, but these will have to wait until chapter 7. For now, let's say that we will focus on the 90 offspring, and from these we will keep the best 30 to be parents for the next generation. This form of selection is described as (μ, λ) where there are μ parents and λ offspring, and then selection is applied to maintain the best μ solutions considering only the offspring. Here, we have a (30,90) procedure.

We could execute this process of random variation by using the two-city reversal to each parent, or the modified two-parent operator, generating three offspring per parent, and then applying selection to cull out the 60 worst new

solutions. Over time, our expectation is that the population should converge toward nearly optimal tours.

6.3 Evolutionary approach for the NLP

Suppose we wanted to solve the NLP given in chapter 1 (figure 1.2) for the case of 10 variables. Given the above descriptions of possible ways to address the SAT and TSP, the NLP should be pretty straightforward. After all, the first thing to choose is a representation, and here the most intuitive choice is a vector of 10 floating-point values, each corresponding to a parameter in the problem. When we think about initializing the population, we have the constraints to worry about, but even then we could just choose values for the parameters uniformly at random from [0,10] and then check to be certain that we didn't violate the problem constraints. If we did, we could just discard that initial vector and generate a new one until we filled the initial population. The volume in $[0, 10]^{10}$ that violates the constraints is relatively small so this shouldn't waste too much computational time.

Next we'll need to generate offspring. For floating-point vectors, a common variation operator is to add a Gaussian[3] random variable to each parameter. You will recall that a Gaussian density function looks like a bell curve (figure 6.4). To define a Gaussian random variable we need a mean and a variance. For our example here, we choose the mean to be 0 and the variance to be 1 (i.e., a standard Gaussian). We can prove later that there are better choices, but this will suffice for now. To generate an offspring, we add a different Gaussian random variable to each component of a parent, making 10 changes, each one independent of the previous one. When we are done, we have an offspring that on average looks a lot like its parent, and the probability of it looking radically different in any parameter drops quite quickly because most of the probability density in a Gaussian lies within one standard deviation of the mean. This is often a desirable feature because we need to be able to maintain a link between parents and offspring. If we set the variance too high, the link can be broken, resulting in essentially a purely random search for new points. This is to be avoided because it fails to take into account what has already been learned over successive generations of evolution. The Gaussian variation also has the advantage of being able to generate any possible new solution, so the evolutionary process will never stall indefinitely at a suboptimum. Given a sufficiently long waiting time, all points in the space will be explored.

Once all of the offspring are generated, we have to evaluate them. Here again, as with the TSP, the objective suggests an evaluation function. We could just use the mathematical expression to be maximized as the evaluation function directly except that we also have constraints to deal with. And while the expression for the objective is defined over all the reals in each dimension, the feasible region is a small finite subset of the entire real space. One possibility in

[3]Also known as a *normal*.

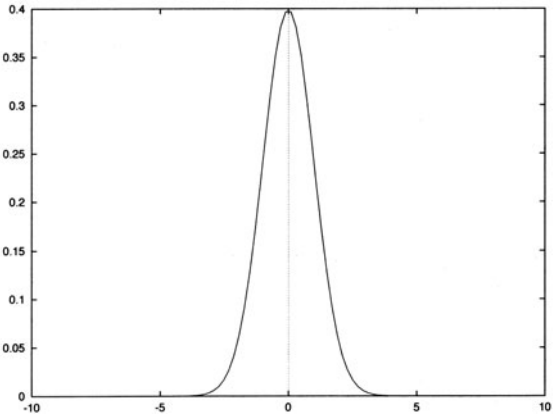

Fig. 6.4. A normal distribution $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-m)^2/(2\sigma^2)}$ with standard deviation $\sigma = 1$ and mean $m = 0$

such cases is to apply a penalty function where the score an individual receives is the value generated by plugging it into the mathematical expression of the objective. This objective is also modified by a penalty for the degree to which it violates the applicable constraints. The further the individual moves into an infeasible region, the greater the penalty. Scaling the penalty function appropriately is a matter of judgment and experience, but for now we might just say that the evaluation function is the mathematical expression of the objective minus the distance from the individual to a boundary of the feasible region (if the individual is infeasible). The goal then is to maximize this sum.[4]

For selection, we could apply the forms we saw in the SAT or TSP, but we can introduce a third choice here: ranking selection using a tournament. Suppose that we started with 30 individuals and generated 30 more through our Gaussian variation (one offspring per parent). We'd then have a collection of 60 vectors. Suppose further that we'll keep the population at the start of each generation constant at 30. To choose these 30 new parents, one tournament method might be to pick pairs of solutions at random and determine the better one, which is placed in the next generation. After 30 such pairings we have a new population where those individuals that are better on average have a greater chance of surviving, but are not completely ensured of passing on into the next generation.

Over successive iterations of variation, evaluation (with penalties), and this form of tournament selection, we could expect to evolve an individual that meets the constraints and resides close to the global optimum. There are a number of ways that might accelerate the rate at which good solutions could be discovered

[4]Alternatively, for a minimization problem, the penalty would be added rather than subtracted.

by changing parameters of the evolutionary operators, but these can be deferred to later chapters.

6.4 Summary

The structure of an evolutionary algorithm is shown in figure 6.5.

```
procedure evolutionary algorithm
begin
  t ← 0
  initialize P(t)
  evaluate P(t)
  while (not termination-condition) do
  begin
    t ← t + 1
    select P(t) from P(t − 1)
    alter P(t)
    evaluate P(t)
  end
end
```

Fig. 6.5. The structure of an evolutionary algorithm

The evolutionary algorithm maintains a population of individuals, $P(t) = \{x_1^t, \ldots, x_n^t\}$ for iteration t. Each individual represents a potential solution to the problem at hand and, in any evolutionary algorithm, is implemented as some data structure S. Each solution x_i^t is evaluated to give some measure of its "fitness." A new population at iteration $t+1$ is then formed by selecting the more fit individuals (the "select" step). Some members of the new population undergo transformations (the "alter" step) by means of variation operators to form new solutions. There are unary transformations u_i that create new individuals by a change in a single individual ($m_i : S \to S$), and higher-order transformations c_j that create new individuals by combining parts from several (two or more) individuals ($c_j : S \times \ldots \times S \to S$). The algorithm is executed until a predefined halting criterion is reached, which might include finding a suitable solution, generating a specified number of candidate solutions, or simply running out of available time.

The history of applying these sorts of algorithms dates back into the early 1950s and was actually invented as many as ten times by different scientists, completely independent of each other [149]. Each procedure was slightly different from the others. Some of the researchers in the 1960s gave their algorithms different names like *genetic algorithms*, *evolution strategies*, or *evolutionary programming*, but not everyone chose to christen their inventions. Over time, more specialized techniques developed that were associated with particular data structures, such as *classifier systems* and *genetic programming*. In the

1990s, however, there has been considerable theoretical and empirical evidence that none of these canonical approaches can offer anything that another approach cannot achieve. What's more, as interest in evolutionary algorithms has rapidly increased, ideas have been borrowed, exchanged, and modified across all of these approaches. As a result, there is no longer any scientific basis for discriminating between evolutionary approaches. In light of the current state-of-the-art, we'll adopt the term *evolutionary algorithm* to describe any of the algorithms that use population-based random variation and selection throughout the remainder of the text, and we encourage you to think in these broad terms as well.

Let's summarize some of the ideas discussed in earlier sections of this chapter and point out a few additional possibilities. To begin, we want to emphasize that evolutionary algorithms may incorporate any representation that appears suitable for the task at hand, whether it's a binary string as suggested in the SAT problem, a permutation of integers as offered for the TSP, or a vector of floating-point numbers as we saw in the NLP. Of course, we're not limited to these representations. We could use matrices, graphs, finite state machines, and other more complex data structures to represent potential solutions. In particular, individuals may represent a set of rules (having a variable-length structure), a blueprint of some complex design, or even a computer program. Indeed, the possibilities are limitless!

Note, however, that the choice of variation operators, which are responsible for changing parents into offspring, depends strongly on the chosen representation. We have already seen (section 6.1) a flip-mutation, which flips the value of a bit, and the so-called "1-point crossover," two operators that have commonly been applied to binary representations. We have also looked (section 6.2) at a swap or reversal operator for integer representations together with special crossover that builds a permutation of integer numbers out of two distinct permutations. We also described (section 6.3) a Gaussian mutation for floating-point representations.

Despite the many differences in approach to these problems, the implementation of an evolutionary algorithm was seen to be fairly easy. Whether or not the problem was an SAT, TSP, or NLP, the recipe was essentially the same: (1) create a population of individuals that represent potential solutions to the problem at hand, (2) evaluate them, (3) introduce some selective pressure to promote better individuals or eliminate those of lesser quality, and (4) apply some variation operators that generate new solutions to be tested. Then repeat the evolutionary loop of evaluation, selection, and variation (steps 2–4) several times.[5]

There are a few other interesting observations to make. We won't provide a detailed analysis here, but rather a few remarks. First, the broadly encompassing approach of the evolutionary technique should be apparent. For example, we can argue that simulated annealing is a special case of an evolutionary algo-

[5]Alternatively, the loop can proceed with variation, selection, and evaluation, depending on the particular implementations.

rithm, where the population size is limited to a single individual, the variation operator relies on a single parent, and the selection procedure is based on an extrinsic parameter called *temperature*. In chapter 10 we'll discuss evolutionary algorithms that change the scope or rates of their mutation operators, and this connection between simulated annealing and evolutionary algorithms will be made more directly.

In contrast to annealing, tabu search is based on very different concepts, but even there, it's possible to extend an evolutionary algorithm using the concept of memory and then tabu search becomes a special case. It's relatively easy to imagine adding either a memory unit to each individual (e.g., in terms of adding additional parameters to the represented structure and some interpretation rule that would operate on these additional parameters to guide the search; see also chapter 11), or a global memory (like a library of past ideas) for the whole population (e.g., cultural algorithms [388]). Evolutionary algorithms can in fact incorporate the main concepts underlying tabu search quite naturally.

In chapter 3, we argued that one particular classification of optimization algorithms can be based on how an algorithm "perceives" the search space. One class of algorithms (e.g., greedy algorithms, dynamic programming, branch and bound, A^*) evaluates subspaces, rather than individual solutions. Algorithms in the other class (e.g., hill-climbers, simulated annealing, tabu search) consider the whole search space as a uniform set of potential solutions. Only single complete solutions are evaluated and there is no concept of a subspace.[6] On the other hand, evolutionary algorithms may combine these two categories by allowing individuals to describe subspaces as well as particular solutions.

There are also various additional possibilities to explore. Many of these are discussed in length later in this text, but we will mention some of them right here to give you a better perspective on this exciting field!

Most algorithms require a setup of some parameters (e.g., to apply simulated annealing, you have to fix the starting temperature, cooling rate, the maximum number of iterations per given temperature, etc.). Similarly, evolutionary algorithms require a few parameters as well. These parameters include the population size, probabilities (rates) of various operators, a parameter to regulate the selective pressure of the system, penalty coefficients (if the penalty function approach is used for solving a constrained optimization problem), and others such as the mutation step size, the number of crossover points, etc. The values of such parameters may determine whether or not the algorithm will find a near-optimum solution and, what's also important, whether or not it will find a reasonable solution efficiently. Choosing effective parameter values, however, is a time-consuming task and considerable effort has gone into developing good heuristics for these choices across many problems.

Recall, though, that evolutionary algorithms implement the idea of evolution, and that evolution itself must have evolved to reach its current state of sophistication. It is thus natural to expect adaptation to be used not only for finding solutions to a problem, but also for tuning the algorithm to the par-

[6]Except of course a neighborhood, which constitutes a subspace.

ticular problem. Technically speaking, this amounts to modifying the values of parameters during the execution of the algorithm taking the actual search process into account. As discussed later in the text, the issue of controlling the values of various parameters of an evolutionary algorithm has the potential of tuning the algorithm to the problem while solving the problem!

Further, many real-world problems require multiple solutions. This is common when there are multiple objectives to meet and it is difficult to aggregate them with a normalizing function. It would be great if we could have an algorithm that returned more than one good solution! Having a few alternatives to choose from is often very useful!

Evolutionary algorithms can be adapted easily to meet this new challenge. It's possible to utilize feedback on the diversity of the population, and in particular, on the presence of other individuals in a neighborhood of a chosen individual. Such feedback, duly incorporated into the evaluation function, can be used to control the number of returned solutions.

In chapter 1, we mentioned that many problems require time-adaptive solutions. We often get a new piece of information that contributes to the evaluation function while solving the problem. This might occur, for example, while traversing the best tour for the TSP. Perhaps we discover that, due to unforeseen circumstances, a particular segment of the tour is no longer available. Or while scheduling a factory, we learn that one of the machines has broken down. These unexpected events occur all the time in the real world, and they demand quick adaptation of current solutions to meet the new challenges.

A vast majority of classic optimization algorithms assume a fixed evaluation function. Any change in the evaluation function requires restarting the algorithm from scratch! In contrast, evolutionary algorithms are inherently adaptive; individuals in the population *adapt* to the current environment, and the quality of that adaptation is measured by the evaluation function. Consequently, an unexpected change in the model, or even in the evaluation function itself, can cause a short-lived disturbance of the evolutionary process, but after some period of time, which varies by problem, individuals in the population can adjust to the requirements of the new setting. Evolutionary techniques don't require restarting every time something changes. Instead, they continue their adaptation in light of similarities between previous experience and current demands. This illustrates a great potential of evolutionary algorithms.

Another advantage of the evolutionary approach to problem solving is that it's often easy to hybridize evolutionary algorithms with other methods. Special variation and selection operators that do the work of hill climbing or other methods can be incorporated into the process directly.

Here's another possible advantage. Recall our discussion from chapter 1, section 1.3 concerning the owner of a supermarket chain who needs to decide where to place a new store, while at the same time, his competitors are making their decisions as well. Of course, they are trying to maximize their profits, in turn quite likely minimizing the profits of the owner of the supermarket chain. Thus, the proper decision making process is very complex (he thinks that they

think that he thinks that they think...), but we can model this situation by running two opposing evolutionary processes. The first evolutionary algorithm optimizes the strategy for the supermarket chain's owner, while the other evolutionary algorithm handles the opponent's strategies. Note that the evaluation of each side's strategy is on the basis of the current (or perhaps projected) strategy of the opponent. This type of evolutionary competition is called a *coevolutionary model.* These sorts of models can often be used to discover solutions to problems that appear too unwieldy to describe in precise mathematics due to the joint relationship of the players.

Another item should be mentioned while discussing advantages of evolutionary algorithms: parallelism. Parallelism promises to put within our reach solutions to problems that were heretofore intractable. Evolutionary algorithms are very suitable for parallel implementations as they are explicitly parallel "in nature"! It's relatively straightforward to assign a processor to each individual in the population, or to split a population into several subpopulations (the latter can be enhanced further by adding the concept of migration of individuals from one subpopulation to another).

While listing potential advantages of evolutionary algorithms, it's worthwhile to mention the final "argument," which is related more to the psychology of programming than the efficiency of the approach. When running an artificial evolutionary process, you are playing the role of *creator*, and you determine the rules of the simulation, much like determining the physics of the real world. You may choose to pay close attention to specific mechanisms in natural evolution, or perhaps consider things not found in biological systems (e.g., mating more than two parents). You can even violate the laws of nature completely and introduce *Lamarckian evolution*, where individuals acquire characteristics during their "lifetime" and then pass those characteristics along to their offspring. The barrier between the *phenotype* (an individual's expressed behavior) and the *genotype* (an individual's genetic composition) doesn't have to exist for you!

You might introduce the concept of gender among the individuals, assign them an age (some number of generations), and then use this to determine how long they might survive. You might include a memory in an individual, or explore the concept of families of individuals and social learning. You could even put entire populations in competition with each other and perhaps migrate solutions between populations. You might explore including the *Baldwin effect* in which individuals learn during their lives and in doing so can affect their fitness and alter the course of evolution. All of these avenues lead to other choices. For example, which solutions should migrate: the best, the average, the worst? Indeed, the possibilities open to you as a programmer and problem solver are virtually unlimited!