CHAPTER 13

Collective Intelligence and Other Extensions of Evolutionary Computation

Chapters 10–12 have introduced the basic concepts of evolutionary algorithms, particularly as they are applied for optimization. In this chapter, we'll cover other aspects of evolutionary algorithms and methods that are related to simulating evolution on computers.

It's important to understand upfront that there are very many extensions of evolutionary algorithms. This text is intended to provide an introduction to the field of evolutionary computation that would be suitable as part of a broader introduction to computational intelligence within the framework of a one semester college course. By consequence, we cannot cover everything here and do it in any level of depth that would be appropriate.

This chapter begins with a population-based optimization approach called particle swarm optimization (PSO), which models the flocking behavior we see in certain animals. We then focus on another population-based approach called differential evolution, which searches a landscape for optima by using different vectors between existing solutions. Another approach follows that is based on modeling how ants search and find food sources. It turns out that the strategy ants use can be applied for finding short paths through graphs and other engineering problems.

After that, we'll focus on specific applications of evolution that are found in hardware, as opposed to only in software. We'll also address problems for which having a person act as the fitness function provides possibilities that are beyond what evolutionary algorithms can do solely in software. Finally, we'll conclude with the application of evolutionary algorithms to multiple criteria optimization problems.

13.1 PARTICLE SWARM OPTIMIZATION

When considering how a flock of birds or a school of fish moves, it's easy to think that the actions of each individual are somehow coordinated with the actions of the others, and that's exactly the case. Research shows, for example, that starlings coordinate

Fundamentals of Computational Intelligence: Neural Networks, Fuzzy Systems, and Evolutionary Computation, First Edition. James M. Keller, Derong Liu, and David B. Fogel.

their movements based on seven neighboring birds [Young *et al*., 2013]. Thousands of starlings may form a flock, but the movements of the flock are based on the local interactions of overlapping groups of just seven birds.

It's interesting that modeling the way birds flock, or fish school, or things in general "swarm" can lead to an optimization algorithm. The organisms that we might model in these cases are likely optimizing something, such as group cohesiveness versus individual effort [Young *et al*., 2013]. That might not correlate to what we'd like to do in terms of using swarming as a method to find points of interest on an objective function, but we can tailor a swarming method toward that end.

Particle swarm optimization was introduced as such an approach [Kennedy and Eberhart, 1995]. Suppose you have a collection of possible solutions to a problem, which we'll call *particles*. The particles reside in $\Re^n$ and the goal is to find the minimum, maximum, or some other point of an objective function $f(\mathbf{x})$, where $\mathbf{x} \in \Re^n$. Much like the solutions in the population of an evolutionary algorithm, these particles are located at various positions and can be evaluated in light of some objective criteria. Each particle is denoted by $\mathbf{x}_i(t)$, where $\mathbf{x}_i$ is the ith particle located at $\mathbf{x}$ (a vector) at a particular time t.

Next comes the swarming part: How do the particles move? They each have a velocity vector, denoted by $\mathbf{v}_i(t)$, where $\mathbf{v}_i$ is the velocity of the ith particle at a particular time t. If there were no swarming, each of the particles would move according to their unchanging velocity forever. The fact that the particles will swarm means that their velocities will change as a function of what is known about other particles in the collective, and also what a particle remembers about where it has been.

Each particle is given a memory. It remembers the location that has yielded the best result from the objective function. Each particle also has knowledge about the results of other particles in a neighborhood (akin to starlings being aware of other starlings), and each particle knows the location of the particle in its neighborhood that has the best result from the objective function. That information is used to change the velocities of the particles and thereby having them move to different locations, searching for a better location.

Each particle's new velocity is a function of (i) its current velocity, (ii) the vector that points to the particle's own best location, and (iii) the vector that points to the best location of the particles in its neighborhood. The vectors that point to known best locations are weighted by random variables. With starlings, the neighborhood is believed to be seven birds, but in PSO the neighborhood can be small, like a particle and its two closest neighbors, or it can expand to be the entire collection of particles. Each particle then moves according to its new velocity.

A little pseudocode can help clarify the update procedure for each particle:

$$\mathbf{v}_i(t+1) = a \times \mathbf{v}_i(t) + b \times U_1 \times (\text{PersonalBest}_i - \mathbf{x}_i(t)) + c \times U_2 \times (\text{NeighborhoodBest}_i - \mathbf{x}_i(t))$$

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1)$$

where U_1 and U_2 are distributed $U(0, 1)$, PersonalBest_i is the location where the ith particle found the best score in its memory, $\text{NeighborhoodBest}_i$ is the location where the best score was found in the ith particle's neighborhood of particles, and a, b, and c

are scaling terms. Some basic settings for the scalars are $a = 1$, and $b + c = 4$. The trade-off of b and c essentially weights the relative importance of a particle's own experience for the importance of its neighborhood's experience.

One issue that comes with these update equations is that the particles can see their velocities increase in magnitude without bound. That's not particularly realistic for modeling real flocking and empirically it's not helpful when searching for optima in our typical objective functions. So, a limit is placed on the velocity in each dimension, v_{max}. Some experimentation may be required to have a good setting for this maximum value.

Another issue that arises is the choice of the number of particles. Many publications have used collections of 10–50 particles, but the appropriate size is problem dependent, as is the choice of how to construct a neighborhood for each particle.

Here's an example that uses 50 particles with a neighborhood of 3 particles and $|v| = 1$ applied to the function $f(x, y) = x^2 + y^2 - 20[\cos(\pi x) + \cos(\pi y) - 2]$, which is depicted in Figure 13.1. The global minimum value of the function is $f(x, y) = 0$ and the second-best minimum value is $f(x, y) = 4$.

In this example, all the particles were initialized uniformly at random between -10 and 10 in each dimension. After 500 iterations, the best score found was 0.0355. Figure 13.2 shows the best score of all the particles at each generation and the average of each particle's historical best score. The graph shows continual improvement toward the global optimum.

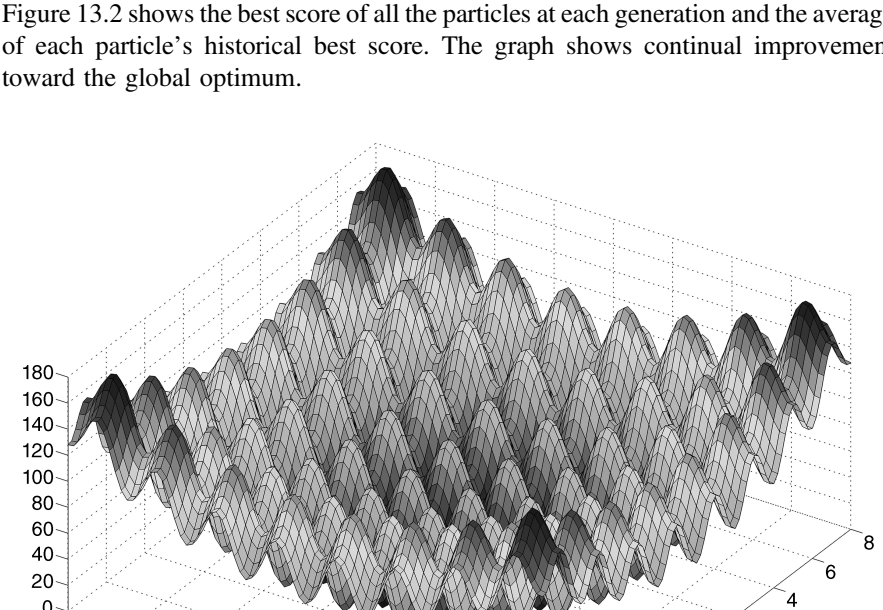

FIGURE 13.1 A surface with multiple local optima used for testing the particle swarm optimization algorithm.

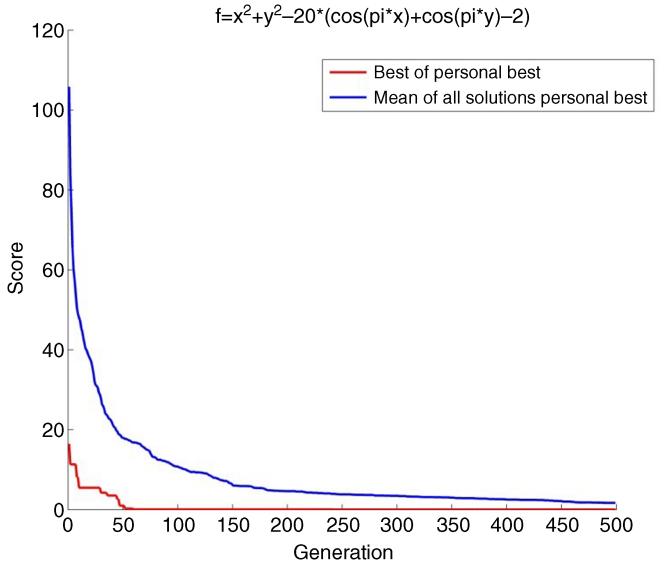

FIGURE 13.2 The rate of optimization for the best score ever found and the mean of all particles' best scores for the problem shown in Figure 13.1.

There are many areas for investigation in particle swarm optimization. For example, it may be that neighborhood size would benefit by varying by particle, or varying by particle over time. The learning factors that represent the weights on the acceleration terms can also be subject to online learning per particle. The velocity update equation can be modified by applying a "constriction" factor to the overall velocity, which relieves the need for setting a value of v_{max}. Other extensions include applications in discrete space and dynamic settings. Reviews can be found in Banks *et al*. [2007], Kiranyaz *et al*. [2013], and Kaveh [2014] with current research found annually in the IEEE Swarm Intelligence Symposium.

13.2 DIFFERENTIAL EVOLUTION

Another population-based search algorithm that relies on updating the location of the individuals in the population is called *differential evolution.* It was introduced in Storn and Price [1996], Storn [1996], and other publications about the same time as particle swarm optimization and is tailored to searching in real-valued spaces for maxima or minima of functions. Each of the individuals in the population is subject to a form of mutation and recombination, as well as selection.

The key ingredient in differential evolution is that individuals move based on the differential vectors from the individual to other individuals in the population. The population size needs to be at least four individuals and as with all evolutionary or related methods, the population is initialized at random or based on some prior knowledge of where to search for a solution to the problem posed.

Each of the individuals in the population is subjected first to mutation, which is based on the individual's relationship to three other distinct individuals in the population, chosen at random each time. Let's call these three individuals $\mathbf{x}_1$, $\mathbf{x}_2$, and $\mathbf{x}_3$, and let's call the individual that we are mutating $\mathbf{x}_0$.

First we pick a random dimension of the problem, d, uniformly from 1 to n, where the problem has n dimensions. We'll remember that dimension. Then, for each dimension $i = 1, \ldots, n$, we create a uniform random number $u_i \sim U(0, 1)$. If $u_i < p_c$ or $i = d$, then the new value of the solution in the ith dimension is given by

$$x_{0i} = x_{1i} + a(x_{2i} - x_{3i})$$

where a is a scalar value between [0, 2] called a *differential weight*; otherwise, the new value of the solution in the ith dimension is retained from x_{0i}. The value p_c is a "crossover probability," which really means that it is transplanting a value for that dimension from another random solution added with a weighted combination of a difference between two other solutions.

Finally, for this solution, if the new version of $\mathbf{x}_0$ is better than the original, it replaces the original in the next generation; otherwise the original $\mathbf{x}_0$ is retained for the next generation. This process continues until a solution of sufficient quality is found, or a maximum number of generations is met.

Let's see how this approach works on two simple functions. The first is the usual quadratic bowl defined by $f(x, y) = x^2 + y^2$ (Figure 13.3), and the second is the same one that we just saw in Figure 13.1, which has multiple local optima.

We'll use 50 solutions in the population in each case, with our scalar value $a = 0.8$ and $p_c = 0.02$, with all solutions initialized uniformly at random between -20 and 20.

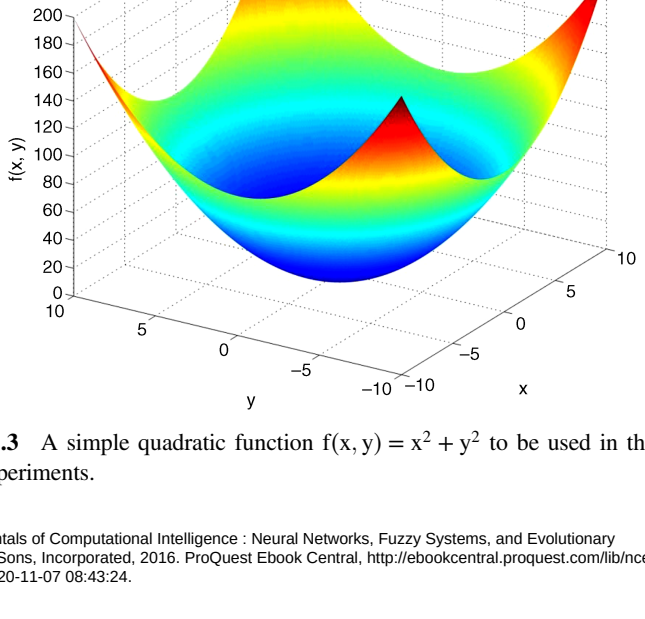

FIGURE 13.3 A simple quadratic function $f(x, y) = x^2 + y^2$ to be used in the differential evolution experiments.

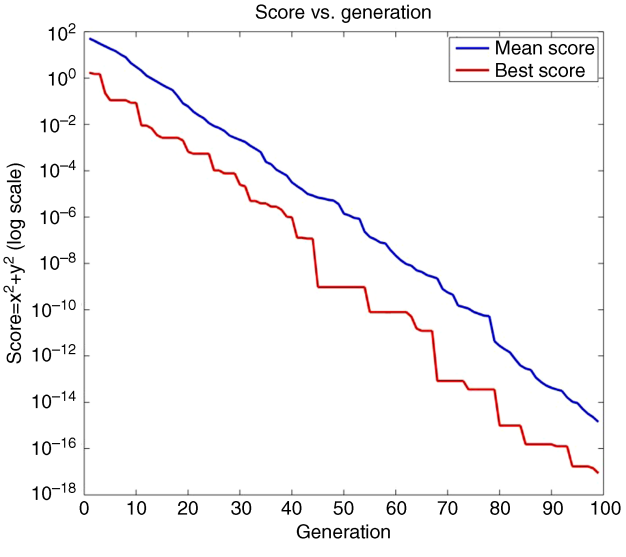

FIGURE 13.4 The average of all scores in the population (n = 50) and the best score in the population when applying differential evolution to the quadratic bowl. Note that the y-axis is on a log scale.

Figure 13.4 shows the average of all the scores in the population and the best score as a function of the number of generations for the quadratic bowl and Figure 13.5 shows these data when applying differential evolution on the multimodal surface.

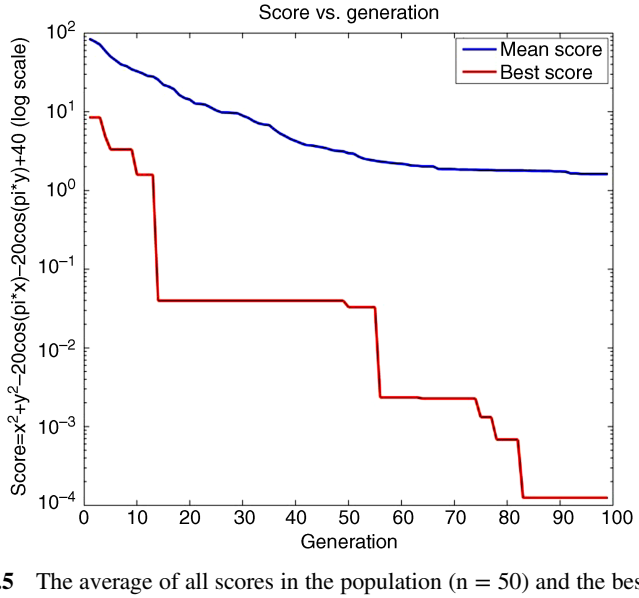

FIGURE 13.5 The average of all scores in the population (n = 50) and the best score in the population when applying differential evolution to the multlimodal function $f(x, y) = x^2 + y^2 - 20[\cos(\pi x) + \cos(\pi y) - 2]$. Note that the y-axis is on a log scale.

For the quadratic bowl, the best solution has less than 10^{-6} error in about 40 generations. For the multimodal function, the method is still able to find a solution that has less than 10^{-4} error in about 80 generations. A quick comparison with the results using PSO on the same multimodal function favors the use of differential evolution; however, it's always important to remember that this may not generalize to other functions, and may be sensitive even simply to the initialization.

13.3 ANT COLONY OPTIMIZATION

Another biologically inspired method for solving problems is called *ant colony optimization* (ACO), and it simulates how ants discover food sources and communicate their discoveries with other ants. ACO predates PSO and differential evolution [Dorigo, 1992] and is used generally for combinatorial optimization problems.

Here's the thinking underlying the approach. Suppose you're an ant and you have to get from your colony's nest to a food source. The problem is that you don't know where the food source is. So you have to search for it. Once you find it, you have to get back home, so you have to search for that too.

It would be great if you could share what you learn as you search with other ants, so that they could search more efficiently. Nature provides a way to do that. It's called a pheromone trail, and ants leave pheromone trails as they move. The strength of the trail is dependent on the number of ants that traverse the trail and how recently the ants have traversed the trail.

More ants on a trail means more pheromone, which entices other ants to follow that trail rather than search somewhere else. The pheromone evaporates over time, so once food sources are exhausted, the ants don't continue to travel to an "empty refrigerator" forever.

Let's see how we can model these principles to address a classic 30-city traveling salesman problem. The ants will start at a particular city and they have to find a complete path that visits every other city once and only once and then returns home. The objective is to complete the path with the shortest possible distance. Let's say we have a set of 50 ants in our population.

It's time for the first ant to start its trek. The probability of it visiting any of the other available city is given by

$$\mathrm{p(i,j)} = \frac{\mathrm{ph(i,j)} \times \mathrm{cost(i,j)}^{-1}}{\sum_{\mathrm{k} \in \mathrm{All}} \mathrm{ph(i,k)} \times \mathrm{cost(i,k)}^{-1}}$$

where $\mathrm{p(i,j)}$ is the probability of going from city i to city j, $\mathrm{ph(i,j)}$ is pheromone factor between city i and city j, and $\mathrm{cost(i,j)}$ is the distance between city i to city j. When starting, we set the value of $\mathrm{ph(i,j)}$ to 1.0 for all pairs of cities.

From the formula, the probability that the ant will go to city j is dependent in part on the pheromone factor. The higher the value of $\mathrm{ph(i,j)}$, the higher the likelihood of traveling to city j. Also, the probability is inversely related to the distance between the

current city i and the new city j. Each ant wanders through the cities in traveling salesman problem according to the probability equation, which is computed for all available (unvisited) cities at each step through the path for each ant.

The pheromone factor can be updated in different ways. For our example, let's say we update the pheromone factor after each of the 50 ants has traversed a tour of the cities. The new pheromone strength is computed as

$$\text{new ph}(i,j) = \alpha \text{ph}(i,j) + \Delta \text{ph}(i,j)$$

where α is the evaporation rate and $\Delta \text{ph}(i,j)$ is the delta increment that comes from ants going from city i to city j. Here, let's say that the evaporation rate $\alpha = 0.9$ and

$$\Delta \text{ph}(i,j) = \sum_{k=1}^{50} \begin{cases} Q, & \text{if kth ant traveled between city i and city j} \\ 0, & \text{else} \end{cases}$$

and $Q = 1/50$ (i.e., the inverse of the number of ants).

After each set of 50 ants traverses the graph, the pheromone factors are updated, encouraging the ants to favor shorter paths that have been well traveled.

Here are some results from this approach on a 30-city traveling salesman problem. Figure 13.6 shows the best path found after 800 iterations of the 50 ants traversing the graph. The problem was created by distributing the cities uniformly at random in a 100×100 square. The final best path is 411.41 units long. Statistical mechanics can be used to estimate the optimum path length for n randomly distributed cities in an

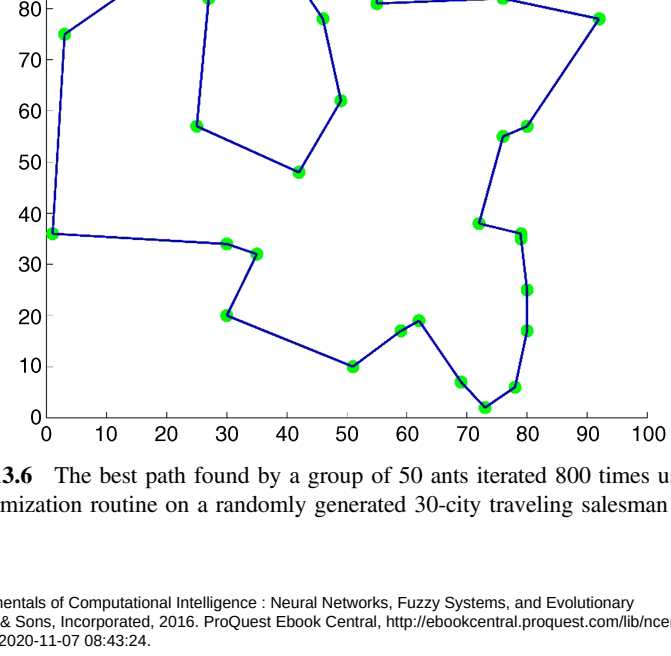

FIGURE 13.6 The best path found by a group of 50 ants iterated 800 times using the ant colony optimization routine on a randomly generated 30-city traveling salesman problem.

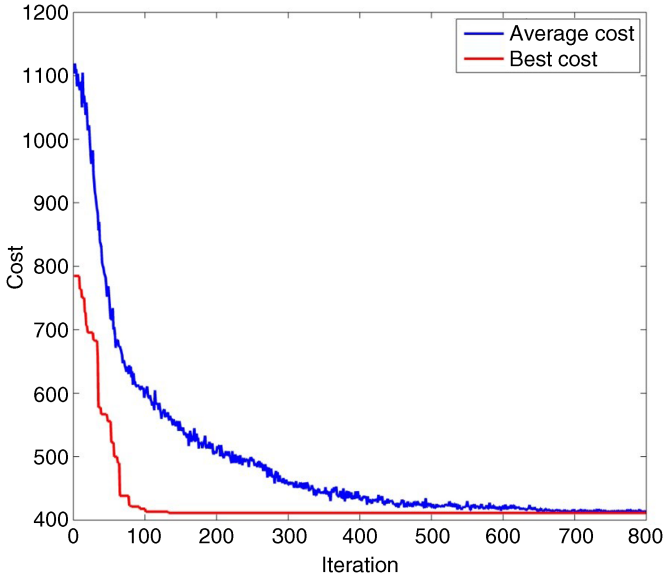

FIGURE 13.7 The average cost over each of the 50 ants at each iteration and cost of the best path found when iterating 800 times using the ant colony optimization routine on the traveling salesman problem shown in Figure 13.6.

area A by the formula $0.749 \times (\text{nA})^{0.5}$ [Bonomi and Lutton, 1984]. In this example, the expected best solution is 410.24. So, the solution is close to the expected best tour length.

Figure 13.7 shows the rate of optimization of the best and average cost of tours found during the search. By about the 100th iteration, the best solution already has nearly the expected best score. By the 800th iteration, the average tour length across all 50 ants is also close to this score, indicating that the search has converged.

Ant colony optimization has been applied to a diverse set of engineering problems. For example, the method has been employed to construct neural network topologies [Salama and Abdelbar, 2014], design type-2 fuzzy controllers [Hsu and Juang, 2013], induce decision trees [Otero *et al.*, 2012], perform fingerprint analysis [Cao *et al.*, 2012], and many other application areas. For reviews, see Dorigo and Gambardella [1997] and Dorigo and Stutzle [2004].

13.4 EVOLVABLE HARDWARE

We've spent a lot of time addressing specific implementations of evolutionary algorithms or other variations of evolutionary computing, all in software. But it's interesting to consider implementing evolutionary principles in the design of hardware too.

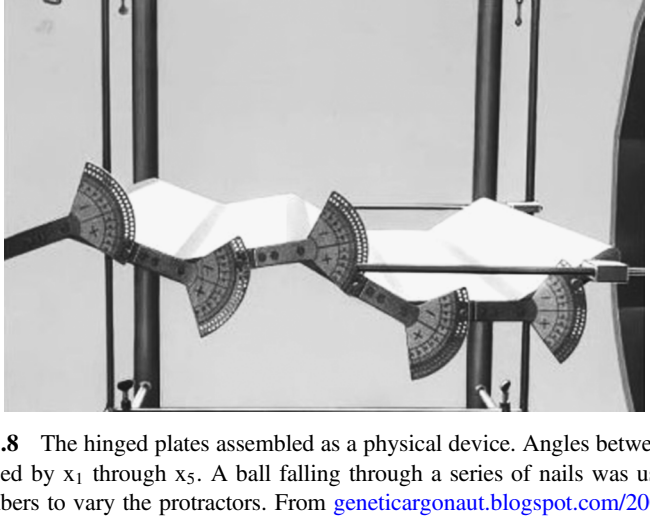

FIGURE 13.8 The hinged plates assembled as a physical device. Angles between the plates were described by x_1 through x_5. A ball falling through a series of nails was used to create random numbers to vary the protractors. From geneticargonaut.blogspot.com/2006/03/evolutionary-computation-classics-vol.html.

Some of the earliest experiments in evolutionary computation accomplished by a group in Berlin, Germany, involved manipulating physical devices [Rechenberg, 1973]. This was performed by constructing a device, such as series of planes attached at hinges with measured protractor settings (Figure 13.8), and then throwing dice or using a pachinko-like device to generate a random number, and making random changes to the physical structures.[1]

Scoring the set of hinged plates was accomplished in a *de facto* wind tunnel, where the drag of the plates was measured using a pitot tube. The objective was to adjust the angles between the plates such that the overall set of plates would have minimum drag. This method of throwing dice and making adjustments based on the random rolls, while retaining the best-found configuration as the next starting point, was able to find the series of flat plates that provides minimum drag.

More complicated experiments involved evolving the mechanical design of a bent pipe and a nozzle that would offer maximum energy efficiency of fluids flowing through the devices. These experiments are reviewed in Fogel [1988].[2]

More recently, experiments in Lohn *et al*. [2015] describe the evolution of S-band omnidirectional and medium gain antennas, which were evolved in simulation and

[1] Ingo Rechenberg and Hans-Paul Schwefel worked in tandem, with a third contributor Peter Bienert, at Technical University of Berlin starting in 1963–1964. The first computer available to the group was a Zuse Z23 that arrived in 1965. The experiments in hardware evolution were conducted in real hardware primarily because the researchers conducting the experiments had no electronic computers available to model the dynamics of the systems and execute the algorithms in software. In addition, whether or not all of the relevant physics in more complicated structures were known was uncertain. Thus, performing experiments *in situ* provided real validation of the physics of the system (I. Rechenberg (1996) personal communication to D. Fogel at Technical University of Berlin, Germany).

[2] Also see http://geneticargonaut.blogspot.com/2006/03/evolutionary-computation-classics-vol.html for a video of the evolution of the flashing nozzle created by H.-P. Schwefel.

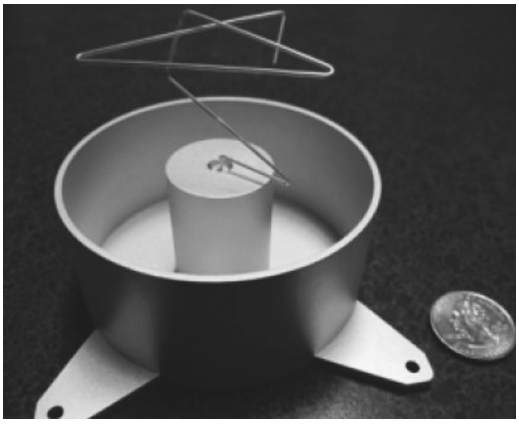

FIGURE 13.9 The final prototype of the S-band antenna from Lohn *et al.* [2015].

then verified in real-world settings. Figure 13.9 shows a sample design of the antenna, which is unconventional. The final evolved designs were launched into space as part of NASA's Lunar Atmosphere and Dust Environment Explorer (LADEE), which launched on September 6, 2013 and orbited the moon from October 2013 through April 2014. The antennas provided "65% increased downlink coverage and 44% cost savings for the mission" [Lohn *et al*., 2015]. For other related work, see Homby *et al*. [2011].

Other interests in evolving hardware have focused on adapting field programmable gate arrays (FPGAs) [Thompson, 1996, 1998; Stoica *et al*., 2003] and recovering from faults that may occur [Greenwood, 2005]. Additional research has been performed in evolving electronic circuits in simulation [Koza *et al*., 1996; Vasicek and Sekanina, 2014]. While a more in-depth treatment of evolving hardware is beyond the scope of this text, interested readers should see Greenwood and Tyrrell [2006] for more information.

13.5 INTERACTIVE EVOLUTIONARY COMPUTATION

Sometimes it's not very easy or even impossible to know how to write a fitness function in a mathematical equation. For example, suppose you wanted to use an evolutionary algorithm to create a piece of artwork or music. What fitness function would you use?

There have been some thoughts on this. For example, symmetry in artwork can be aesthetically pleasing [Fogel, 1992]. Certain known chord progressions resonate with our expectations when listening to music [Fukumoto, 2014]. These and other aspects might be measureable and therefore could be put in a heuristic for scoring a particular drawing or musical composition.

But, as the cliché goes, sometimes beauty really is in the eye of the beholder. In such cases, it's possible to use a human's judgment as the fitness function and guide

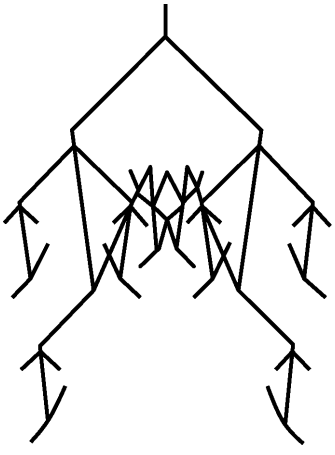

FIGURE 13.10 A starting biomorph described as a "frog" at www.emergentmind.com/biomorphs

evolution to solutions that fit the individual's judgment. The result is something called *interactive evolutionary computation*, because it involves the interaction of the human operator for the scoring function.

One early example is found in the *Biomorphs* program introduced in Dawkins [1986]. The program is replicated presently at www.emergentmind.com/biomorphs so that you can try it for yourself. The program provides you nine different stick figure objects. You can guide an evolutionary search to see if you can create various shapes.

One of the authors (D. Fogel) tried this and started with the frog-like figure shown in Figure 13.10. Over a succession of 25 generations, he was able to create the rocketship-like drawing shown in Figure 13.11. His goal was the prototype spaceship shown in

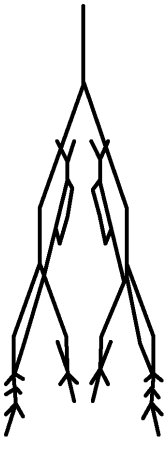

FIGURE 13.11 The biomorph D. Fogel evolved over 25 generations with the idea of making the image look like a rocketship.

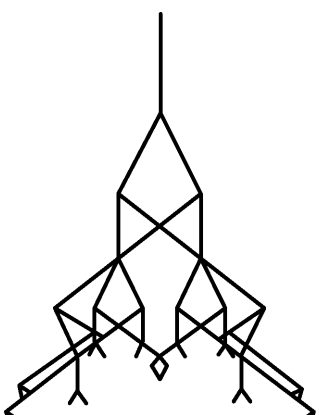

FIGURE 13.12 The image that D. Fogel had in mind while adapting the original biomorph from Figure 13.10 over 25 generations. This prototype image was provided at the Web site as an example of a possible outcome.

Figure 13.12. Although the fit between the two images is not exact, the image in Figure 13.11 is qualitatively closer to that of Figure 13.12 than is Figure 13.10.

In this same vein, one of the first practical applications of interactive evolutionary computation was in area of police sketch artistry [Caldwell and Johnson, 1991], where someone can say whether or not a given face looks more or less like the suspect he/she has in mind and the algorithm can adjust that face until it comes close to the person's mental image.

Interactive evolutionary algorithms have been used in designing ergonomic systems [Brintrup *et al*. 2008], portions of video games [Walsh and Gade, 2010; Cardamone *et al*. 2011], personal hearing aids [Takagi and Ohsaki, 2007; also see Fogel and Fogel [2012]], and even finding a nice blend of coffee [Herdy, 1996]. For a review of many other examples, see Takagi [2001].

13.6 MULTICRITERIA EVOLUTIONARY OPTIMIZATION

The final topic to cover involves finding solutions that satisfy multiple criteria. This is often the case in real-world problem solving, where a solution may be measured in multiple ways. For example, a financial asset management algorithm might be measured in terms of the return on investment, but also in terms of its volatility (such as the annualized standard deviation of its monthly returns). In this case, we'd like the ROI to be high, but we also want the volatility (which is a way of measuring risk) to be low.

One approach to handling multiple criteria is to combine them in a single utility function that returns a real value. For example, we could say that the value of the asset management algorithm was determined by

$$f(\mathbf{x}) = ax_1/x_2$$

where x_1 is the ROI, x_2 is the volatility, and a is a scaling constant. As ROI increases, so does $f(\mathbf{x})$. As volatility decreases, $f(\mathbf{x})$ increases. So, the best values of $f(\mathbf{x})$ will represent some trade-off between ROI and volatility.[3]

A different approach to handling multiple criteria involves finding solutions that are not "dominated" by other solutions. A solution dominates another when it is equally good or better in all measurable criteria, but better in at least one criterion. There may be many nondominated solutions for a given multicriteria problem.

The entire set of these solutions is called the *Pareto set*.[4] This is the set of solutions for which, for any solution in the set, no objective can be improved without reducing performance with respect to a different objective. It's sometimes of interest to find the entire Pareto set, or approximate it as closely as possible.

Evolutionary algorithms have been applied to this problem for many years [Fonseca and Flemming, 1993; Zitzler and Thiele, 1998; Knowles and Corne, 1999]. One interesting approach to the problem is called NSGA-II [Deb *et al*., 2002], which stands for nondominated sorting genetic algorithm.

The procedure starts by looping over all individuals x_i, $i = 1, \ldots, k$ in the population. Each solution begins with a set S_i initialized to the empty set and a number n_i set equal to 0. The set S_i ultimately contains all the individuals that are dominated by x_i. The number n_i is the number of individuals in the solution that dominates x_i. When the loop is complete, every individual that has $n_i = 0$ belongs to the "first front" (another set) and its rank $r_i = 1$.

After determining the members in the first front (presuming it's not empty), a procedure is implemented to determine the individuals to be stored in the second front. For each individual x_i in the first front, a loop is performed over all individuals in the corresponding S_i (which is the set of individuals dominated by x_i). For each individual j in S_i, its value n_j is decremented. If that value is zero, then that individual is assigned to the second front. Once that loop is completed, the second front is also complete. The procedure is repeated until the subsequent front remains empty.

Next, the NSGA-II algorithm works to determine the crowding of individuals within each front. Initially, all crowding scores are set to zero. Then, for each objective function m, individuals are sorted based on the objective. Individuals at the extremes of each front are assigned a value of positive infinity so that they will always be propagated into the next generation. In between the extremes, the crowding scores are incremented for each individual based on the Euclidean distance between each pair of neighboring individuals (within in front) across all of the objectives. The crowding procedure sets up a tournament selection function, which is based on nondomination and crowding criteria.

Selection can be implemented to fill the population for each next generation from each front in turn (that is, first from the first front, then from the second front, and so on) until the number of solutions in the remaining front would exceed the population

[3] A similar form of measure is called the Sharpe Ratio, which is computed as (ROI–RFR)/STD, where ROI is the return on investment, RFR is the risk-free rate of return, and STD is the standard deviation of the return on investment. In essence, the ratio measures how many standard deviations above the risk-free rate of return the asset is producing.

[4] Named after Vilfredo Pareto (1848–1923).

size. Solutions from this last front can be selected at random or based on crowding criteria.

New offspring solutions can be created from the new parents by any appropriate evolutionary algorithm. For example, here is an implementation of a 50 parent–50 offspring NSGA-II procedure on the problem:

Find the Pareto front for x_1, x_2, x_3 for the two criteria:

$$f_1 = 1 - \exp\left(-\sum_{i=1}^{3}\left(x_i - \frac{1}{\sqrt{3}}\right)^2\right)$$

$$f_2 = 1 - \exp\left(-\sum_{i=1}^{3}\left(x_i + \frac{1}{\sqrt{3}}\right)^2\right)$$

The solutions were initialized at random in the range $[-5, 5]$. Mutation was based on a fixed Gaussian random variation of a normal random variable with zero mean and standard deviation of 1 added to each dimension of a parent.[5]

For this problem, we can compute the actual Pareto front before we start the evolution, so we can determine how well the NSGA-II algorithm can fit the Pareto front. The equation of the Pareto front is

$$A^2 - 2AB + B^2 + 8A + 8B + 16 = 0$$

where

$$A = \log(1 - f_1)$$

$$B = \log(1 - f_2)$$

with f_1 and $f_2 \in [0, 1 - e^{-4}]$.

The population after generation 10 is shown in Figure 13.13. By generation 30, you can see the population starting to converge toward the Pareto-optimal front (Figure 13.14). At generation 70, the front is well estimated (Figure 13.15) and there are only four solutions remaining in the second front. By generation 100, all solutions are in the first front and the process has evidently converged close to the correct answer (Figure 13.16).

Remember that this problem and the approach are provided only for illustration. The variation operator here was not constructed to converge quickly. It used a fixed step size, with a standard deviation that was clearly larger than optimal given the distribution of solutions shown in Figure 13.16. Still, the NSGA-II process was able to guide the simple evolutionary algorithm to come close to the optimal Pareto set in only 100 generations.

[5] This was done to keep the illustration simple. Faster convergence could be expected by a more appropriate choice of mutation, as discussed in earlier chapters.

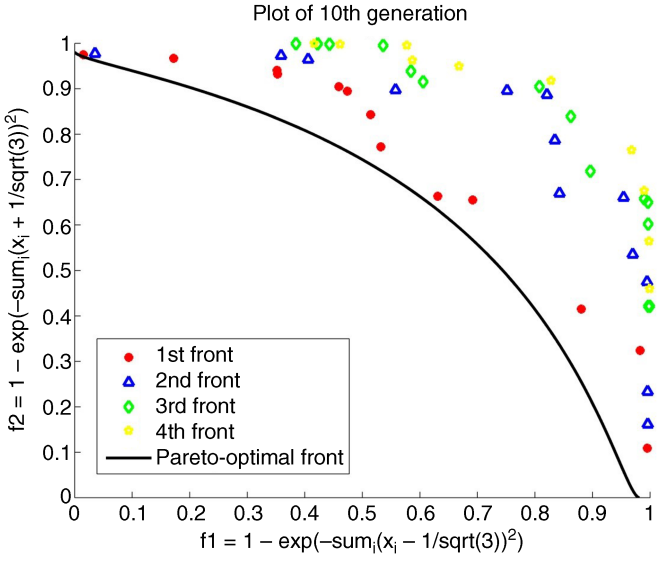

FIGURE 13.13 The population at the 10th generation on the multicriteria problem described in the text. The legend indicates which front each of the solutions is in. The solid black line represents the mathematical optimum solution for this problem.

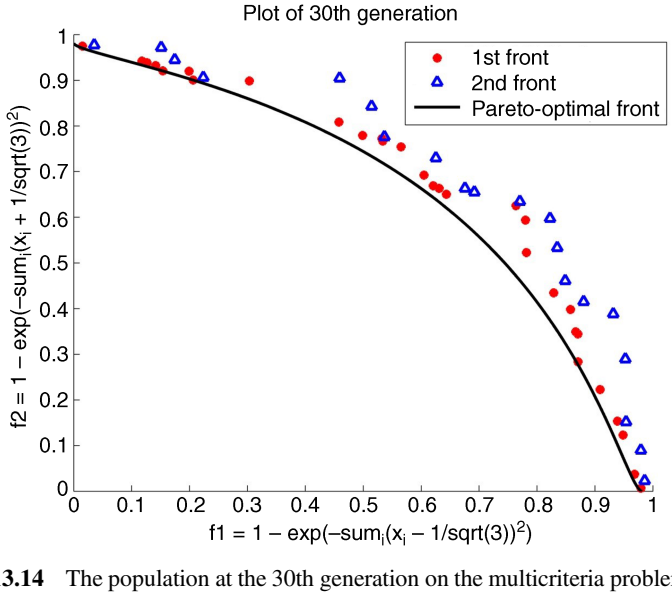

FIGURE 13.14 The population at the 30th generation on the multicriteria problem described in the text. Solutions in the population are now contained in one of two fronts. The solid black line represents the mathematical optimum solution for this problem.

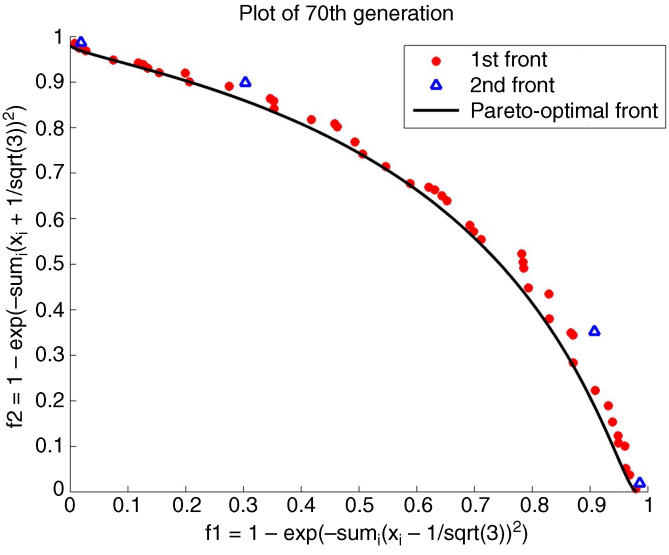

FIGURE 13.15 The population at the 70th generation on the multicriteria problem described in the text. Only four solutions remain in the second front. The first front is close to the solid black line that represents the mathematical optimum solution for this problem.

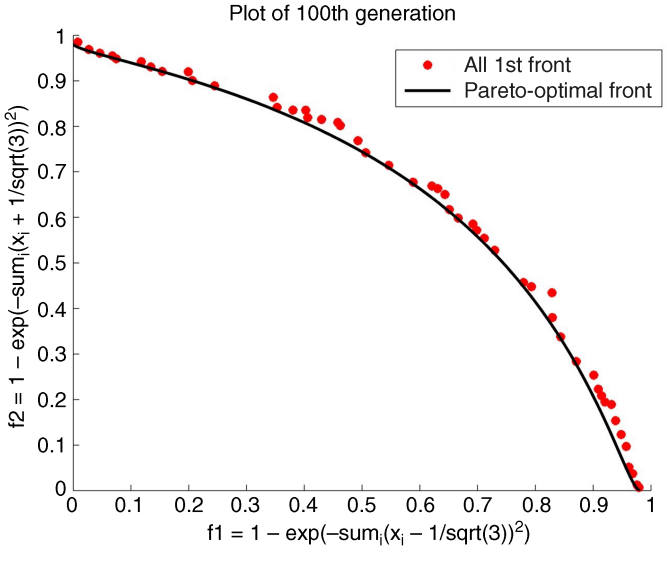

FIGURE 13.16 The population at the 100th generation on the multicriteria problem described in the text. All surviving solutions are in the first front, which has converged close to the solid black line that represents the mathematical optimum solution for this problem.

Readers are encouraged to review Deb *et al*. [2002] for implementing NSGA-II and Deb and Jain [2014] for an extension of NSGA-II, as well as Zhou *et al*. [2011] for a survey of multiobjective evolutionary algorithm techniques.

13.7 SUMMARY

This and the prior chapters on evolutionary computation are intended to introduce the main concepts of the field as part of a semester course on computational intelligence. We've covered many topics in these four chapters. But there's more to learn about, as there are many aspects of evolutionary computation that we haven't covered.

If you'd like to discover more about topics such as evolutionary search on constrained problems [Michalewicz and Schoenauer, 1996; Coello Coello, 2012], classifier systems [Wilson, 1995; Bull, 2015], cultural algorithms [Reynolds, 1994; Ali *et al*. 2014], memetic algorithms [Mei *et al*. 2011], evolutionary robotics [Lipson and Pollack, 2000; Nolfi and Floreano, 2000; Nouyan *et al*., 2009; Lipson, 2014], artificial immune systems [Dasgupta *et al*., 2011], artificial life [Conrad and Pattee, 1970; Ray, 1992; Ostman and Adami, 2013], and coevolution [Fan *et al*., 2014; Omidvar *et al*., 2014], please see the references included for you. Even when including those topics, that wouldn't cover everything in evolutionary computation. But as they say in statistical regression analysis: "You have to draw the line somewhere." And in evolutionary computation, that line is always evolving.

This chapter has presented several extensions of evolutionary computation that are based on modeling natural search mechanisms and other population-based search procedures. The chapter also has identified that sometimes it's helpful to evolve solutions in hardware or use a human operator to offer a judgment about the quality of evolved solutions. Finally, the topic of multicriteria optimization was introduced in terms of both optimizing against a single multiattribute utility (fitness) function and finding a Pareto set of solutions.

Together with the earlier chapters, these materials can provide you with a strong foundation for designing and applying evolutionary algorithms to help address your own challenging problems.

EXERCISES

13.1. Implement a particle swarm optimization algorithm to find the minimum of the function $f(x, y) = x^2 + y^2 - 20[\cos(\pi x) + \cos(\pi y) - 2]$ as shown in Figure 13.1. Then extend the function to be a function of n dimensions:

$$f(\mathbf{x}) = \sum_{i=1}^{n} x_i^2 - 20\left(\sum_{i=1}^{n}(\cos(\pi x_i) - 1)\right)$$

Estimate the convergence rate in terms of number of generations to reach a target level of the function for an initialization that you choose as a function of n for $n = 2, 3, 5,$ and 10.

13.2. Refer back to software you have written or use an available feed forward neural network simulator that employs sigmoid functions for the nodes and instead of using backpropagation, substitute a differential evolution algorithm to do the training. Test the results on data from http://archive.ics.uci.edu/ml/datasets/Wine, which provide 13 attributes about wine from three different wineries. The object is to have the neural network identify which winery made the wine based on the attributes. Compare your performance with differential evolution with that of backpropagation. Based on your results, what is your intuition about the relative convergence rate and reliability of the two approaches on these data?

13.3. Implement an ant colony optimization algorithm to demonstrate the double-bridge problem. The problem has ants that start on one side of a double bridge and must go across one of the two bridges to get to a food source. Start with one of the bridges at half the length of the other. How quickly does the population of ants converge to traversing the shorter bridge? Now increase the length of the shorter bridge and estimate convergence time as a function of the ratio of the two bridge lengths. For more details on the double-bridge experiment, see http://www.scholarpedia.org/article/File:SameLengthDoubleBridge.png.

13.4. Imagine that you wanted to design a possible improvement to the winglets that are now ubiquitous on commercial airplane wingtips. Winglets are a relatively recent invention, having been developed in the 1970s at NASA. Think about a series of experiments you could conduct using evolvable hardware that might assist in finding a new design to improve the lift and drag performance of a wing that has winglets.

13.5. Consider the problem of finding a personalized shoe for elite professional athletes. How could you use interactive evolutionary computation to help design such a shoe? Detail the steps and experiments you would conduct.

13.6. For the Pareto problem in Section 13.6, code the NSGA-II algorithm and see if you can replicate the results presented in the chapter to your own satisfaction. Next, increase the number of variables to four and five and rerun the algorithm. What is your expectation about the algorithm's run time until convergence as a function of the number of variables on this problem?