

Chapter 1

An Overview of Evolutionary Computation

©1996-2003 by Daniel Ashlock

Evolutionary computation is an ambitious name for a simple idea: use the theory of evolution as an algorithm. The field has many fathers and many names. A concise summary of the origins of evolutionary computation can be found in [5]. You may wonder how the notion of evolutionary computation could be discovered a large number of times without later discoverers noticing those before them. The reasons for this are complex and serve as a good starting point for this introduction.

The simplest reason evolutionary computation was discovered multiple times is that useless or hopeless techniques are not remembered. During the Italian Renaissance, Leonardo da Vinci produced drawings for machines, such as the helicopter, that did not exist as working models for centuries. The idea of taking the techniques used by life to produce diverse complex systems and use them as algorithms is a natural one. Fields like artificial neural nets and fuzzy logic also draw inspiration from biology. The problem is that, before the routine availability of extremely powerful computers, these biologically derived ideas were not too useful. Without big iron, even extremely simplified simulated biology is too slow for most applications.

Limited work with various levels of application and interest began in the 1950s. Sustained and widespread research in evolutionary computation began in the 1970s. In the late 1980s computer power and human ingenuity combined to create an explosion of research. Searching the world wide web with any of the keys, “Artificial Life,” “Evolutionary Computation,” “Genetic Algorithms,” “Evolutionary Programming,” “Evolution Strategies,” or “Genetic Programming,” will turn up vast numbers of articles. To get a manageable-sized stack, you must limit these search keys to specific applications or problem domains.

The second reason that evolutionary computation was discovered a large number of times is because of its interdisciplinary character. The field is an application of biological theory to

computer science used to solve problems in dozens of fields. This means that several groups of people who *never* read one another's publications can have the idea of using evolution as an algorithm. Early articles appear in journals as diverse as the *IBM Journal of Research and Development*, *The Journal of Theoretical Biology*, and *Physica D*. It is a very broad-minded scholar who reads journals that many floors apart in the typical university library. The advent of the world wide web has lowered, but not erased, the barriers that enabled the original multiple discoveries of evolutionary computation. Even now, the same problem is often attacked by different schools of evolutionary computation with years passing before the different groups notice one another.

The third source of the confused origins of evolutionary computation is the problem of naming. Most of the terminology used in evolutionary computation is borrowed from biology by computational scientists with essentially no formal training in biology. As a result, the names are pretty arbitrary and sometimes offensive to biologists. People who understand one meaning of a term are resistant to alternate meanings. This leads to a situation in which a single word, e.g., "crossover," describes a biological process and a handful of different computational operations. These operations are quite different from one another and linked to the biology only by a thin thread of analogy. A perfect situation for confusion over who discovered what and when they did so. If you are interested in the history of evolutionary computation, you should read *Evolutionary Computation, the Fossil Record*[13]. In this book, David Fogel has compiled early papers in the area together with an introduction to evolutionary computation.

As you work through this text, you will have ideas of your own about how to modify experiments, new directions to take, etc. Beware of being overenthusiastic: someone may have already had your clever idea; check around before trying to publish, patent, or market it. However, evolutionary computation is far from being a mature field, and relative newcomers can still make substantial contributions. Don't assume your idea is obvious and must have already been tried; being there first is a pleasant experience.

1.1 A Little Biology

The theory of evolution is central to this text. Evolution itself is dead simple and widely misunderstood. The theory of evolution is subtle, complex, and widely misunderstood. Misunderstanding of evolution and the theory that describes evolution flows not from the topic's subtlety and complexity, though they help, but from active and malicious opposition to the theory. Because of this, we stop at this point for a review of the broad outline of the biology that inspires the techniques in the rest of the text.

The first thing we need is some definitions. If you don't know what DNA is or want a lot more detail on genes, look in any standard molecular biology text, e.g., [26]. A *gene* is a sequence of DNA bases that code for a trait, e.g., eye color or ability to metabolize alcohol.

An *allele* is a value of a trait. The eye color gene could have a blue allele or a hazel allele in different people. We are now ready to define evolution.

Definition 1.1 Evolution *is the variation of allele frequencies in populations over time.*

This definition is terse, but it is the definition accepted by most biologists. The term *frequency* means “fraction of the whole,” in this case. Its precise meaning is the one used in statistics. Each time any creature is born or dies, the allele frequencies in its population change. When a blond baby is born, the fraction of blond alleles for some hair color gene goes up. When a man who had black hair in his youth dies, the frequency of black hair alleles drops. Clearly, evolution happens all the time.

Why, then, is there any controversy? The controversy exists partly because people who oppose evolution have never even heard the definition given here. Try asking people who dislike evolution what the definition of evolution is. If you do this, try to figure out where (and from whom) the person to whom you are talking learned their definition of evolution.

The main reason for the controversy surrounding evolution is that people dislike the logical conclusions that follow from the above definition juxtaposed with a pile of geological, paleontological, molecular, and other evidence. It is not evolution, but the theory of evolution, that they dislike. The *theory of evolution* is the body of thought that examines evidence and uses it to deduce the consequences of the fact that evolution is going on all the time. In science, a theory means “explanation” not “tentative hypothesis.” Scientific theories can be anywhere from entirely tentative to well supported and universally accepted. Within the scientific community, evolution is viewed as well supported and universally accepted.

Why mention this in what is, essentially, a computer science text? Because of the quite vigorous opposition to the teaching of evolution, most students come into the field of evolutionary computation in a state much worse than ignorance. They have often heard only myths, falsehoods, and wildly inaccurate claims about evolution. A wonderful essay on this problem is [11]. Since we will attempt to re-forge evolution into an algorithm, fundamental misunderstandings about evolution are a handicap. Let us start with the concept of fitness.

The following is utter nonsense if considered as biology: “Evolution is the survival of the fittest.” How do you tell who is fit? Clearly, the survivors are the most fit. Who survives? Clearly, the most fit are those that survive. This piece of circular logic both obscures the correct notion of fitness in biological evolution and makes it hard to understand the differences between biological evolution and the digital evolution we will work with in this text. In biology, the only reasonable notion of fitness is related to reproductive ability. If you have offspring that live long enough to have offspring of their own, then you are fit. Biologically, a Nobel prize winning Olympic triple-medalist who never has children is completely unfit. Consider a male praying mantis. As part of his mating ritual, he gets eaten. He does not survive. The female that eats him goes on to lay hundreds of eggs. A male praying mantis is, thus, potentially a highly fit non-survivor.

Oddly enough, “evolution is the survival of the fittest” is a pretty good description of many evolutionary computation systems. When we use evolutionary computation to solve a problem, we operate on a collection (population) of data structures (creatures). These creatures will have explicitly computed fitnesses used to decide which creatures will be partially or completely copied (have offspring). This fundamental difference in the notion of fitness is a key difference between biological evolution (or models of biological evolution) and most evolutionary computation. (Some sorts of evolutionary computation do use computer models of the biological notion of fitness, but they are a minority.)

Evolution produces new forms over time. This is clear from examination of the fossil record and from looking at molecular evidence or “genetic fossils.” This ability to produce new forms, in essence to innovate without outside direction other than the imperative to have children that live long enough to have children themselves, is the key feature we wish to reproduce in software.

How does evolution produce new forms? There are two opposing forces that drive evolution: variation and selection. Variation is the process that produces new alleles and, more slowly, genes. Variation can also change which genes are or are not expressed in a given individual. The simplest method of doing this is sexual reproduction with its interplay of dominant and recessive genes. Selection is the process whereby some alleles survive and others do not. Variation builds up genetic diversity; selection reduces it.

In biology, the process of variation is quite complex and operates mostly at the molecular level. At the time of this writing, biologists are learning about whole new systems for generating variation at the molecular level. Biological selection is better understood than biological variation. Natural selection, the survival of forms better adapted to their current environment, has been the main type of biological selection. Selective breeding, such as that which produced our many breeds of dogs, is another example of biological selection.

Evolutionary computation operates on populations of data structures. It accomplishes variation by making random changes in these data structures and by blending parts of different structures. These two processes are called *mutation* and *crossover* and together are referred to as *variation operators*. Selection is accomplished with any algorithm that favors data structures with a higher fitness score. There are many different possible selection methods.

Let’s consider the issue of “good” and “bad” mutations operating on a population of data structures. A good mutation is one that increases the fitness of a data structure. A bad mutation is one that reduces the fitness of a data structure. Imagine, for the sake of discussion, that we view our data structures as living on a landscape made of a vast flat plane with a single hill rising from it. The structures move at random when mutated, and fitness is equivalent to height. For structures on the plane, any mutation that does not move them to the hill is neither good nor bad. Mutations that are neither good nor bad are called *neutral mutations*. Most of these mutations are neutral.

Let’s focus on structures on or near the hill. For structures at the foot of the hill,

slightly over half the mutations are neutral and the other half are good. The average effect of mutations at the foot of the hill is positive. Once we are well off the plane and onto the slope of the hill, mutations are roughly half good with a slightly higher fraction being bad. The net effect of mutation is slightly negative. Near or at the top of the hill, almost all movements result in lower fitness; almost all mutations are bad. Using this palette of possibilities, let's examine the net effect of mutation during evolution.

Inferior creatures, those not on the hill, cannot be harmed by mutation. Creatures on the hill but far from the top see little net effect from mutation. Good creatures are affected negatively by mutation. If mutation were operating in a vacuum, creatures would end up mostly on the hill with some bias toward the top. Mutation does not operate in a vacuum, however. Selection causes better structures to be saved. Near the top of the hill, those structures that leap downhill can be replaced and more tries can be made to move uphill from the better structures. The process of selection permits us to cherry-pick better mutations.

Biological mutations, random changes in an organism's DNA, are typically neutral. Much DNA does not encode useful information. The DNA that does encode useful information uses a robust encoding so that many single-base changes do not change what the gene does. The network of interaction among genes is itself robust with multiple copies of some genes and multiple different genes capable of performing a specific task. Biological organisms are often "near the top of the hill" in the sense of their local environment, but the hilltops are usually large and fairly flat. In addition, life has adapted over time to the process of evolution. Collections of genes that are adapted to other hilltops lie dormant or semi-functional in living organisms. Studying these adaptations, the process of "evolving to evolve" is fascinating but well beyond the scope of this text.

If you find some of the proceeding material distressing, for whatever reason, I offer the following thought. The concept of evolution exists entirely apart from the reality of evolution. Even if biological evolution is a complete fantasy, it is still the source from which the demonstrably useful techniques of evolutionary computation spring. We may set aside controversy, or at least wait and discuss it over a mug of coffee, later.

Since biological reality and evolutionary computation are not inextricably intertwined, we can harvest the blind alleys of biological science as valid avenues for computational research. Consider the idea of Lamarck that acquired characteristics can be inherited. In Lamarck's view, a muscular child can result from having parents work out and build muscle tissue prior to conceiving. Lamarck's version of evolution would have it that a giraffe's neck is long because of stretching up to reach the high branches. We are certain sure this is not how biology works. However, there is no reason that evolutionary computation cannot work this way and, in fact, some types of it do. The digital analog to Lamarckian evolution is to run local optimization on a data structure and save the optimized version: acquired characteristics are inherited.

Issues to Consider

In the remainder of this text, you should keep the following notions in mind.

- The *representation* used in a given example of evolutionary computation is the data structure used together with the choice of variation operators.
- The *fitness function* is the method of assigning a heuristic numerical estimate of quality to members of the evolving population. It may only be necessary to decide which of two structures is better rather than to assign an actual numerical quality.

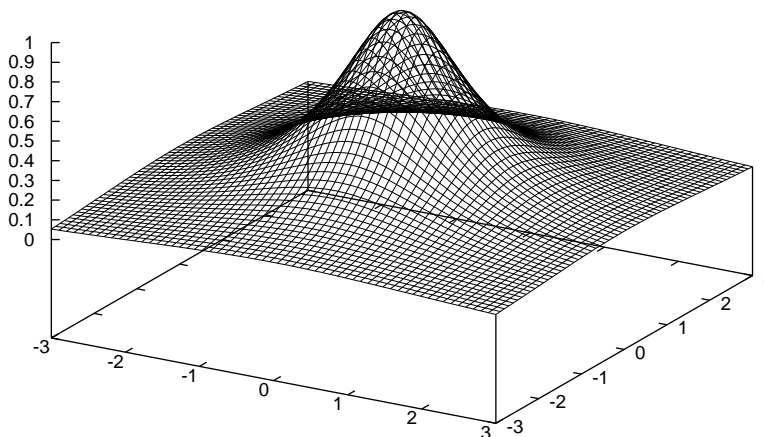
The choice of representation and fitness function can have a huge impact on the way an evolutionary computation system performs.

This text presumes no more familiarity with mathematics than a standard introduction to the differential and integral calculus. Various chapters use solid calculus, graph theory, Markov chains, and some statistics. The material used from these disciplines appears, in summary form, in the appendixes. Instructors who are not interested in presenting these materials can avoid them without much difficulty - they are in specific chapters and sections and not foundational to the material on evolutionary computation presented. The level of algorithmic competence needed varies substantially from chapter to chapter; the basic algorithms are nearly trivial *qua* algorithm. Genetic programming involves highly sophisticated use of pointers and dynamic allocation. Students whose programming skills are not up to this can be given software to use to preform experiments.

Problems

Problem 1.1 Consider a system in which the chance of a good mutation is 10%, the chance of a bad mutation is 50%, and the chance of a neutral mutation is 40%. The population has two creatures. It is updated by copying the better creature over the worse and then mutating the copy. A good mutation adds one point of fitness and bad mutations subtract one point of fitness. If we start with two creatures that have fitness zero, compute the expected fitness of the best creature as a function of the number of population updatings.

Hill Function —



Problem 1.2 *The function*

$$f(x, y) = \frac{1}{x^2 + y^2 + 1}$$

is graphed above. It is a single hill with its peak at $(0, 0)$. Suppose we have a data structure holding real values (x, y) with fitnesses $f(x, y)$. Mutation consists of moving a distance of exactly 1 in a direction selected uniformly at random.

- (i) Give a minimal length sequence of mutations that take the point $(2, 2)$ to the point $(0, 0)$ without ever lowering the fitness.
- (ii) Prove that every point in the plane has a sequence of mutations that take it to the top of the hill.
- (iii) Give a point (x, y) that cannot be taken by a sequence of mutations to $(0, 0)$ without lowering the fitness along the way.
- (iv) Compute the minimal number of mutations needed to take (x, y) to $(0, 0)$ as a function of x and y .
- (v) For which points (x, y) can the paths found in (iv) avoid a step in which fitness goes down?

Problem 1.3 Type “evolution” into an internet search engine. Set a cutoff, say the first 25 sites, and tally how many of the sites, in your opinion, were created by a person or organization that is using the definition of evolution given in this section. Report the relative numbers of the two types of website found.

Problem 1.4 Essay. Some genes generate traits fairly directly: if you block the gene, that trait goes away and the organism is otherwise unchanged. Other genes are more like control points. Knocking out a control gene can turn whole complexes of other genes off (or on). Which of these two sorts of genes are better targets for selective breeders? Imagine, for example, trying to breed high yield corn or a dog with an entirely new appearance.

Problem 1.5 Essay. Consider the following animals: rabbit, box turtle, and deer. All three are herbivores living in North America. Do your best to assess, or at least discuss, the relative fitness of these creatures.

Problem 1.6 Essay. Compare and contrast North American deer, African antelopes, and Australian kangaroos. Do these animals live in similar environments? Do they do similar “jobs?” Is there a best way to be a large herbivore.

1.2 Evolutionary Computation

We already know that evolutionary computation uses algorithms that operate on populations of data structures by selection and variation. Figure 1.1 gives a very simple version of the basic loop for an evolutionary algorithm.

```
    Create an initial population.
    Repeat
        Test population member quality.
        Copy solutions with a quality bias.
        Vary the copies of the solutions.
    Until Done
```

Figure 1.1: **The basic loop for most evolutionary algorithms**

In an evolutionary algorithm, the first step is to create a population of data structures. These structures may be filled in at random, designed to some standard, or be the output of some other algorithm. A fitness function is used to decide which solutions deserve further

attention. In the main loop of the algorithm, we pick solutions so that *on average* better solutions are chosen. This is the process of selection. The selected solutions are copied over other solutions. The solutions slated to die may be selected at random or with a bias toward worse solutions. The copied solutions are then subjected to variation. This variation can be in the form of random tweaks to a single structure or exchange of material between structures. Changing a single structure is called *unary variation* or *mutation*. Exchanging material between structures is called a *binary variation* or *crossover*.

The main loop iterates this process of population updating via selection and variation. In line with a broad outline of the theory of evolution, this should move the population toward more and more fit structures. This continues until you reach an optimum in the space of solutions defined by your fitness function. This optimum may be the best possible place in the entire fitness space, or it may merely be better than all structures “nearby” in the data structure space. Adopting the language of optimization, we call these two possibilities a *global optimum* and a *local optimum*. Unlike many other types of optimizer, an evolutionary algorithm can jump from one optimum to another. Even when the population has found an optimum of the fitness function, the population members scatter about the peak of that optimum. Some population members can leak into the area near another optimum. Figure 1.2 shows a fitness function with two optima.

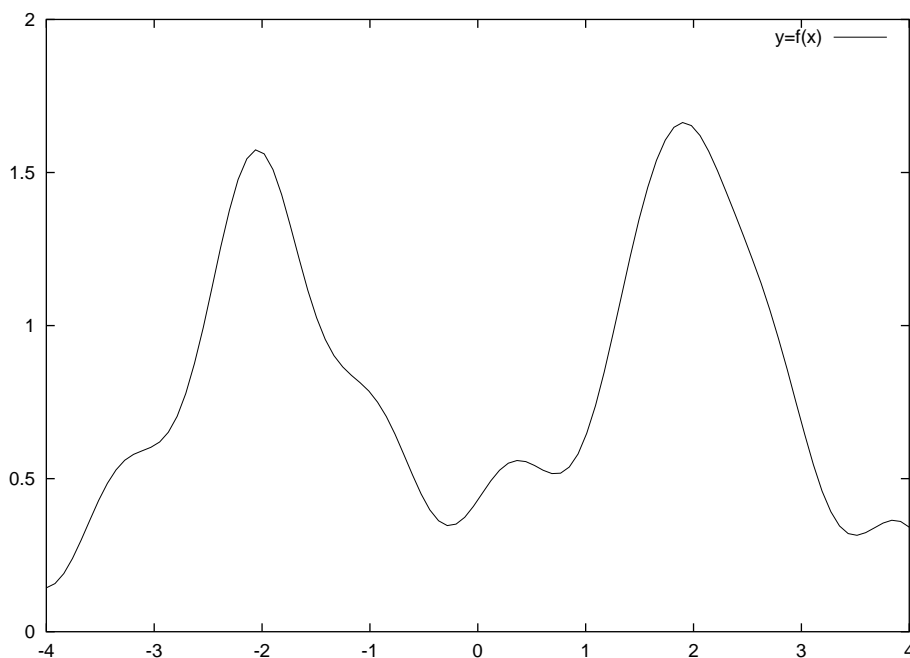


Figure 1.2: **A function with two major optima and several local optima**

An evolutionary algorithm operates on a population of candidate solutions rather than

a single solution. By playing with the fitness function, failing to be strict about allowing only the very best solutions to reproduce, tinkering with the population size, and running multiple populations, one can get evolutionary algorithms to solve difficult problems.

It is important to keep in mind that not all problems have solutions. The above description applies to problems with exact and well-defined solutions, for example finding the maximum of a function. Evolutionary algorithms can also be used to solve problems which provably fail to have optimal solutions. Suppose that the task at hand is to play a game against another player. Some games, like tick-tack-toe, are futile and you cannot learn to win them when playing against a competent player. Other games, like chess, may have exact solutions, but finding them in general lies beyond the computational ability of any machine envisioned within our current understanding of natural law. Finally, games like the Iterated Prisoner's Dilemma, described in Robert Axelrod's book, *The Evolution of Cooperation* [4], are intransitive: for every possible way of playing the game in a multi-player tournament, there is another way of playing it that can tie or beat the first way. Oddly, this does not put Prisoner's Dilemma in a class with tick-tack-toe, but rather makes it especially interesting.

Many real world situations have strategies which work well in some contexts and badly in others. The "best strategy" for Prisoner's Dilemma varies depending on whom you are playing. Political science and evolutionary biology both make use of Prisoner's Dilemma as a model of individual and group interaction. Designing a good fitness function to evolve solutions to these kinds of problems is less straightforward. We will treat Prisoner's Dilemma in greater depth in Chapter 6.

Genetic algorithms are, perhaps, the best known type of evolutionary algorithm. Genetic algorithms are evolutionary algorithms that operate on a fixed-sized data structure and which use both mutation and crossover to accomplish variation. It is problem and context dependent whether crossover helps an evolutionary algorithm locate new structures efficiently. We will experiment with the utility of crossover in later chapters. There are a large variety of different types of crossover, even for fixed data structures. An example of crossover of two 6-member arrays of real numbers and of two 12-character strings is shown in Figure 1.3.

Possibly the central issue in evolutionary computation is the representation issue. Suppose, for example, that you are optimizing a real function with 20 variables. Would it be more sensible to evolve a gene that is an array of 20 real numbers or a gene which is a 960-bit string which codes for real numbers in some fashion? Should the crossover in the algorithm respect the boundaries of the real numbers or be allowed to cleave the structure in the middle of a real number? What about problems more complex than real function optimization? What data structure works best for them? We will address these questions with experiments, problems, and examples in later chapters. This text will introduce a broad variety of representations.

Another important concept in evolutionary computation is co-evolution. In his paper on evolving sorting networks [19], W. Daniel Hillis built an evolving system in which both the

Parent 1	3.2	5.6	1.4	7.6	6.7	3.3
Parent 2	1.4	6.7	6.8	9.2	2.1	4.3
Child 1	3.2	5.6	6.8	9.2	2.1	4.3
Child 2	1.4	6.7	1.4	7.6	6.7	3.3

Parent 1	a	a	a	b	b	b	c	c	c	d	d	d
Parent 2	A	A	A	B	B	B	C	C	C	D	D	D
Child 1	a	a	a	b	B	B	C	C	C	d	d	d
Child 2	A	A	A	B	b	b	c	c	c	D	D	D

Figure 1.3: An example of crossover of data structures consisting of 6 real numbers and of 12 characters (Crossover occurs after gene position 2 for the real-number structures and between position 5 and 9 for the strings.)

population of sorting networks he was operating on and the collection of test cases being used to evaluate their fitness were allowed to evolve. The solutions were judged to have fitness in proportion to the number of test cases they solved, while the test cases were judged to have fitness in proportion to the number of solutions they fooled. As Hillis sorters got better, the problems they were tested on became harder (or at least focused on the weaknesses of the current crop of sorters). The biological idea that inspired Hillis was parasitism; a biologist might more properly term the Hillis technique co-evolution of competing species. (The fact that a biologist might not like Hillis's analogy does not invalidate Hillis's technique - exactness of biological analogy is not only not required but may not really be possible.)

There are two broad classes of evolutionary software that we will call *evolving* and *co-evolving* in this text. An evolving population has members whose fitness is judged by some absolute and unchanging standard, e.g., smallness of the dependent variable when minimizing a function. The smaller the value of the evaluation function a given creature in an evolving system has found, the more fit it is. In a co-evolving population the fitness of a member of the evolving population is found by a context dependent standard. A data structure may be quite fit at one time, unfit later in time, and then later return to being very fit. For example, when we evolve creatures to play Prisoner's Dilemma, the fitness of a creature will depend on the exact set of strategies in the current population. The intransitivity of Prisoner's Dilemma makes every strategy suboptimal in some situation. Returning to Hillis sorting networks, the use of co-evolving test problems did indeed enhance the performance of his search algorithm over that observed in earlier runs with a fixed set of test cases. By transforming a system that evolved to one that co-evolved, Hillis enhanced performance.

Another example of this transformation of an evolving system into a co-evolving system appears in David Goldberg's classic *Genetic Algorithms in Search, Optimization, and Ma-*

chine Learning [14]. His suggestion is to reduce the fitness of a member of a population in proportion to the number of other solutions that are essentially the same. In a real function optimizer, this might be the number of solutions that are close by in the domain space. The effect of this is to make solutions less good once they have been discovered by several members of the population. This reduces the accumulation of solutions onto a good, but suboptimal, solution found early on in the search. This technique is called *niche specialization* and is inspired by the notion of biological niches. The kangaroo in Australia, the deer in North America, and the gazelle in Africa are in the same biological niche. In theory, once a niche is filled, it becomes hard for new species to enter the niche. This is because the existing residents of the niche are already using the resources it contains.

Notice that niche specialization is a transformation from evolution to co-evolution. The standard of fitness changes from an absolute one - the function being optimized - to one in which the current membership of the population is also relevant. This example, while co-evolutionary, is in some sense closer to being evolutionary than the Prisoner's Dilemma example. There is not a strict dichotomy between evolution and co-evolution. Rather there is a spectrum of intermediate behaviors.

In biology, a creature that is obviously an inferior competitor (if placed in a gladiatorial ring and told to fight) can survive by living in out of the way places. In terms of long-term survival of a population, this is a good thing. An individual who is suboptimal in one sense may have traits which become useful when the environment changes or which are valuable when crossed over with another individual. The history of hybridization of cereal grains contains examples of these phenomena. This idea can be applied to evolutionary algorithms through the addition of a population structure that restricts mating or reproduction. By only allowing creatures to breed with a limited set of other creatures, superior creatures take over more slowly. This prevents premature convergence to local optima or, in a simulation that is not an optimizer, it prevents premature diversity loss in the population. We will explore techniques of this type in Chapter 13.

Definition 1.2 *A string evolver is an evolutionary algorithm that tries to match a reference string starting from a population of random strings. The underlying character set of the string evolver is the alphabet from which the strings are drawn.*

Several of the Problems for this section involve string evolvers. String evolvers often serve as a baseline or source of reference behavior in evolutionary algorithm research. An evolutionary algorithm for a string evolver functions as follows. Start with a reference string and a population of random strings. The fitness of a string is the number of positions in which it has the same character as the reference string. To evolve the population, split it into small random groups called *tournaments*. Copy the most fit string (break ties by picking at random among the most fit strings) over the least fit string in each tournament. Then, change one randomly chosen character in each copy (mutation). Repeat until an exact match with

the reference string is obtained. Typically, one records the number of tournaments, called *generations*, required to find a copy of the reference string.

A word of warning to student and instructor alike. The string evolver problem is a *trivial* problem. It is a place to cut your teeth on evolutionary algorithms, *not* an intrinsically interesting problem. It is an odd feature of the human mind that people immediately think of fifty or sixty potential improvements as soon as they hear a description of your current effort. If you have a cool idea about how to improve evolutionary algorithms, then you might try it out on a string evolver. However, bit twiddling improvements that are strongly adapted to the string evolver problem are probably not of much value. An example of a string evolver's output is given in Figure 1.4.

Best String	Fitness	Appeared in Generation

HadDe Q' /--<jlm'	3	5
HadDe.em3m/<Ijm-	4	52
HadDe,em3m/<Ijm-	5	54
HadDm,ex3m/#Ijmj	6	73
HadDm,eI8m/#Ijmj	7	86
HadDm,eI8m[Ajjmt	8	118
HadDm,UI8m[Ajjm.	9	135
MadDm,zI8m4AJ1m.	10	154
Madam,zIXm4AJ1m.	11	163
Madam, InmqAJym.	12	256
Madam, I'mqArHm.	13	327
Madam, I'm AC~m.	14	473
Madam, I'm APam.	15	512
Madam, I'm Adam.	16	647

Figure 1.4: The output of a string evolver operating over the printable ASCII characters with a population of 60 strings (The reference string is “Madam, I’m Adam.” Shown are each string that achieves a new best fitness together with its generation of appearance.)

Problems

Problem 1.7 Write and debug your own version of the string evolver described following Definition 1.2. Let your population contain 60 strings and set the size of the tournaments to $n = 2$. Run the algorithm 50 times and report the mean, standard deviation, and maximum and minimum number of generations needed to find solutions.

Problem 1.8 For the string evolver in Problem 1.7, what is the best value of n for the tournament size? In this case, “best” means “minimizes time-to-solution”.

Problem 1.9 Assume we are evolving strings of a fixed length l . Prove that the amount of time it takes the string evolver to converge is independent of the choice of characters in the reference string.

Problem 1.10 For tournament size 2, estimate mathematically and/or experimentally (consult your instructor) the dependence of the number of generations needed on average to find the reference string on the length l of the reference string. If you are taking the experimental route, give a careful description of your experiment. The material on Markov chains in Appendix A may be helpful.

Problem 1.11 Modify the string evolver from Problem 1.7 to have crossover. Use tournament size $n = 4$. For each group of 4 strings, let the two with highest fitness cross over to produce two children which replace the two with lowest fitness. With probability m for each child, randomly change (mutate) one character of the child. The probability m is termed the mutation rate. To do crossover, as in the first part of Figure 1.3, select a random position in the parental strings and exchange the suffixes starting at that position to obtain the crossed over strings of the children. Do 50 runs for $m = 0.4$ and $m = 0.8$ and compare the two mutation rates.

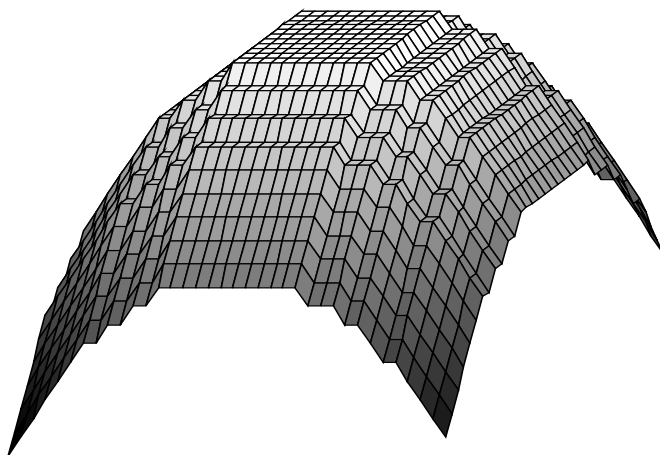


Figure 1.5: $f(x, y) = 18 - [x^2] - [y^2]$, $-3 \leq x, y \leq 3$

Problem 1.12 *Implement an evolutionary algorithm that can find the maximum or minimum of a real function of n real variables. The real function can be hard-coded into your algorithm and the number of variables should be something you can easily change. The data structures will be arrays of real numbers whose dimension is equal to the number of variables in the function you are optimizing. The fitness function is the function you are maximizing (minimizing) with the functional values interpreted appropriately. Crossover is done as in the string evolver, Problem 1.11, treating individual real numbers as if they were characters. Mutation consists of adding a uniformly distributed real number in the range $-0.2 \leq x \leq 0.2$ to some one random position in the creature's gene. Test your program on the following functions:*

(i) Minimize $f(x, y, z) = x^2 + y^2 + z^2$, $-2 \leq x, y, z \leq 2$.

(ii) Maximize $f(x, y, z) = \frac{1}{x^2 + y^2 + z^2 + 1}$, $-2 \leq x, y, z \leq 2$.

(iii) Maximize $f(x, y) = \frac{\cos(\sqrt{x^2 + y^2})}{x^2 + y^2 + 1}$, $-2\pi \leq x, y \leq 2\pi$.

(iv) Maximize $f(x, y) = 18 - \lfloor x^2 \rfloor - \lfloor y^2 \rfloor$, $-3 \leq x, y \leq 3$.

The fourth function is shown in Figure 1.5 to aid your intuition.

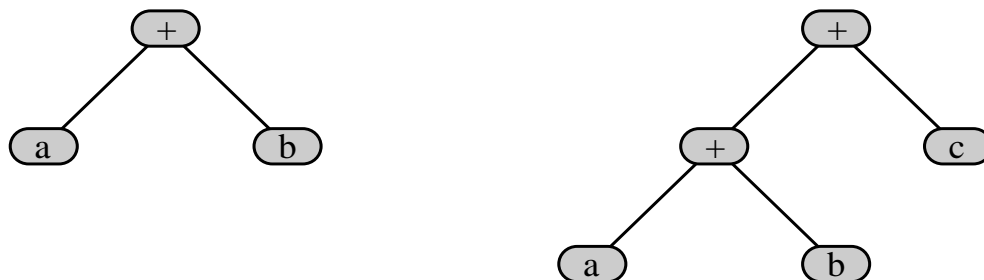
1.3 Genetic Programming

In this section, we will study an example of the use of a variable-sized data structure in evolutionary computation. In simple evolutionary algorithms, the data structure storing a member of the evolving population of solutions is of fixed size. This means care must be taken to write a data structure which is general enough that it has the potential to contain a solution. For real function optimization this isn't a terribly difficult task - an array of real numbers sized to match the number of variables in the function suffices.

When solving more subtle problems, having a sufficiently general data structure can be a significant problem. An approach to this problem was invented by John Koza and David Rice and is called *genetic programming*[23, 21, 24, 22, 6, 25]. Genetic programming (abbreviated GP) is, in spirit, the same as other evolutionary algorithms. The major difference is that the solutions are stored as parse trees. These parse trees represent general formulas, save that there is typically an upper bound on the size of the formulas. Operations are internal nodes of the trees, constants and variables are leaves, (called *terminals*). Taken together, the operations and terminals of a parse tree are called the *nodes* of the parse tree.

Since almost any imaginable computational problem with a solution can be solved with one or more formulas, possibly involving iterative operations, this gives a general solution space. In fact, the problem becomes one of having a gigantic solution space, large enough to

be quite difficult to search efficiently. Some example parse trees are given in Figure 1.6. To save some space we will usually give parse trees in a LISP-like notation in which a node and all its descendants are simply listed between parentheses, recursively. In LISP-like notation, we replace $f(x)$ with $(f x)$ and $(a + b)$ with $(+ a b)$.



$(+ a b)$, $(+ (+ a b) c)$

Figure 1.6: Some parse trees, together with their LISP-like form

Genetic programming is a good technique that often works well, but it does have a problem. This problem turns out to be like the difference between searching for a needle in a haystack and searching for a needle in Nebraska; the search space gets quite a lot larger. Several well known techniques exist for searching a haystack for a needle: sitting in the haystack and sifting through it, burning it and sorting through the ash with a magnet, etc. There are no known techniques for efficiently locating a needle randomly placed in the state of Nebraska. If you use genetic programming to evolve formulas that solve your problem, then you are in effect searching the space of formulas. This solution space is extra-large and the search will take forever unless it is done sensibly. You have to narrow your search to the right parts of Nebraska. This involves writing special purpose languages for the formulas you evolve and using heuristics to bias the initial population of formulas.

Suppose, for example, that we are using genetic programming to find an efficient formula for encoding and interpolating a data set with two independent variables and one dependent variable. In addition, no pair of independent variables occur more than once. (In other words, the data set describes a function.) One possible special purpose language for this genetic programming task consists of real constants, the variables x and y , and the arithmetic operators $+$, $-$, \times , and \div . The fitness of a given parse tree is the sum, over the data set, of the square of the difference between the function the parse tree represents and the values given in the data set. In this case, we are minimizing the fitness function. This special purpose language has no iterative operations. Every “program” (formula) returns a real number result. It may be *just* complex enough to represent data sets that don’t have too

much wrong with them. An example of a function that could be encoded by this language is

$$f(x, y) = x^2 + y^2,$$

shown in parse tree form in Figure 1.7.

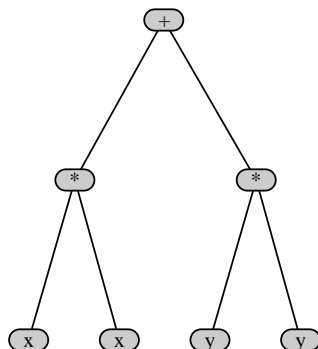


Figure 1.7: $f(x, y) = x^2 + y^2$

Notice that every possible formula and subformula in the preceding example returns the same data type (real). This is done to simplify crossover and mutation operators and the creation of “random” parse trees for the initial population. The recursive calls to generate subtrees are easier to write if they need not do type-checking. Mutation in genetic programming consists of picking a random node in the parse tree, deleting the subtree of which it is the root, and then generating a new random subtree about the same size to replace it. Crossover consists of locating a random node in each parent tree and then exchanging the subtrees of which they are the root. Examples of crossover and mutation are given in Figure 1.8. Paradoxically, it turns out that, in genetic programming, crossover is computationally much easier than mutation; it is just a simple exchange of pointers.

Parent 1	$(+(+ a b) c)$
Parent 2	$(* (* a (+ x y)) (+ a x))$
Child 1	$(+ (+ a (* a (+ x y))) c)$
Child 2	$(* b (+ a x))$
	Crossover at b and $(* a (+ x y))$
Mutation (of child 1)	$(+ (+ a (* (+ x y) (+ x y))) c)$ mutation takes a to $(+ x y)$

Figure 1.8: Crossover and Mutation

One feature many GP implementations have is subroutines or *automatically defined functions* (ADFs). The number of ADFs may be fixed (often at one) or variable. When a creature has an ADF, its gene contains additional parse trees, one for each ADF. There is an operation for each ADF, available only outside the parse tree for which that ADF is defined, that is computed by executing the ADF parse tree. There are terminals in each ADF parse tree that give it access to the arguments in the ADF from the calling parse tree. This chunking of the functionality of the program into subroutines (ADFs) is useful for many of the same reasons it is useful in standard programming. It also allows us to draw on a powerful biological paradigm: evolution by subsumption.

Another brief digression into biology is in order here. In the cells of your body, there are many organelles: ribosomes, mitochondria, etc. Some of these organelles have their own genetic code *different* from the genetic code used by the cell nucleus. It is thought that these subcellular organelles are descended from free living organisms that eons ago joined into colonial association with the single-celled ancestors of the type of cells that make up our body. This process, forming a colonial association that allows diverse types of organisms to share functionality they evolved independently, is called *evolution by subsumption*. It's not hard to see that a variation operator that spliced together the parse trees of ADFs and the main part of a GP-creature could allow this more powerful sort of evolution to take place in a GP environment.

The preceding discussion suggests that GP has a more acute case of the representation problem than even other branches of evolutionary computation. In addition to selecting population size and structure, crossover and mutation type, mutation rate, and the plethora of other relevant parameters, a genetic programmer must select the parts of his special purpose GP-language.

Another issue that arises in genetic programming is that of *disruption*. In a string evolver, the positions in the data structure each have a simple mission; to match a character in the reference string. These missions are completely independent. When we modified the string evolver to be a real function optimizer the mission-specificity of each position in the array remained, but the independence was lost. Parse trees don't even have positions, except maybe "root node," and, so, they completely lack mission specificity of their entries. A node on the left side of an ancestor may end up on the right side of a descendant.

Crossover in the real function optimizer could break apart blocks of variables that had values that worked well together. This is what is meant by *disruption*. It is intuitively obvious and has been experimentally demonstrated that the crossover used in genetic programming is far more disruptive than the crossover used in algorithms with data structures organized as arrays of strings. Thus, the probability of crossover reducing fitness is higher in genetic programming. Oddly enough, this means that evolving to evolve is easier to observe in genetic programming. Contemplate the tree fragment $(* \mathbf{0} \mathbf{T})$ where \mathbf{T} is some subtree. This tree returns a zero no matter what. If \mathbf{T} is large, then there are many locations for crossover which will not change the fitness of the tree. This sort of crossover resistance has

been observed in many genetic programming experiments. It leads to a phenomenon called *bloat* in which trees quickly grow as large as the software permits.

We will explore genetic programming and compare the technique with other evolutionary algorithms in Chapters 8, 9, 10, 12, 13, 14, and 15. Upper level students should be able to write their own parse-tree managing routines. Lower level students can use existing software and simply modify the input parameters. Skilled students at the lower levels may want to try coding their own parse-trees routines. One well-written set of routines can support a large number of special purpose languages and allows you to experiment with a wide variety of potential genetic programming applications.

Problems

Problem 1.13 *Suppose we are working in a genetic programming environment in which the language consists of the constant 1 and the operation $+$. How many nodes are in the smallest parse tree that can compute n ? As an example, computing 3 can be done with the tree $(+ (+ 1 1) 1)$, given in Lisp-like notation. This tree has 5 nodes.*

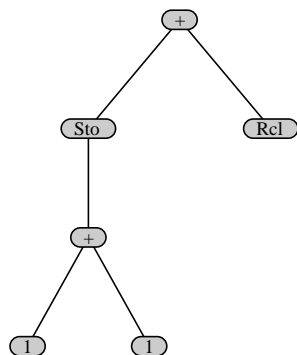


Figure 1.9: Computing 4.

Problem 1.14 *Start with the same setup as Problem 1.13. Add to the language store (STO) and recall (RCL) instructions. The STO instruction takes a single argument and puts it into an external storage location and returns its value. The RCL instruction recalls the value in the storage location. Assume that the left argument of a $+$ is executed before the right one. Find the smallest trees that compute the numbers $n = 3, 4, 5, 6, 7, 8, 9, 10, 11, 12$. The program in Figure 1.9 computes 4, for example, using 6 nodes. (We would like to ask how many nodes are in the smallest parse tree that can compute n , but solving this problem in full generality is extremely hard.)*

Problem 1.15 Using the language described in problem 1.14, find the largest number that can be computed by a tree with k nodes for $k = 3, 4, 5, 6, 7, 8, 9, 10, 11, 12$. Advanced students with a great deal of time might want to solve this problem for general k .

Problem 1.16 Assume you are in a GP environment which has the operations $+$, $-$, $*$, $/$, a variable x , and integer constants. Give an example of a parse tree that can compute $x^5 + 3x^3 - 4x^2 - 4x + 1$ with as few nodes as possible. Advanced students should prove that they are using the smallest possible number of nodes.

Problem 1.17 The STO operation given in Problem 1.14 is mathematically the identity function; it simply returns its argument. It is valuable not for the computations it performs but rather because of its side effect of storing its argument into a memory for later access with RCL. Using some sort of node with a side effect, define a set of operations and terminals that will let you compute the solutions to

$$ax^2 + bx + c = 0.$$

Give a single parse tree that returns the first solution the first time it is called and the second solution the second time it is called. The tree should also deal in a reasonable fashion with cases in which there are not two roots. You may use real or complex arithmetic as your base type.

Problem 1.18 One important difference between the string and tree crossover operators given in this chapter is that the string crossover operator changes nothing when it crosses over two identical structures while the tree crossover operator can create children quite unlike a pair of identical parents. Give an example of two identical parents and crossover points that yield children unlike the parents.

Problem 1.19 Essay. Suppose we have a large number of data structures available and wish to test a crossover operator for disruptiveness. Give and defend a scheme for such testing.

Problem 1.20 Essay. In computer science, one of the famous and foundational results concerns the halting problem. The main point of interest is that it is, in principle, impossible to separate the set of all computer programs into those that will eventually stop and those that will run forever. There are some programs that obviously fit into each category, but some interesting programs are utterly unclassifiable. Write an essay that first discusses why the function approximation scheme given in this section produces parse trees that always halt, and then discuss methods for avoiding the halting problem even if the special purpose language used in a given instance of genetic programming is rich enough to allow general computer programs.

Problem 1.21 Essay. *Discuss how a genetic program could be used to enhance the source code of other software. Be sure to discuss what notion of fitness would be used by the genetic programming environment. You may want to discuss trade-offs between accuracy and speed in the evolved code.*

