Revised April 24, 1993, for *Computing with Biological Metaphors*, edited by Ray Paton for Chapman & Hall.

Evolution of Emergent Cooperative Behavior using Genetic Programming

John R. Koza
Computer Science Department
Margaret Jacks Hall
Stanford University
Stanford, California 94305-2140 USA
Koza@Cs.Stanford.Edu
415-941-0336

Abstract: *The recently developed genetic programming paradigm provides a way to genetically breed a computer program to solve a wide variety of problems. Genetic programming employs Darwinian survival and reproduction of the fittest and the genetic crossover (sexual recominbation) operation to evolve progressively better solutions to problems. In this chapter, a computer program that governs the behavior of a group of independently acting agents is genetically evolved to solve the "painted desert" problem. The evolved program exhibits emergent and cooperative behavior.*

Keywords: Genetic programming, emergent behavior, painted desert problem, genetic algorithms, independent agents, artificial ants, cooperative behavior, crossover, recombination.

1 Introduction and Overview

In nature, populations of biological individuals compete for survival and the opportunity to reproduce on the basis of how well they grapple with their environment. Over many generations, the structure of the individuals in the population evolve as a result of the Darwinian process of natural selection and survival of the fittest. That is, complex structures arise from fitness. The question arises as to whether behaviors can also be evolved using the metaphor of natural selection. Another question is whether complex artificial structures, such as computer programs, can also be evolved using this metaphor. This chapter explores these questions in the context of a problem requiring the discovery of emergent cooperative behavior in the form of a computer program that controls a group of independently acting agents (e.g., robots or social insects such as ants).

2 The Painted Desert Problem

The repetitive application of seemingly simple rules can lead to complex overall behavior (Steels 1990, 1991; Forrest 1991, Manderick and Moyson 1988, Moyson and Manderick 1990, Collins and Jefferson 1991). The study of such emergent behavior is one of the main themes of research in the field of artificial life (Langton 1991; Varela and Bourgine 1992; Meyer and Wilson 1991).

The "painted desert" problem was proposed by Mitchel Resnick and Uri Wilenski of the MIT Media Laboratory at the `*Logo` workshop at the Third Artificial Life conference in Santa Fe in 1992. In the modified, low-dimensionality version of this problem discussed in this chapter, there are 10 ants and 10 grains of sand of each of three colors (black, gray and striped). The 10 ants and 30 grains start at random positions in a 10 by 10 toroidal grid as shown in figure 1.

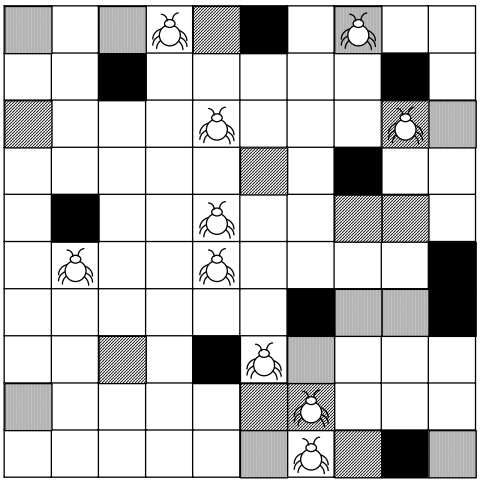


Figure 1 Starting arrangement of 10 ants, 10 black grains, 10 gray grains, and 10 striped grains of sand

Each ant executes a common computer program and can only sense information about its immediate local position in the grid. There is no direct communication between the ants. There is no central controller directing the ants.

The objective is to find a common computer program for directing the behavior of the 10 ants which, when executed in parallel by all 10 ants, enables the ants to arrange the 30 grains of sand into three vertical bands of like color. Specifically, the 10 black grains are to be arranged into a vertical band immediately to the right of the *Y*-axis, the 10 gray grains are to be arranged into a second vertical band immediately to the right of the black band, and the 10 striped grains are to be arranged into a third vertical band immediately to the right of the gray band. No more than one grain of sand is allowed at any position in the grid at any time.

Figure 2 shows the desired final arrangement of the 30 grains of sand.

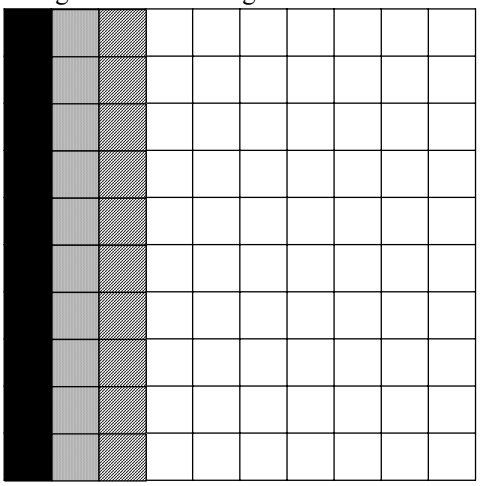


Figure 2 Desired final arrangement of the sand

It has been demonstrated that it is possible for higher-level cooperative behavior to emerge via the purely local sensing and the parallel application by each ant of an identical (and relatively simple) program. Deneubourg et al. (1986, 1991) conceived and wrote an intriguing program involving only local interactions which, when simultaneously executed by a group of ants, enables the ants to consolidate widely dispersed pellets of food into one pile. Travers and Resnick (1991) conceived and wrote a parallel program and produced a videotape that showed how a group of ants could locate piles of food and transport the food to a central place. Their central place foraging program permitted the ants to drop slowly dissipating chemicals, called pheromones, to recruit and guide other ants to the food. See also Resnick 1991. These programs were written using considerable ingenuity and human intelligence.

In this chapter, the goal is to genetically breed a common computer program that, when simultaneously executed by a group of ants, causes the emergence of interesting higher-level collective behavior. In particular, the goal is to evolve a common program that solves the painted desert problem as described above.

3 Background on Genetic Algorithms

John Holland's pioneering 1975 *Adaptation in Natural and Artificial Systems* described how the evolutionary process in nature can be applied to artificial systems using the genetic algorithm operating on fixed length character strings [Holland 1975]. Holland demonstrated that a population of fixed length character strings (each representing a proposed solution to a problem) can be genetically bred using the Darwinian operation of fitness proportionate reproduction, the genetic operation of recombination, and occasional mutation. The recombination operation combines parts of two chromosome-like fixed length character strings, each selected on the basis of their fitness, to produce new offspring strings. Holland established, among other things, that the genetic algorithm is a mathematically near optimal approach to adaptation in that it maximizes expected overall average payoff when the adaptive process is viewed as a multi-armed slot machine problem requiring an optimal allocation of future trials given currently available information.

The genetic algorithm has proven successful at searching nonlinear multidimensional spaces in order to solve, or approximately solve, a wide variety of problems [Goldberg 1989, Davis 1987, Davis 1991, Michalewicz 1992, Belew and Booker 1991, Schwefel and Maenner 1991, Davidor 1991, Whitley 1992, Maenner and Manderick 1992].

4 Background on Genetic Programming

For many problems the most natural representation for the solution to the problem is a computer program (i.e., a composition of primitive functions and terminals), not merely a single point in a multidimensional numerical search space. Moreover, the exact size and shape of the composition needed to solve many problems is not known in advance.

Genetic programming provides a domain-independent way to search the space of computer programs composed of certain given terminals and primitive functions to find a program of unspecified size and shape which solves, or approximately solves, a problem. The book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* [Koza 1992a] describes the genetic programming paradigm and demonstrates that populations of computer programs (i.e., compositions of primitive functions and terminals) can be genetically bred to solve a surprising variety of different problems in a wide variety of different fields. Specifically, genetic programming has been successfully applied to problems such as

- planning (e.g., navigating an artificial ant along a trail, developing a robotic action sequence that can stack an arbitrary initial configuration of blocks into a specified order) [Koza 1991],
- discovering control strategies for backing up a tractor-trailer truck, centering a cart, and balancing a broom on a moving cart,
- discovering inverse kinematic equations to control the movement of a robot arm to a designated target point,
- evolution of a subsumption architecture for robotic control [Koza 1992c, Koza and Rice 1992b],
- emergent behavior (e.g., discovering a computer program which, when executed by all the ants in an ant colony, enables the ants to locate food, pick it up, carry it to the nest, and drop pheromones along the way so as to produce cooperative emergent behavior) [Koza 1991],
- classification and pattern recognition (e.g., distinguishing two intertwined spirals),
- generation of maximal entropy random numbers,
- induction of decision trees for classification,
- optimization problems (e.g., finding an optimal food foraging strategy for a lizard) [Koza, Rice, and Roughgarden 1992],
- symbolic "data to function" regression, symbolic integration, symbolic differentiation, symbolic solution to general functional equations (including differential equations with initial conditions, and integral equations) and sequence induction (e.g., inducing a recursive computational procedure for generating sequences such as the Fibonacci sequence),
- empirical discovery (e.g., rediscovering Kepler's Third Law, rediscovering the well-known non-linear econometric "exchange equation" $MV = PQ$ from actual, noisy time series data for the money supply, the velocity of money, the price level, and the gross national product of an economy),
- Boolean function learning (e.g., learning the Boolean 11-multiplexer function and 11-parity functions),
- finding minimax strategies for games (e.g., differential pursuer-evader games, discrete games in extensive form) by both evolution and co-evolution, and
- simultaneous architectural design and training of neural networks.

A videotape visualization of 22 applications of genetic programming can be found in the *Genetic Programming: The Movie* [Koza and Rice 1992a]. See also Koza [1992b].

In genetic programming, the individuals in the population are compositions of primitive functions and terminals appropriate to the particular problem domain. The set of primitive functions used typically includes arithmetic operations, mathematical functions, conditional logical operations, and domain-specific functions. The set of terminals typically includes inputs appropriate to the problem domain and various constants.

The compositions of primitive functions and terminals described above correspond directly to the computer programs found in programming languages such as LISP (where they are called symbolic expressions or S-expressions). In fact, these compositions in turn correspond directly to the parse tree that is internally created by the compilers of most programming languages. Thus, genetic programming views the search for a solution to a problem as a search in the space of all possible functions that can be recursively composed of the available primitive functions and terminals.

5 Steps Required to Execute Genetic Programming

Genetic programming is a domain independent method that proceeds by genetically breeding populations of compositions of the primitive functions and terminals (i.e., computer programs) to solve problems by executing the following three steps:

(1) Generate an initial population of random computer programs composed of the primitive functions and terminals of the problem.

(2) Iteratively perform the following sub-steps until the termination criterion for the run has been satisfied:

   (a) Execute each program in the population so that a fitness measure indicating how well the program solves the problem can be computed for the program.

   (b) Create a new population of programs by selecting program(s) in the population with a probability based on fitness (i.e., the fitter the program, the more likely it is to be selected) and then applying the following primary operations:

      (i) *Reproduction*: Copy an existing program to the new population.

      (ii) *Crossover*: Create two new offspring programs for the new population by genetically recombining randomly chosen parts of two existing programs.

(3) The single best computer program produced during the run is designated as the result of the run of genetic programming. This result may be a solution (or approximate solution) to the problem.

5.1 Crossover (Recombination) Operation

In genetic programming, the genetic crossover (sexual recombination) operation operates on two parental computer programs and produces two offspring programs using parts of each parent.

For example, consider the following computer program (LISP symbolic expression):

```
(+ (* 0.234 Z) (- X 0.789)),
```

which we would ordinarily write as

$$0.234z + x - 0.789.$$

This program takes two inputs (`X` and `Z`) and produces a floating point output. In the prefix notation used in LISP, the multiplication function `*` is first applied to the terminals `0.234` and `Z` to produce an intermediate result. Then, the subtraction function `-` is applied to the terminals `X` and `0.789` to produce a second intermediate result. Finally, the addition function `+` is applied to the two intermediate results to produce the overall result.

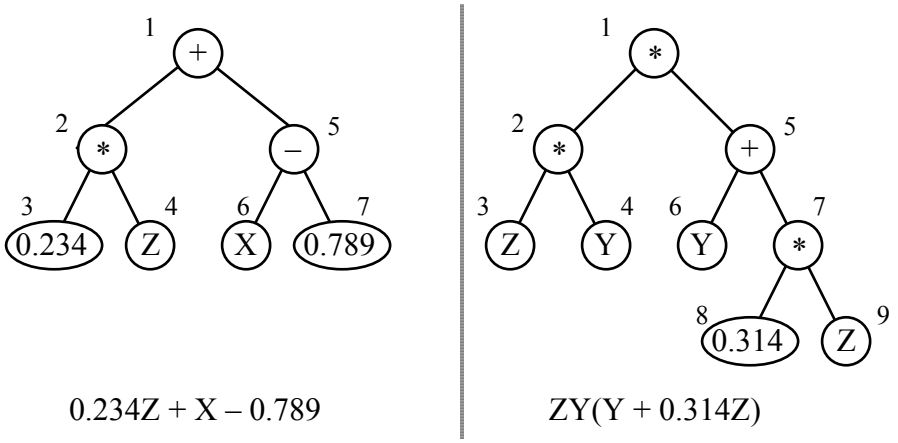
Also, consider a second program:

```
(* (* Z Y) (+ Y (* 0.314 Z))),
```

which we would ordinarily write as

$$zy\,(y + 0.314\,z).$$

In figure 3, these two parental programs are depicted as rooted, point-labeled trees with ordered branches. Internal points (i.e., nodes) of the tree correspond to functions (i.e., operations) and external points (i.e., leaves, endpoints) correspond to terminals (i.e., input data). The numbers beside the function and terminal points of the tree appear for reference only.

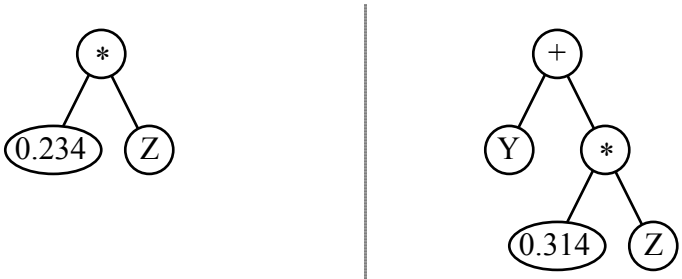


0.234Z + X – 0.789 ZY(Y + 0.314Z)

Figure 3 Two parental computer programs

The crossover operation creates new offspring by exchanging sub-trees (sublists, subroutines, subprocedures) between the two parents.

Assume that the points of both trees are numbered in a depth-first way starting at the left. Suppose that the point number 2 (out of the 7 points of the first parent) is randomly selected as the crossover point for the first parent and that the point number 5 (out of the 9 points of the second parent) is randomly selected as the crossover point of the second parent. The crossover points in the trees above are therefore the * in the first parent and the + in the second parent. The two crossover fragments are the two sub-trees shown in figure 4.



0.234Z Y + 0.314Z

Figure 4 Two crossover fragments

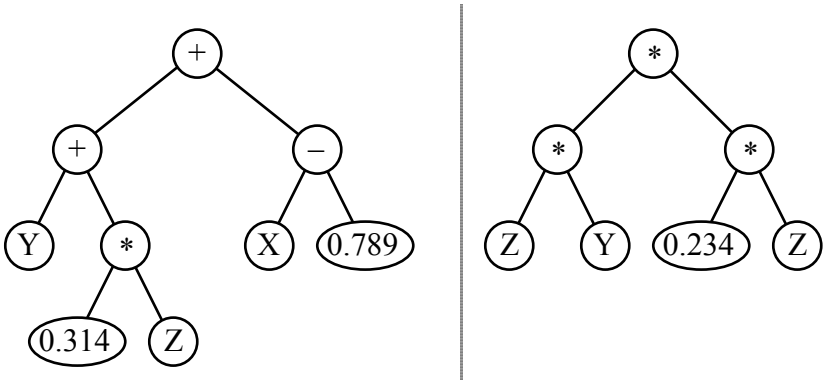
These two crossover fragments correspond to the underlined sub-programs (sub-lists) in the two parental computer programs. The two offspring resulting from crossover are

```
(+ (+ Y (* 0.314 Z)) (- X 0.789))
```

and

```
(* (* Z Y) (* 0.234 Z)).
```

The two offspring are shown in figure 5.



Y + 0.314Z + X – 0.789 $0.234Z^2Y$

Figure 5 Two offspring

Thus, crossover creates new computer programs from parts of existing parental programs. Because entire subtrees are swapped, if the set of functions and terminals are closed, this crossover operation always produces syntactically and semantically valid programs as offspring regardless of the choice of the two crossover points. Because programs are selected to participate in the crossover operation with a probability proportional to their fitness, crossover allocates future trials to parts of the search space whose programs contain parts from promising programs.

6 Preparatory Steps for Using Genetic Programming

There are five major steps in preparing to use genetic programming, namely determining

(1) the set of terminals,
(2) the set of primitive functions,
(3) the fitness measure,
(4) the parameters for controlling the run, and
(5) the method for designating a result and the criterion for terminating a run.

The individual programs in the population are compositions of the primitive functions from the function set and terminals from the terminal set. Since the set of ingredients from which the computer programs are composed must be sufficient to solve the problem, it seems reasonable that the programs would include (1) the inputs to the common computer program being evolved to govern the behavior of each ant, (2) functions enabling the ant to move, to pick up a grain of sand, to drop a grain of sand, and (3) functions enabling the ant to make decisions based on the limited information that it receives as input.

The first major step in preparing to use genetic programming is the identification of the set of terminals. In this problem, the inputs to each ant consist only of information concerning the ant's current location in the grid. This information consists of

- `X`, an indication of the ant's current horizontal location (from 0 to 9) in the grid,
- `Y`, an indication of the ant's current vertical location (from 0 to 9) in the grid,
- `CARRYING`, an indication of the color of the grain of sand (0 for black, 1 for gray, or 2 for striped) that the ant is currently carrying or –1 otherwise, and
- `COLOR`, an indication of the color of the grain of sand, if any, on the square of the grid where the ant is currently located or –1 if there is no grain at that square.

The side-effecting functions `GO-N`, `GO-E`, `GO-S`, and `GO-W` enable the ant to move one square north, east, south, and west, respectively. The side-effecting function `MOVE-RANDOM` randomly selects one of the above four functions and executes it. The side-effecting function `PICK-UP` picks up the grain of sand (if any) at the ant's current position provided the ant is not already carrying a grain of sand. These side-effecting functions return the value (1, 2, or 3) of the color of sand at the ant's current position. Since these six side-effecting functions take no arguments, they are best viewed as terminals. That is, the terminal set represents the endpoints of the program tree.

Thus, the terminal set $\mathcal{T}$ for this problem is

$$\mathcal{T} = \{\texttt{X, Y, CARRYING, COLOR, (GO-N), (GO-E), (GO-S), (GO-W), (MOVE-RANDOM), (PICK-UP)}\}.$$

The second major step in preparing to use genetic programming is the identification of a function set for the problem.

The four-argument conditional branching function `IFLTE` evaluates and returns its third (then) argument if its first argument is less than or equal to its second argument and otherwise evaluates and returns its fourth (else) argument. For example, `(IFLTE 2 3 A B)` evaluates to the value of `A`. The three-argument conditional branching function `IFLTZ` evaluates and returns its second (then) argument if its first argument is less than zero and otherwise evaluates and returns its third (else) argument. If the ant is currently carrying a grain of sand and if the ant's current position does not have a grain of sand, the two-argument conditional branching function `IF-DROP` drops the sand that the ant is carrying and evaluates and returns its first (then) argument, otherwise `IF-DROP` evaluates and returns its second (else) argument. Testing functions such as these are commonly used in genetic programming and are implemented as macros.

Thus, the function set $\mathcal{F}$ for this problem consists of

$$\mathcal{F} = \{\texttt{IFLTE, IFLTZ, IF-DROP}\}$$

taking 4, 3, and 2 arguments, respectively.

Each computer program in the population is a composition of primitive functions from the function set $\mathcal{F}$ and terminals from the terminal set $\mathcal{T}$. Since genetic programming operates on an initial population of randomly

generated compositions of the available functions and terminals and later performs genetic operations, such as crossover, on these individuals, each primitive function in the function set should be well defined for any combination of arguments from the range of values returned by every primitive function that it may encounter and the value of every terminal that it may encounter. The above choices of terminals and functions (specifically, the use of numerically valued logic and the return of a numerical value for the side-effecting functions) ensures this desired closure.

The third major step in preparing to use genetic programming is the identification of the fitness measure that evaluates the goodness of an individual program in the population. The fitness of a program in the population is measured by executing a simulation for 300 time steps with the 10 ants and the 30 grains of sand starting from the randomly-selected positions shown in figure 1.

The fitness of an individual program is the sum, over the 30 grains of sand, of the product of the color of the grain of sand (1, 2, or 3) and the distance between the grain and the *Y*-axis when execution of the particular program ceases. If all 30 grains of sand are located in their proper vertical band, the fitness will be 100. This fitness measure is higher and less favorable when the grains of sand are less perfectly positioned. If a program leaves all the grains of sand at their starting positions shown in figure 1, it will score a fitness of 395.

This work was done on a serial computer, so the common computer program is, in practice, executed for each ant in sequence. Thus, the locations of the grains of sand are altered by the side-effecting actions of earlier ants in the sequence. In addition, certain special situations must also be considered in implementing this particular problem. To conserve computer time, an individual computer program is never allowed to execute more than 1,000 side-effecting operations over the 300 occasions that each individual program is allowed to be executed for each ant. If this number is exceeded, further execution of that particular program ceases and fitness is then measured at that moment. If an ant happens to be carrying a grain of sand when fitness is being measured and the ant's current position does not have a grain of sand, the grain is deemed to be located at that position. If the ant's current position is occupied by a grain of sand, but the place from which it picked up its grain is unoccupied by a grain, the grain is deemed to be located at that place (i.e., it is rubber-banded there). If both these positions are occupied by a grain, the grain is deemed to be located at the first unoccupied position near the ant's current position (these positions being considered in a particular orderly way).

The fourth major step in preparing to use genetic programming is the selection of values for certain parameters. Our choice of values for the primary parameters (i.e., 500 for the population size, *M*, and 51 as the maximum number of generations, *G* to be run) reflects our judgment as to the likely complexity of the solution to this problem. These same two choices were made for the vasy majority of problems presented in Koza 1992a. Our choice of values for the various secondary parameters that control the run of genetic programming are with one exception, the same default values as we have used on numerous other problems [Koza 1992a]. The exception is that that we used tournament selection (with a group size of seven), rather than fitness proportionate selection. In tournament selection, a group of a specified size (seven here) is selected at random from the population and the single individual with the best value of fitness is selected from the group. For the crossover operation, two such selections are made to select the two parents to participate in crossover.

Finally, the fifth major step in preparing to use genetic programming is the selection of the criterion for terminating a run and the selection of the method for designating a result. We will always terminate a given run after 51 generations. We count each grain of sand that is in its proper vertical band as a "hit." We also terminate a run if 30 hits are attained at any time step. We designate the best individual obtained during the run (the best-so-far individual) as the result of the run.

7 Results for One Run

A review of one particular run will serve to illustrate how genetic programming discovers progressively better computer programs to solve the painted desert problem.

One would not expect any individual from the randomly generated initial population to be very good. In generation 0, the median individual in this population of 500 consists of four points (i.e., functions and terminals), and is

```
(IFLTZ Y (GO-S) X).
```

Depending on the value of the first argument, `Y`, to the conditional branching function `IFLTZ`, this program either moves an ant to the south or pointlessly evaluates the third argument, `X`. This program does not move any grains of sand and therefore scores a fitness of 395. This median level of performance for generation 0 represents a baseline value for a random search of the space of possible compositions of terminals and primitive functions for this problem. Of the 500 programs in generation 0, 64% score 395.

The fitness of the worst individual program among the 500 individuals in the population is 436. This worst-of-generation program consists of eight points and actually moves some grains of sand to less desirable positions than their random starting positions.

However, even in a randomly created population, some individuals are better than others. The fitness of the best-of-generation individual program from generation 0 is 262. This program has 494 points and is not shown here.

Figure 6 shows the arrangement of the grains of sand after execution of the best-of-generation individual from generation 0. As can be seen, the three colors of sand are not arranged in anything like the desired final configuration. Nonetheless, 90% of the black grains, 60% of the gray grains, and 40% of the striped grains have been moved onto the left half of the grid. This compares favorably to the starting arrangement shown in figure 1 where less than half of each color happened to start on the left half of the grid.

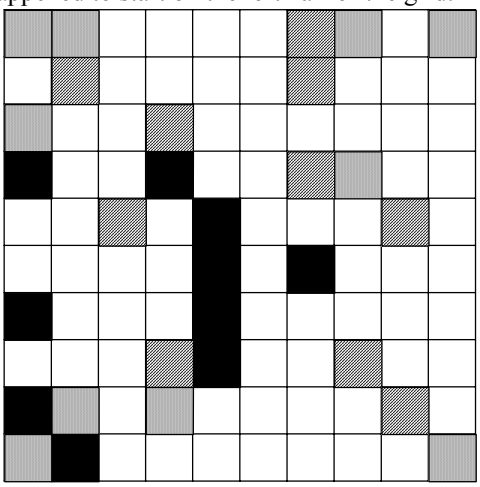


Figure 6 Sand arrangement after execution of the best-of-generation individual from generation 0.

In succeeding generations, the fitness of the median individual and the best-of-generation individual all tend to progressively improve (i.e., drop). In addition, the average fitness of the population as a whole tends to improve. For example, the best-of-generation individual from generation 1 has a fitness value of 167 and 11 points. The best-of-generation individual from generation 2 is the same individual as generation 1. The best-of-generation individual from generation 3 has a fitness value of 151 and 11 points. Of course, the vast majority of individual computer programs in the population of 500 still perform poorly.

By generation 4, the fitness of the best-of-generation individual improves to 135. This individual has 17 points and is shown below:

```
(IFLTE (GO-W) (IF-DROP (GO-N) (IFLTE (MOVE-RANDOM) COLOR Y COLOR)) (IFLTE X COLOR
COLOR (IF-DROP CARRYING (PICK-UP))) (MOVE-RANDOM)).
```

Figure 7 shows the result of executing the best-of-generation individual from generation 4. As can be seen, 70% of the black grains are in their proper vertical band immediately adjacent to the *Y*-axis. 80% of the gray grains are in their proper vertical band. 60% of the striped grains are in their proper band. This represents a considerable improvement over earlier generations.

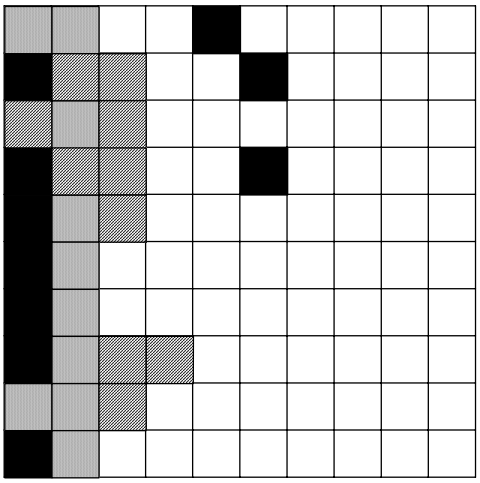


Figure 7 Result of executing the best-of-generation individual from generation 4

The fitness of the best-of-generation individual drops to 127 for generation 5, to 112 for generation 6, and then rises to 115 for generation 7.

By generation 8, the best-of-generation individual has 15 points and a fitness value of 100 (i.e., it scores 30 hits and is a perfect solution to this problem). This best-of-run individual is shown below:

```
(IFLTE (GO-W) (IF-DROP COLOR (IFLTE (MOVE-RANDOM) X (GO-S) (PICK-UP))) (IFLTE X
COLOR COLOR (PICK-UP)) (MOVE-RANDOM)).
```

Figure 8 shows this best-of-run individual from generation 9 as a rooted, point-labeled tree with ordered branches.

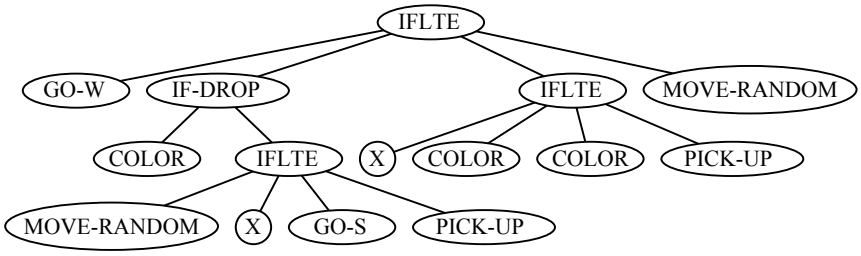


Figure 8 Best-of-run individual from generation 8.

As one watches an animated simulation of the execution of the above best-of-run program by all 10 ants, the `GO-W` and the `GO-S` functions cause the ants and sand to move in a generally south-westerly direction. The best-of-run program works by first moving to the left (west) and unconditionally dropping its sand if the square is free. If the grain of sand is to the right of its correct vertical column (as it usually will be), the ant then picks the sand up again. If the sand cannot be dropped at the ant's current location and the sand is in its correct vertical column or to the right of its correct vertical column, the ant moves south and again tries to drop the sand in a different position in the same vertical column. If the ant ever carries the sand to the left of its correct vertical column, the toroidal geometry of the grid returns the ant to the far right.

Figure 9 shows the result of executing the best-of-run individual from generation 8 after only 30 time steps. As can be seen, 25 of the 30 grains of sand are already located in the leftmost three vertical columns of the grid. Moreover, 50% of each color are already located in their correct vertical column.

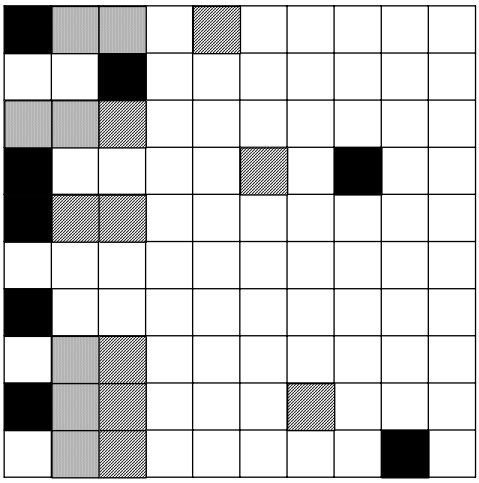


Figure 9 Intermediate result of executing the best-of-run individual from generation 8 after 30 time steps

This particular 100% correct program operates by filling each of the three vertical columns at the far left with grains of the correct color at approximately the same rate.

Note that we did not pre-specify the size and shape of the result. As we proceeded from generation to generation, the size and shape of the best-of-generation individuals changed as a result of the selective pressure exerted by Darwinian natural selection and crossover.

7.1 Genealogical Audit Trail

A genealogical audit trail can provide further insight into how genetic programming works. An audit trail consists of a record of the ancestors of a given individual and of each genetic operation that was performed on the ancestors in order to produce the individual. For the crossover operation, the audit trail includes the particular crossover points chosen within each parent.

The creative role of crossover is illustrated by an examination of the genealogical audit trail for the best-of-generation individual from generations 7 and 8 of this run.

Parent 1 is the 51st best individual from generation 7. It has a fitness value of 139, has 15 points, and scores 24 hits, and is shown below:

```
(IFLTE (GO-W)
       (IF-DROP COLOR (IFLTE (MOVE-RANDOM) J (GO-S) (GO-E))
       (IFLTE J COLOR COLOR (PICK-UP)) (MOVE-RANDOM))
```

Parent 2 is the 27th best individual from generation 7. It has a fitness value of 131, has 31 points, and scores 20 hits, and is shown below:

```
(IFLTE (GO-W)
       (IF-DROP (GO-N)
                (IFLTE (MOVE-RANDOM)
                       J
                   (GO-S)
                   (IFLTE (GO-W)
                          (IF-DROP (GO-N) (IFLTE (MOVE-RANDOM) J (GO-S) (GO-E)))
                      (IFLTE J COLOR COLOR (PICK-UP))
                      (MOVE-RANDOM))))
   (IFLTE J COLOR COLOR (PICK-UP))
   (IF-DROP (GO-N) COLOR))
```

Parent 1 is not 100% effective in solving this problem because it has no way to deal with a grain of sand that is located to the left (west) of its correct vertical column. This defect is relatively minor since it only involves the relatively small number of grains of sand that might be improperly located in the first and second vertical columns. The insertion of the `PICK-UP` function into parent 1 from generation 7, in lieu of its underlined `GO-E` function, corrects the defect in parent 1 by enabling the ant to pick up such a grain. The general westward movement of sand and the toroidal geometry of the grid then causes this grain to reappear at the far right so that it can eventually be located in its correct vertical column.

As it happens in this run, parent 1 from generation 7 and its grandparent 1 from generation 6 are identical.

Great-grandparent 1 from generation 5 is the 23rd best individual from its generation, has a fitness value of 157, has 15 points, and scores 17 hits, and is shown below:

```
(IFLTE (GO-W)
       (IF-DROP (GO-E) (IFLTE (MOVE-RAN) J (GO-S) (GO-E)))
```

```
    (IFLTE J COLOR COLOR (PICK-UP))
    (MOVE-RAN))
```

Great-grandparent 2 from generation 5 is the 42nd best individual from its generation, has a fitness value of 164, has 18 points, and scores 9 hits, and is shown below:

```
 (IFLTE (IFLTZ (GO-W) (MOVE-RAN) (GO-E))
        (IF-DROP (GO-N) (IFLTE (MOVE-RAN) COLOR I COLOR))
    (IFLTE J COLOR COLOR (PICK-UP))
    (MOVE-RAN))
```

Great-grandparent 1 from generation 5 is less effective than it might be, in part, because it does not check the color of the sand it is currently carrying and because it moves back towards the east. The insertion of the `COLOR` function into great-grandparent 1 from generation 5, in lieu of its underlined `GO-E` function, enables the ant to check the color of the sand it is currently carrying and thereby improve its fitness from 157 to 139 and to increase its number of hits from 17 to 24.

Great-great-grandparent 1 from generation 4 of great-grandparent 1 from generation 5 has a `GO-N` function in lieu of the `GO-E` function found in great-grandparent 1 from generation 5. This `GO-N` function in generation 4 was actually superior to the `GO-E` function found in its offspring in generation 5 because it prevented the counter-productive and regressive movement back to the east. Consequently, great-great-grandparent 1 from generation 4 scored 20 hits, whereas its offspring in generation 5 scored 17 hits. This highlights the fact that genetic algorithms are not mere hill-climbers. Crossover most typically recombines parts of individuals that are not the current optimum to produce an improved offspring in the next generation.

7.2 Fitness Curve Showing Progressive Learning

Figure 10 shows that, as we proceed from generation to generation, the fitness of the best-of-generation individual and the average fitness of the population as a whole tends to progressively improve (i.e., drop).

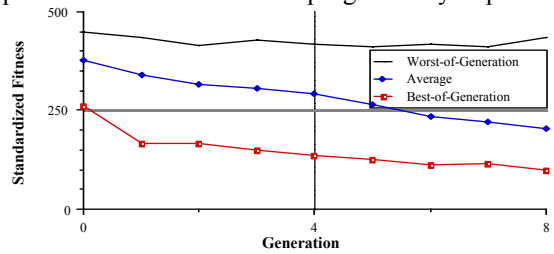


Figure 10 Fitness curves

7.3 Hits Histogram Showing Progressive Learning

The hits histogram is a useful monitoring tool for visualizing the progressive learning of the population as a whole during a run. The horizontal axis of the hits histogram represents the number of hits (0 to 30) while the vertical axis represents the number of individuals in the population (0 to 500) scoring that number of hits.

Figure 11 shows the hits histograms for generations 0, 4, and 8 of this particular run. Note, in the progression the left-to-right undulating movement of both the high point and the center of mass of these three histograms. This "slinky" movement reflects the improvement of the population as a whole.

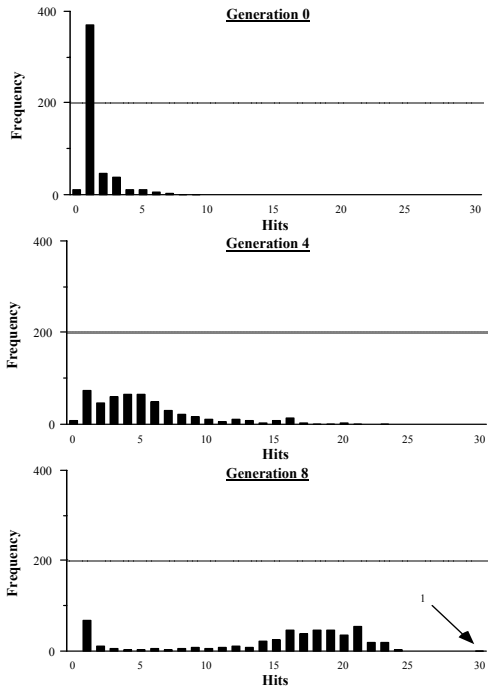


Figure 11 Hits histograms for generations 0, 4, and 8

7.4 Retesting of Best-of-Run Individual

As it happens, the best-of-run individual from generation 8 from the particular run described above was sufficiently parsimonious and understandable that we were able to satisfy ourselves that it indeed is a solution to the general painted desert problem. However, programs that score 100% are generally not as parsimonious or as understandable as the above best-of-run individual. We envision a general "painted desert" problem and we think of the randomly selected starting arrangement for the ants and sand in figure 1 as one instance of the possible starting positions for this general problem. However, there is no guarantee that a computer program that is evolved while being exposed to only one random starting arrangement of ants and sand (or any incomplete sampling of starting arrangements) will necessarily generalize so as to correctly handle all other possible starting arrangements. Genetic programming knows nothing about the more general problem or about the larger class of potential starting positions. There is a possibility that the best-of-run individual from generation 8 described above is idiosyncratically adapted (i.e., overfitted) to the one random starting arrangement (i.e., fitness case) shown in figure 1. We can acquire evidence as to whether the solution evolved by genetic programming generalizes to the more general problem that we envision by retesting the above best-of-run individual scoring 100% using different random starting positions. When we retested this individual on five different random starting positions, we found that it correctly solved the general "painted desert" problem.

8 Results over a Series of Runs

In the foregoing section, we illustrated genetic programming by describing a particular successful run. In this section, we discuss the performance of genetic programming over a series of 32 runs. The solutions evolved in different runs exhibit very different behavior from the solution previously described. Some solutions, for example, tend to move the ants and sand in north-westerly, south-easterly, or north-easterly directions while others exhibit no directional bias. Other solutions completely fill the third vertical column with 10 grains of striped sand before placing much sand in the first and second columns. Moreover, all of the other solutions are considerably less parsimonious and less easily understood than the particular run described above. The number of points in the nine solutions found in the 32 runs averaged 92.6 points, as compared to the 15 points found in the unusually parsimonious solution discussed in detail in the previous section. The crossover operations in those other runs almost always recombined larger subtrees, rather than individual terminals.

8.1 Performance Curves

Figure 12 presents two curves, called the performance curves, for the painted desert problem over a series of 32 runs. The curves are based on a population size, *M*, of 500 and a maximum number of generations to be run, *G*, of 51.

The rising curve in this figure shows, by generation, the experimentally observed cumulative probability of success, $P(M,i)$, of yielding an acceptable result for this problem by generation i. We define an acceptable result for this problem to be at least one S-expression in the population which scores 30 hits (this being equivalent to scoring a value of fitness of 100). As can be seen, the experimentally observed value of the cumulative probability of success, $P(M,i)$, is 25% by generation 36 and 28% by generation 50 over the 32 runs.

The second curve (which first falls and then rises) in this figure shows, by generation, the number of individuals that must be processed, $I(M,i,z)$, to yield, with probability z, a solution to the problem by generation i. $I(M,i,z)$ is derived from the experimentally observed values of $P(M,i)$. Specifically, $I(M,i,z)$ is the product of the population size M, the generation number i, and the number of independent runs, $R(z)$, necessary to yield a solution to the problem with probability z by generation i. In turn, the number of runs $R(z)$ is given by

$$R(z) = \left\lceil \frac{log(1-z)}{log(1-P(M,i))} \right\rceil ,$$

where the brackets indicates the ceiling function for rounding up to the next highest integer. The probability z is 99% herein.

As can be seen, the $I(M,i,z)$ curve reaches a minimum value at generation 36 (highlighted by the light dotted vertical line). For a value of $P(M,i)$ of 25%, the number of independent runs $R(z)$ necessary to yield a solution to the problem with a 99% probability by generation i is 14. The two summary numbers (i.e., 36 and 259,000) in the oval in this figure indicate that if this problem is run through to generation 36 (the initial random generation being counted as generation 0), processing a total of 259,000 individuals (i.e., 500 ∞ 37 generations ∞ 14 runs) is sufficient to yield a solution to this problem with 99% probability. This number, 259,000, is a measure of the computational effort necessary to yield a solution to this problem with 99% probability.

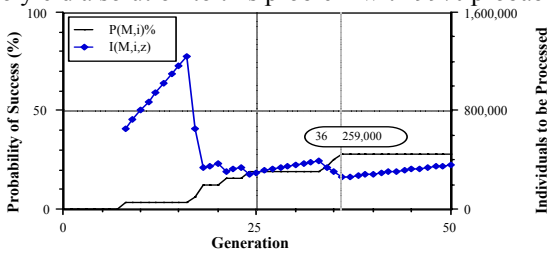


Figure 12 Performance curves showing that it is sufficient to process 259,000 individuals to yield a solution with 99% probability

8.2 Fitness Curves over a Series of Runs

We have already seen, for the single run of this problem presented above, that the fitness of the best-of-generation individual and the average fitness of the population tended to progressively improve via small, evolutionary improvements from generation to generation. The same is true when fitness is tracked over a series of runs.

Figure 13 shows the incremental learning behavior of genetic programming by depicting, by generation, the average, over the 32 runs, of the fitness of the best-of-generation individual and the average of the average fitness of the population as a whole (the "average average" fitness).

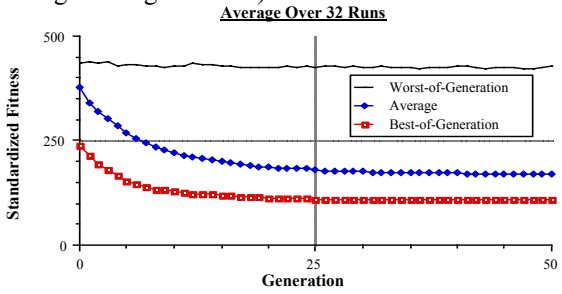


Figure 13 Fitness curves over a series of 32 runs

8.3 Structural Complexity Curves over a Series of Runs

Figure 14 shows, by generation, the average, over the 32 runs, of the structural complexity of the best-of-generation individual and the average of the average structural complexity of the population as a whole (the "average average" structural complexity). We can see that as fitness improves over the generations, the structural complexity changes.

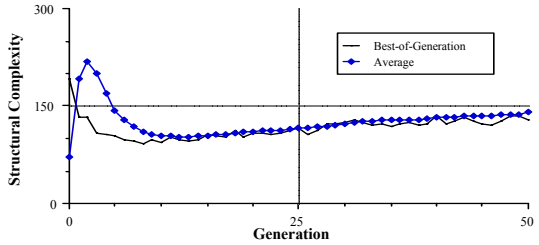


Figure 14 Structural complexity curves over a series of 32 runs

8.4 Variety Curve over a Series of Runs

Figure 15 shows, by generation, the average, over the 32 runs, of the variety of the population (i.e., the percentage of unique individuals). As can be seen, average variety starts at 100% for generation (because we specifically remove duplicates from the initial random population) and remains near 100% over the generations with genetic programming (i.e., there is no loss of genetic diversity or premature convergence).

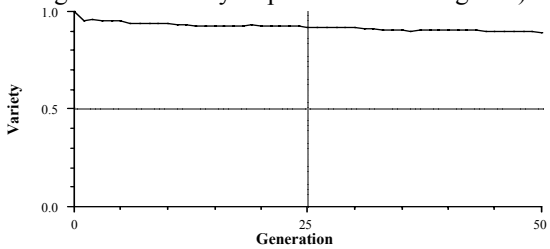


Figure 15 Variety curve over a series of 32 runs

9 Conclusions

We have shown that a computer program to solve a version of the painted desert problem can be evolved using genetic programming. The computer program arises from the selective pressure exerted by a Darwinian reproduction operation, a genetic crossover (recombination) operation, and a fitness measure appropriate to the problem.

Past research has indicated that surprisingly complex behavior can emerge from the repetitive application of seemingly simple, locally applied rules. We have shown here that complex overall behavior, such as the behavior required to solve the painted desert problem, can be evolved using a domain-independent genetic method based on the Darwinian principle of survival of the fittest and genetic operations. The fact that such emergent behavior can be evolved for this problem is suggestive of the fact that a wide variety of other complex behaviors can be evolved.

ACKNOWLEDGEMENTS

James P. Rice of the Knowledge Systems Laboratory at Stanford University made numerous contributions in connection with the computer programming of the above. Simon Handley of the Computer Science Department at Stanford University made helpful comments on this chapter.

REFERENCES

Belew, Richard and Booker, Lashon (editors) *Proceedings of the Fourth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc. 1991.

Collins, Rob and Jefferson, David. Ant Farm: Toward simulated evolution. In Langton, Christopher, Taylor, Charles, Farmer, J. Doyne, and Rasmussen, Steen (editors). *Artificial Life II, SFI Studies in the Sciences of Complexity*. Volume X. Redwood City, CA: Addison-Wesley 1991. Pages 579-601. .

Davidor, Yuval. *Genetic Algorithms and Robotics*. Singapore: World Scientific 1991.

Davis, Lawrence (editor) *Genetic Algorithms and Simulated Annealing.* London: Pittman l987.

Davis, Lawrence. *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold 1991.

Deneubourg, J. L., Aron, S., Goss, S., Pasteels, J. M., and Duerinck, G. Random behavior, amplification processes and number of participants: How they contribute to the foraging properties of ants. In Farmer, Doyne, Lapedes, Alan, Packard, Norman, and Wendroff, Burton (editors). *Evolution, Games, and Learning*. Amsterdam: North-Holland 1986. Pages 176-186.

Deneubourg, J. L., Goss, S., Franks, N., Sendova-Franks, A., Detrain, C., and Chretien, L. The dynamics of collective sorting robot-like ants and ant-like robots. In Meyer, Jean-Arcady, and Wilson, Stewart W. (editors).

From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior. Cambridge, MA: The MIT Press 1991. Pages 356-363.

Forrest, Stephanie (editor). *Emergent Computation: Self-Organizing, Collective, and Cooperative Computing Networks*. Cambridge, MA: The MIT Press 1990. Also in *Physica D* 1990.

Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley l989.

Holland, John H. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press 1975. Revised Second Edition 1992 from The MIT Press.

Koza, John R. Genetic evolution and co-evolution of computer programs. In Langton, Christopher, Taylor, Charles, Farmer, J. Doyne, and Rasmussen, Steen (editors). *Artificial Life II, SFI Studies in the Sciences of Complexity*. Volume X. Redwood City, CA: Addison-Wesley 1991. Pages 603-629. 1991.

Koza, John R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: The MIT Press 1992. 1992a.

Koza, John R. The genetic programming paradigm: Genetically breeding populations of computer programs to solve problems. In Soucek, Branko and the IRIS Group (editors). *Dynamic, Genetic, and Chaotic Programming*. New York: John Wiley 1992. 1992b.

Koza, John R. Evolution of subsumption using genetic programming. In Varela, Francisco J., and Bourgine, Paul (editors). *Toward a Practice of Autonomous Systems: Proceedings of the first European Conference on Artificial Life*. Cambridge, MA: The MIT Press 1992. Pages 110-119. 1992c.

Koza, John R. and Rice, James P. *Genetic Programming: The Movie.* Cambridge, MA: The MIT Press 1992a.

Koza, John R., and Rice, James P. Automatic programming of robots using genetic programming. In *Proceedings of Tenth National Conference on Artificial Intelligence*. Menlo Park, CA: AAAI Press / The MIT Press 1992. Pages 194-201. 1992b.

Koza, John R., Rice, James P., and Roughgarden, Jonathan. Evolution of food foraging strategies for the Caribbean *Anolis* lizard using genetic programming. *Simulation of Adaptive Behavior*. Volume 1, number 2, pages 47-74. 1992.

Langton, Christopher, Taylor, Charles, Farmer, J. Doyne, and Rasmussen, Steen (editors). *Artificial Life II, SFI Studies in the Sciences of Complexity*. Volume X. Redwood City, CA: Addison-Wesley 1991.

Maenner, Reinhard, and Manderick, Bernard. *Proceedings of the Second International Conference on Parallel Problem Solving from Nature*. North Holland 1992.

Manderick, Bernard, and Moyson, Frans. The collective behavior of ants: An example of self-organization in massive parallelism. In *Proceedings of the AAAI Spring Symposium on Parallel Models of Intelligence.* Stanford, CA: American Association of Artificial Intelligence. 1988.

Meyer, Jean-Arcady, and Wilson, Stewart W. *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior,* Cambridge, MA: MIT Press 1991.

Michalewicz, Zbignlew. *Genetic Algorithms + Data Structures = Evolution Programs*. Berlin: Springer-Verlag 1992.

Moyson, Frans, and Manderick, Bernard. Self-organization versus programming in massively parallel systems: A case study. In Eckmiller, R. (editor). *Proceedings of the International Conference on Parallel Processing in Neural Systems and Computers* (ICNC-90), 1990.

Resnick, Mitchel. Animal simulations with *Logo: Massive parallelism for the masses. In Meyer, Jean-Arcady, and Wilson, Stewart W. *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior.* Cambridge, MA: The MIT Press 1991. Pages 534-539.

Schwefel, Hans-Paul and Maenner, Reinhard (editors). *Parallel Problem Solving from Nature.* Berlin: Springer-Verlag 1991.

Steels, Luc. Cooperation between distributed agents using self-organization. In Demazeau, Y., and Muller, J-P (editors). *Decentralized AI*. Amsterdam: North-Holland 1990.

Steels, Luc. Toward a theory of emergent functionality. In Meyer, Jean-Arcady, and Wilson, Stewart W. *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior.* Cambridge, MA: The MIT Press 1991. Pages 451-461.

Travers, Michael, and Resnick, Mitchel. Behavioral dynamics of an ant colony: Views from three Levels. In Langton, Christopher G. (editor). *Artificial Life II Video Proceedings*. Addison-Wesley 1991.

Varela, Francisco J., and Bourgine, Paul (editors). *Toward a Practice of Autonomous Systems: Proceedings of the first European Conference on Artificial Life*. Cambridge, MA: The MIT Press 1992.

Whitley, Darrell (editor). *Proceedings of Workshop on the Foundations of Genetic Algorithms and Classifier Systems*, *Vail, Colorado 1992.* San Mateo, CA: Morgan Kaufmann Publishers Inc. 1992.