Revised November 29, 1990 for *Proceedings of Second Conference on Artificial Life (AL-2)*

GENETIC EVOLUTION AND CO-EVOLUTION OF COMPUTER PROGRAMS

John R. Koza

Computer Science Department

Margaret Jacks Hall

Stanford University

Stanford, CA 94305 USA

E-MAIL: Koza@Sunburn.Stanford.Edu

PHONE: 415-941-0336

FAX: 415-941-9430

INTRODUCTION AND OVERVIEW

Research in the field of artificial life focuses on computer programs that exhibit some of the properties of biological life (e.g. self-reproducibility, evolutionary adaptation to an environment, etc.). In one area of artificial life research, human programmers write intentionally simple computer programs (often incorporating observed features of actual biological processes) and then study the "emergent" higher level behavior that may be exhibited by such seemingly simple programs. In this chapter, we consider a different problem, namely, "How can computer programs be automatically written by the computer itself using only measurements of a given program's performance?" In particular, this chapter describes the recently developed "genetic programming paradigm" which genetically breeds populations of computer programs in order to find a computer program that solves the given problem. In the genetic programming paradigm, the individuals in the population are hierarchical compositions of functions and arguments. The hierarchies are of various sizes and shapes. Increasingly fit hierarchies are then evolved in response to the problem environment using the genetic operations of fitness proportionate reproduction (Darwinian survival and reproduction of the fittest) and crossover (sexual recombination). In the genetic programming paradigm, the size and shape of the hierarchical solution to the problem is not specified in advance. Instead, the size and shape of the hierarchy, as well as the contents of the hierarchy, evolve in response to the Darwinian selective pressure exerted by the problem environment.

This chapter also describes an extension of the genetic programming paradigm to the case where two (or more) populations of hierarchical computer programs simultaneously co-evolve. In co-evolution, each population acts as the environment for the other population. In particular, each individual of the first population is evaluated for "relative fitness" by testing it against each individual in the second population, and, simultaneously, each individual in the second population is evaluated for relative fitness by testing them against each individual in the first population. Over a period of many generations, individuals with high "absolute fitness" may

evolve as the two populations mutually bootstrap each other to increasingly high levels of fitness.

The genetic programming paradigm is illustrated by genetically breeding a population of hierarchical computer programs to allow an "artificial ant" to traverse an irregular trail. In addition, we genetically breed a computer program controlling the behavior of an individual ant in an ant colony which, when simultaneously executed by a large number of ants, causes the emergence of interesting collective behavior for the colony as a whole. Co-evolution is illustrated with a problem involving finding an optimal strategy for playing a simple discrete two-person competitive game represented by a game tree in extensive form.

BACKGROUND ON GENETIC ALGORITHMS

Genetic algorithms are highly parallel mathematical algorithms that transform populations of individual mathematical objects (typically fixed-length binary character strings) into new populations using operations patterned after natural genetic operations such as sexual recombination (crossover) and fitness proportionate reproduction (Darwinian survival of the fittest). Genetic algorithms begin with an initial population of individuals (typically randomly generated) and then iteratively (1) evaluate the individuals in the population for fitness with respect to the problem environment and (2) perform genetic operations on various individuals in the population to produce a new population. John Holland of the University of Michigan presented the pioneering formulation of genetic algorithms for fixed-length character strings in 1975 in *Adaptation in Natural and Artificial Systems* (6). 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 in the search space, given currently available information. Recent work in genetic algorithms and genetic classifier systems can be surveyed in Goldberg (4), Davis (2), and Schaffer (14).

BACKGROUND ON GENETIC PROGRAMMING PARADIGM

Representation is a key issue in genetic algorithm work because genetic algorithms directly manipulate the coded representation of the problem and because the representation scheme can severely limit the window by which the system observes its world. Fixed length character strings present difficulties for some problems — particularly problems where the desired solution is hierarchical and where the size and shape of the solution is unknown in advance. The need for more powerful representations has been recognized for some time (3).

The structure of the individual mathematical objects that are manipulated by the genetic algorithm can be more complex than the fixed length character strings first described by Holland (6) in 1975. Steven Smith (16) departed from the early fixed-length character strings by introducing variable length strings, specifically, strings whose elements were if-then rules, rather than single characters. Holland's introduction of the genetic classifier system (7) continued the trend towards increasing the complexity of the structures undergoing adaptation. The classifier system is a cognitive architecture containing a population of string-based if-then rules (whose condition and action parts are fixed length binary strings) which can be modified by the genetic algorithm.

The recently developed genetic programming paradigm further continues the above trend towards increasing the complexity of the structures undergoing adaptation. In the genetic programming paradigm, the individuals in the population are hierarchical compositions of functions and terminals appropriate to the particular problem domain. The hierarchies are of various sizes and shapes. The set of functions typically includes arithmetic operations, mathematical functions, conditional logical operations, and domain-specific functions. Each function in the function set should be well defined for any element in the range of every other function in the set. The set of terminals used typically includes inputs (sensors) appropriate to the problem domain and various constants. The search space is the hyperspace of all possible compositions of functions and terminals that can be recursively composed of the available func-

tions and terminals. The symbolic expressions (S-expressions) of the LISP programming language are an especially convenient way to create and manipulate the compositions of functions and terminals described above. These S-expressions in LISP correspond directly to the "parse tree" that is internally created by most compilers.

The basic genetic operations for the genetic programming paradigm are fitness based reproduction and crossover (recombination).

Fitness proportionate reproduction is the basic engine of Darwinian reproduction and survival of the fittest. It copies individuals with probability proportionate to fitness from one generation of the population into the next generation. In this respect, it operates for the genetic programming paradigm in the same way as it does for conventional genetic algorithms. The crossover operation for the genetic programming paradigm is a sexual operation that operates on two parental LISP S-expressions and produces two offspring S-expressions using parts of each parent. Typically the two parents are hierarchical compositions of functions of different size and shape. In particular, the crossover operation starts by selecting a random crossover point in each parent and then creates two new offspring S-expressions by exchanging the sub-trees (i.e. sub-lists) between the two parents. Because entire sub-trees are swapped, this genetic crossover (recombination) operation produces syntactically and semantically valid LISP S-expressions as offspring regardless of which point is selected in either parent.

For example, consider the parental LISP S-expression:

```
(OR (NOT D1) (AND D0 D1))
```

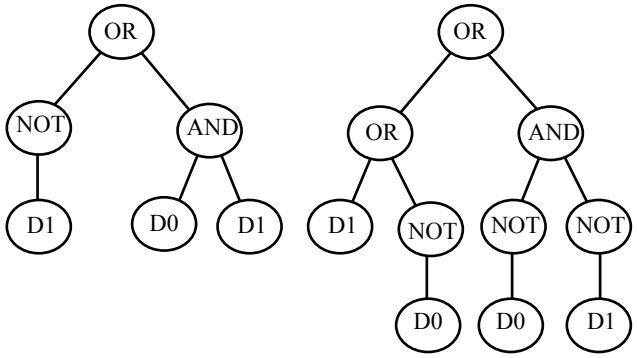
And, consider the second parental S-expression below:

```
(OR (OR D1 (NOT D0))
    (AND (NOT D0) (NOT D1))
```

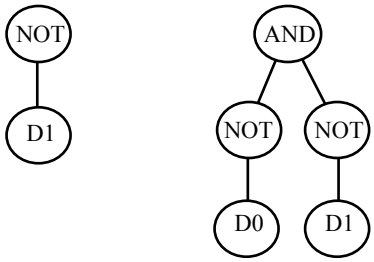
These two LISP S-expressions can be depicted graphically as rooted, point-labeled trees with ordered branches. Assume that the points of both trees are numbered in a depth-first way starting at the left. Suppose that the second point (out of 6 points of the first parent) is randomly selected as the crossover point for the first parent and that the sixth point (out of 10 points of the second

parent) is randomly selected as the crossover point of the second parent. The crossover points are therefore the NOT in the first parent and the AND in the second parent.

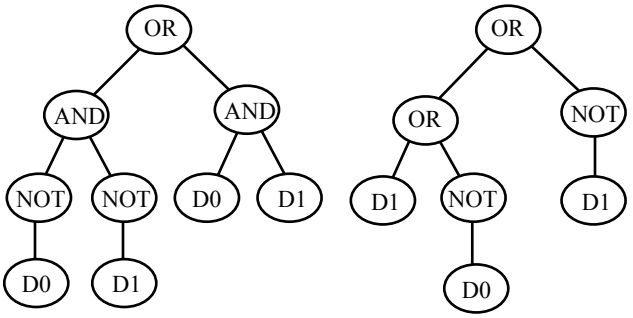
The two parental LISP S-expressions are shown below:



The two crossover fragments are two sub-trees shown below:



These two crossover fragments correspond to the bold, underlined sub-expressions (sub-lists) in the two parental LISP S-expressions shown above. The two offspring resulting from crossover are shown below.

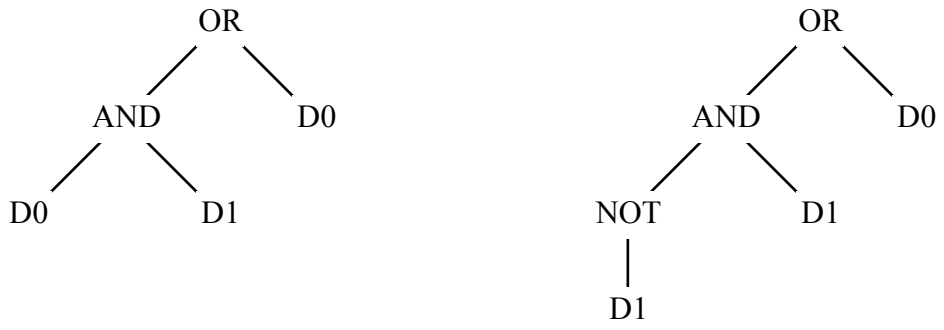


Note that the first offspring above is a perfect solution for the exclusive-or function, namely

```
(OR (AND (NOT D0) (NOT D1)) (AND D0 D1)).
```

In addition to the basic genetic operations of fitness proportionate reproduction and crossover, a

mutation operation can also be defined to provide a means for occasionally introducing small random mutations into the population. The mutation operation is an asexual operation in that it operates on only one parental S-expression. selected based on fitness. The result of this operation is a single offspring S-expression. The mutation operation selects a point of the LISP S-expression at random. The point can be an internal (function) or external (terminal) point of the tree. This operation removes whatever is currently at the selected point and inserts a randomly generated sub-tree at the randomly selected point of a given tree. This randomly generated subtree is created in the same manner as the initial random individuals in the initial random generation. This operation is controlled by a parameter which specifies the maximum depth for the newly created and inserted sub-tree. A special case of this operation involves inserting only a single terminal (i.e. a sub-tree of depth 0) at a randomly selected point of the tree. For example, in the figure below, the third point of the S-expression shown on the left below was selected as the mutation point and the sub-expression (NOT D1) was randomly generated and inserted at that point to produce the S-expression shown on the right below.



Before After

The mutation operation potentially can be beneficial in reintroducing diversity in a population that may be tending to prematurely converge.

Additional details can be found in Koza (10,11).

We have shown that entire computer programs can be genetically bred to solve problems in a

variety of different areas of artificial intelligence, machine learning, and symbolic processing (10, 11). In particular, this new paradigm has been successfully applied to example problems in several different areas, including

- machine learning of functions (e.g. learning the Boolean 11-multiplexer function),
- planning (e.g. developing a robotic action sequence that can stack an arbitrary initial configuration of blocks into a specified order),
- automatic programming (e.g. discovering a computational procedure for solving pairs of linear equations, solving quadratic equations for complex roots, and discovering trigonometric identities),
- sequence induction (e.g. inducing a recursive computational procedure for the Fibonacci and the Hofstadter sequences),
- pattern recognition (e.g. translation-invariant recognition of a simple one-dimensional shape in a linear retina),
- optimal control (e.g. centering a cart and balancing a broom on a moving cart in minimal time by applying a "bang bang" force to the cart)
- symbolic "data to function" regression, symbolic "data to function" integration, and symbolic "data to function" differentiation,
- symbolic solution to functional equations (including differential equations with initial conditions, integral equations, and general functional equations),
- empirical discovery (e.g. rediscovering Kepler's Third Law, rediscovering the well-known 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),
- finding the minimax strategy for a differential pursuer-evader game, and
- simultaneous architectural design and training of neural networks.

The genetic programming paradigm permits the evolution of computer programs which can perform alternative computations conditioned on the outcome of intermediate calculations, which can perform computations on variables of many different types, which can perform iterations and recursions to achieve the desired result, which can define and subsequently use computed values and sub-programs, and whose size, shape, and complexity is not specified in advance.

THE "ARTIFICIAL ANT" PROBLEM

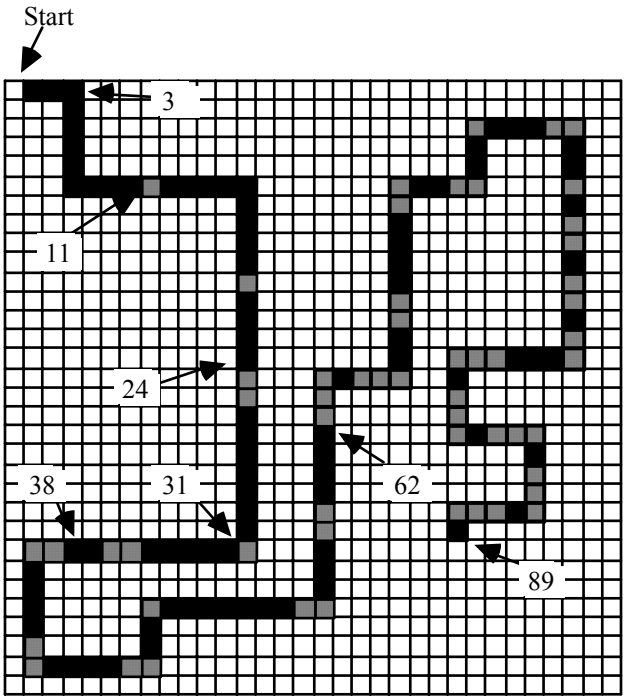
In order to illustrate the genetic programming paradigm, we consider the complex planning task devised by Jefferson et. al. (9) for an "artificial ant" attempting to traverse a trail.

Jefferson et. al. successfully solved a string-based genetic algorithm to discover a finite state automaton enabling the "artificial ant" to traverse the trail.

The setting for the problem is a square 32 by 32 toroidal grid in the plane. The "Santa Fe trail" is a winding trail with food in 89 of the 1024 cells. This trail (designed by Christopher Langton) is considered the more difficult of the two trails tested by Jefferson et. al.. The trail is irregular and has single gaps, double gaps, single gaps at some corners, double gaps (knight moves) at other corners , and triple gaps (long knight moves) at other corners . The "artificial ant" begins in the cell identified by the coordinates (0,0) and is facing in a particular direction (i.e. east). The artificial ant has a sensor that can see only the single adjacent cell in the direction the ant is currently facing. At each time step, the ant has the capacity to execute any of four operations, namely, to move forward (advance) in the direction it is facing, to turn right (and not move), to turn left (and not move), or to sense the contents of the single adjacent cell in the direction the ant is facing.

The objective of the ant is to traverse the entire trail and collect all of the food. Jefferson, Collins et. al. limited the ant to a certain number of time steps (200).

Jefferson, Collins et. al. started by assuming that the finite automaton necessary to solve the problem would have 32 or fewer states. They then represented an individual in their population of automata by a 453-bit string representing the state transition diagram (and its initial state) of the individual automaton. The ant's sensory input at each time step was coded as one bit and the output at each time step was coded as two bits. The next state of the automaton was coded with 5 bits. The complete behavior of an automaton was thus specified with a genome consisting of a binary string with 453 bits (5 bits representing the initial state of the automaton plus 64 substrings of length 7 representing the state transitions). Jefferson, Collins et. al. then processed a population of 65,536 individual bit strings of length 453 on a Connection Machine™ using a genetic algorithm using crossover and mutation operating on a selected (relatively small) fraction of the population. After 200 generations in a particular run (taking about 10 hours on the Connection Machine), they reported that a single individual in the population emerged which attained a perfect score of 89 stones. As it happened, this single individual completed the task in exactly 200 operations.



In our approach to this task using the genetic programming paradigm, we used the function set consisting of the functions F = {IF-SENSOR, PROGN}. The IF-SENSOR function has two arguments and evaluates the first argument if the ant's sensor senses a stone or, otherwise, evaluates the second argument. The PROGN function is the LISP connective (glue) function that sequentially evaluates its arguments as individual steps in a program. The terminal set was T = {ADVANCE, TURN-RIGHT, TURN-LEFT}. These three terminals are actually functions with no arguments. They

operate via their side effects on the ant's state (i.e. the ant's position on the grid or the ant's facing direction). Note that IF-SENSOR, ADVANCE, TURN-RIGHT, and TURN-LEFT correspond directly to the operators defined and used by Jefferson, Collins et. al. We allowed 400 time steps before timing out. Note that we made no assumption about the complexity of the eventual solution. The complexity of this problem is such that it cannot be solved by any random search method using either method.

The initial generation (generation 0) consisted of randomly generated individual S-expressions recursively created using the available functions and available terminals of the problem. Many of these randomly generated individual did nothing at all. For example, (PROGN (TURN-RIGHT) (TURN-RIGHT)) turns without ever moving the ant anywhere. Other random individuals move without turning [e.g. (ADVANCE)]. Other individuals in the initial random population move forward after sensing food but can only handle a right turn in the trail [e.g. (IF-SENSOR (ADVANCE) (TURN-RIGHT))].

Throughout this chapter (and in virtually all of our experiments), each new generation was created from the preceding generation by applying the fitness proportionate reproduction operation to 10% of the population and by applying the crossover operation to 90% of the population (with reselection allowed). The selection of crossover points in the population was biased 90% towards internal (function) points of the tree and 10% towards external (terminal) points of the tree. For practical reasons (i.e. conservation of computer time), a limit of 4 was placed on the depth of initial random S-expressions and a limit of 15 was placed on the depth of S-expressions created by crossover. As to mutation, our experience has been that no run using only mutation and fitness proportionate reproduction (i.e. no crossover) ever produced a solution to any problem (although such solutions are theoretically possible given enough time). In other words, "mutating and saving the best" does not work any better for hierarchical genetic algorithms than it does for string-based genetic algorithms. This conclusion as to the relative unimportance of the mutation operation is similar to the conclusions reached by most other research work on string-

based genetic algorithms [see, for example, Holland (6) and Goldberg (4)]. Accordingly, mutation was not used here.

In one run, a reasonably parsimonious individual LISP S-expression scoring 89 out of 89 emerged on the seventh generation, namely,

```
(IF-SENSOR (ADVANCE)
  (PROGN (TURN-RIGHT)
         (IF-SENSOR (ADVANCE) (TURN-LEFT))
         (PROGN (TURN-LEFT)
                (IF-SENSOR (ADVANCE)
                  (TURN-RIGHT))
                (ADVANCE)))).
```

This plan is graphically depicted in Figure 1.

This individual LISP S-expression is the solution to the problem. In particular, this plan moves the ant forward if a stone is sensed. Otherwise it turns right and then moves the ant forward if a stone is sensed but turns left (returning to its original orientation) if no stone is sensed. Then it turns left and moves forward if a stone is sensed but turns right (returning to its original orientation) if no stone is sensed. If the ant originally did not sense a stone, the ant moves forward unconditionally as its fifth operation. Note that there is no testing of the backwards directions.

We can measure the performance of a probabilistic algorithm by estimating the expected number of individuals that need to be processed by the algorithm in order to produce a solution

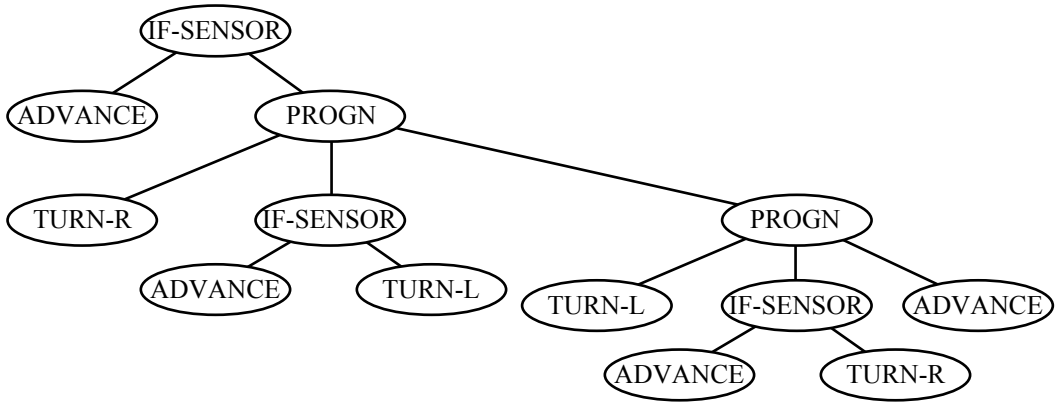
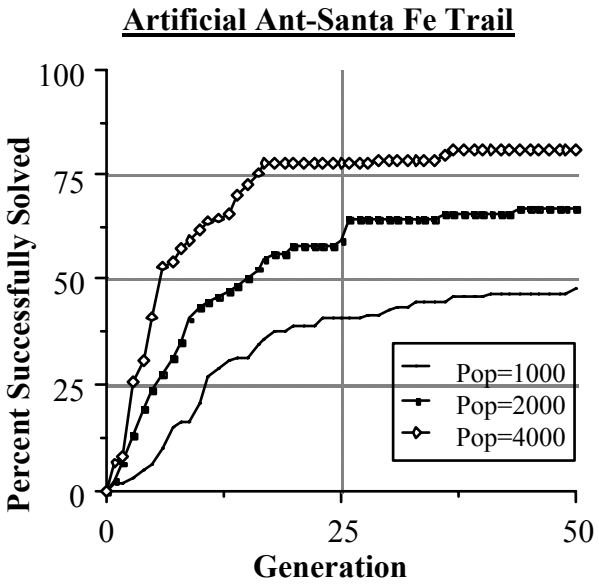


Figure. 1 Artificial Ant Solution

to the given problem with a certain probability (say 99%). Suppose, for example, a particular run of a genetic algorithm produces the desired result with only a probability of success p_S after a specified choice (perhaps arbitrary and non-optimal) of number of generations N_{gen} and population of size N. Suppose also that we are seeking to achieve the desired result with a probability of, say, $z = 1 - \varepsilon = 99\%$. Then, the number K of independent runs required is

$$K = \frac{\log (1-z)}{\log (1-p_S)} = \frac{\log \varepsilon}{\log (1-p_S)}, \text{ where } \varepsilon = 1-z.$$

For example, we ran 111 runs of the Artificial Ant problem with a population size of 1000 and 51 generations. We found that the probability of success p_S on a particular single run was 43%. With this probability of success, K = 8 independent runs are required to assure a 99% probability of solving the problem on at least one of the runs. That is, it is sufficient to process 408,000 individuals to achieve the desired 99% probability. Processing a full 408,000 individuals requires about 6 hours of computing time on the Texas Instruments Explorer II+™ workstation for this problem. In addition, as the graph below shows, the probability of success p_S of a run with a population size of 2000 with 51 generations is 67% so that a population of 2000 requires K = 4 independent runs (i.e. 408,000 individuals to be processed) to achieve the desired 99% probability. In contrast, the probability of success of a run with a population of 4000 with 51 generations.is 81% so that a population of 4000 requires K = 3 independent runs (i.e. 612,000 individuals to be processed) to achieve the desired 99% probability.



The 6 hours of computer time required to process the 408,000 individuals required by this problem (either with a population of size 1000 or 2000) on the considerably smaller and slower serial computer (Explorer workstation) represents substantially less computational resources than even a single 10-hour run on the massively parallel Connection Machine with 65,536 processors (even if it were the case that all such 10-hour runs on the Connection Machine were successful in solving the problem). Thus, the genetic programming paradigm is comparatively speedy for this problem.

EMERGENCE OF COLLECTIVE BEHAVIOR IN AN ANT COLONY

Conway's "game of life" and other work in cellular automata, fractals, chaos, and Lindenmayer systems (L-systems) are suggestive of how the repetitive application of seemingly simple rules can lead to complex "emergent" overall behavior. In this section, we present an example of how such rules (computer programs) can be evolved using genetic recombination and the Darwinian principle of survival of the fittest as contained in the genetic programming paradigm.

In particular, we show the emergence of interesting collective behavior in a colony of ants by genetically breeding a computer program to govern the behavior of the individual ants in the colony. The goal is to genetically evolve a common computer program governing the behavior of

the individual ants in a colony such that the collective behavior of the ants consists of efficient transportation of food to the nest. In nature, when an ant discovers food, it deposits a trail of pheromones as it returns to the nest with the food. The pheromonal cloud (which dissipates over time) aids other ants in efficiently locating and exploiting the food source.

In this example, 144 pellets of food are piled eight deep in two 3-by-3 piles located in a 32-by-32 toroidal area. There are 20 ants. The state of each ant consists of its position and the direction it is facing (out of eight possible directions). Each ant initially starts at the nest and faces in a random direction. Each ant in the colony is governed by a common computer program associated with the colony. The computer program is a composition of the following nine available functions:

- MOVE-RANDOM randomly changes the direction in which an ant is facing and then moves the ant two steps in the new direction.
- MOVE-TO-NEST moves the ant one step in the direction of the nest. This implements the gyroscopic ability of ants to navigate back to their nest.
- PICK-UP picks up food (if any) at the current position of the ant.
- DROP-PHEROMONE drops a pheromone at the current position of the ant (if the ant is carrying food). The pheromone immediately forms a 3-by-3 cloud around the drop point. The cloud decays over a period of time.
- IF-FOOD-HERE is a two-argument function that executes its first argument is there is food at the ant's current position and, otherwise, executes the second (else) argument.
- IF-CARRYING-FOOD is a similar two-argument function that tests whether the ant is currently carrying food.
- MOVE-TO-ADJACENT-FOOD-ELSE is a one-argument function that allows the ant to test for immediately adjacent food and then move one step towards it. If food is present in more than one adjacent position, the ant moves to the position requiring the least change of

direction. If no food is adjacent, the "else" clause of this function is executed.

- MOVE-TO-ADJACENT-PHEROMONE-ELSE is a function similar to the above based on adjacent pheromone.
- PROGN is the LISP connective function that executes its argument in sequence

Each of the 20 ants in a given colony executes the colony's common computer program. Since the ants initially face in random directions, make random moves, and encounter a changing pattern of food and pheromones created by the activities of other ants, the 20 individual ants each have different states and pursue different trajectories.

The fitness of a colony is measured by how many of the 144 food pellets are transported to the nest within the "allotted time" (which limits both the total number of time steps and the total number of operations which any one ant can execute). The goal is to genetically evolve increasingly fit computer programs to govern the colony.

Mere random motion by the 20 ants in a colony will typically bring the ants into contact with only about 56 of the 144 food pellets within the allotted time. Moreover, the ants' task is substantially more complicated than merely coming in contact with food. After randomly stumbling into food, the ant must pick up the food. Then, the ant must move towards the nest. Moreover, while this sequence of behavior is desirable, it is still insufficient to efficiently solve the problem in any reasonable amount of time. It is also necessary that the ants that accidentally stumble into food must also establish a pheromonal trail as they carry the food back to the nest. Moreover, all ants must always be on the lookout for such pheromonal trails established by other ants and must follow such trails to the food when they encounter such trails. In a typical run, 93% of the random computer programs in the initial random generation did not transport even one of the 144 food pellets to the nest within the allotted time. About 3% of these initial random programs transported only one of the 144 pellets. Even the best single computer program of the random computer programs created in the initial generation successfully transported only 41 pellets.

As the genetic programming paradigm is run, the population as a whole and its best single individual both generally improve from generation to generation. In the one specific run which we describe in detail hereinbelow, the best single individual in the population on generation 10 scored 54; the best single individual on generation 20 scored 72; the best single individual on generation 30 scored 110; the best single individual on generation 35 scored 129; and, the best single individual on generation 37 scored 142.

On generation 38, a program emerged which causes the 20 ants to successfully transport all 144 food pellets to the nest within the allotted time. This 100% fit program is shown below:

```
(PROGN (PICK-UP) (IF-CARRYING-FOOD (PROGN (MOVE-TO-ADJACENT-PHEROMONE-
ELSE (MOVE-TO-ADJACENT-FOOD-ELSE (MOVE-TO-ADJACENT-FOOD-ELSE (MOVE-TO-
ADJACENT-FOOD-ELSE (PICK-UP))))) (PROGN (PROGN (PROGN (PROGN (MOVE-TO-
ADJACENT-FOOD-ELSE (PICK-UP)) (PICK-UP)) (PROGN (MOVE-TO-NEST) (DROP-
PHEROMONE))) (PICK-UP)) (PROGN (MOVE-TO-NEST) (DROP-PHEROMONE)))) (MOVE-
TO-ADJACENT-FOOD-ELSE (IF-CARRYING-FOOD (PROGN (PROGN (DROP-PHEROMONE)
(MOVE-TO-ADJACENT-PHEROMONE-ELSE (IF-CARRYING-FOOD (MOVE-TO-ADJACENT-
FOOD-ELSE (PICK-UP)) (MOVE-TO-ADJACENT-FOOD-ELSE (PICK-UP))))) (MOVE-TO-
NEST)) (IF-FOOD-HERE (PICK-UP) (IF-CARRYING-FOOD (PROGN (IF-FOOD-HERE
(MOVE-RANDOM) (IF-CARRYING-FOOD (MOVE-RANDOM) (PICK-UP))) (DROP-
PHEROMONE)) (MOVE-TO-ADJACENT-PHEROMONE-ELSE (MOVE-RANDOM)))))))))
```

The 100% fit program above is equivalent to the simplified program below (except for the special case when an ant is in the nest):

```
1 (PROGN (PICK-UP)
2       (IF-CARRYING-FOOD
3         (PROGN (MOVE-TO-ADJACENT-PHEROMONE-ELSE
4                  (MOVE-TO-ADJACENT-FOOD-ELSE (PICK-UP)))
5                (MOVE-TO-ADJACENT-FOOD-ELSE (PICK-UP))
6                (MOVE-TO-NEST) (DROP-PHEROMONE)
7                (MOVE-TO-NEST) (DROP-PHEROMONE))
8        (MOVE-TO-ADJACENT-FOOD-ELSE
9           (IF-FOOD-HERE
10             (PICK-UP)
11             (MOVE-TO-ADJACENT-PHEROMONE-ELSE (MOVE-RANDOM))))))
```

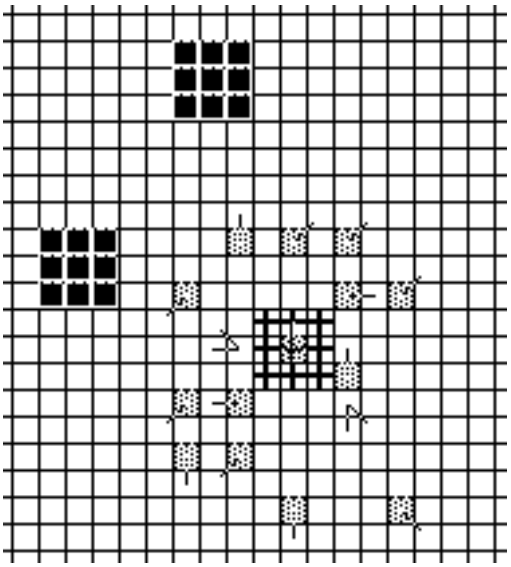
This simplified program can be interpreted as follows: The ant begins by picking-up the food, if any, located at the ant's current position. If the ant is now carrying food (line 2), then the six parts of the PROGN beginning on line 3 and ending on line 7 are executed. Line 3 moves the ant to the adjacent pheromone (if any). If there is no adjacent pheromone, line 4 moves the ant to

the adjacent food (if any). In view of the fact that the ant is already carrying food, these two potential moves on lines 3 and 4 generally distract the ant from the most direct return to the nest and therefore merely reduce efficiency. Line 5 is a similar distraction. Note that the PICK-UP operations on lines 4 and 5 are redundant since the ant is already carrying food.

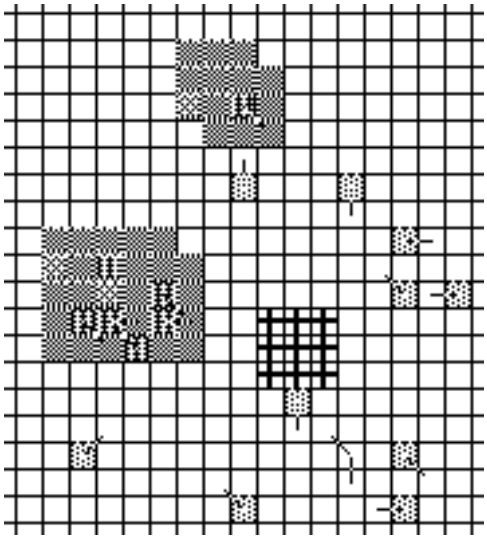
Given that the ant is already carrying food, the sequence of MOVE-TO-NEST and DROP-PHEROMONE on lines 6 and 7 is the winning combination that establishes the pheromone trail as the ant moves towards the nest with the food. The establishment of the pheromone trail between the pile of food and the nest is an essential part of efficient collective behavior for exploiting the food source.

The sequence of conditional behavior in lines 8 through 11 efficiently prioritizes the search activities of the ant. If the ant is not carrying food, line 8 moves the ant to adjacent food (if any). If there is no adjacent food but there is food at the ant's current position (line 9), the ant picks up the food (line 10). On the other hand, if there is no food at the ant's current position (line 11), the ant moves towards any adjacent pheromones (if any). If there are no adjacent pheromones, the ant moves randomly. This sequence of conditional behavior causes the ant to pick up any food it may encounter. Failing that, the second priority established by this conditional sequence causes the ant to follow a previously established pheromonal trail. And, failing that, the third priority of this conditional sequence causes the ant to move at random.

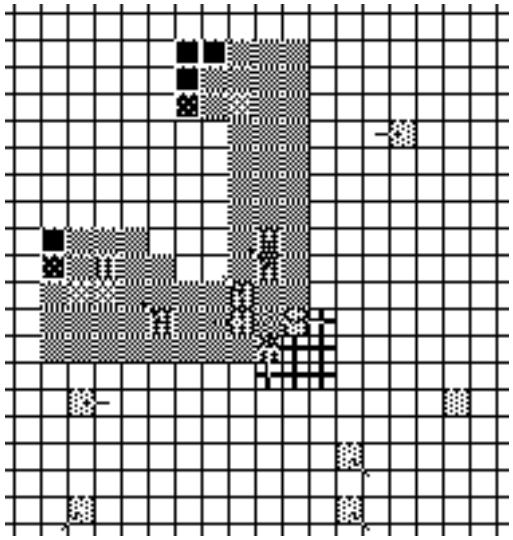
The collective behavior of the ant colony governed by the above 100% fit program above can be visualized as a series of major phases. The first phase occurs when the ants are dispersing from the nest and are randomly searching for food. In the figure below (representing time step 3 of one execution of the 100% fit program above), the two 3-by-3 piles of food are shown in black in the western and northern parts of the grid. The nest is indicated by nine + signs slightly southeast of the center of the grid. The ants are shown in gray with their facing direction indicated.



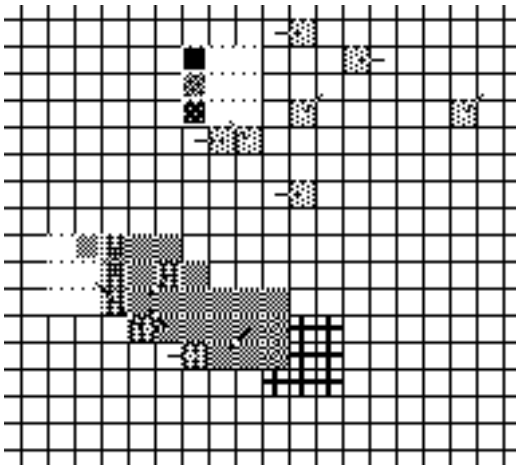
The second phase occurs when some ants have discovered some food, have picked up the food, and have started back towards the nest dropping pheromones as they go. The beginnings of the pheromonal clouds around both the western and northern pile of food are shown below (representing time step 12).



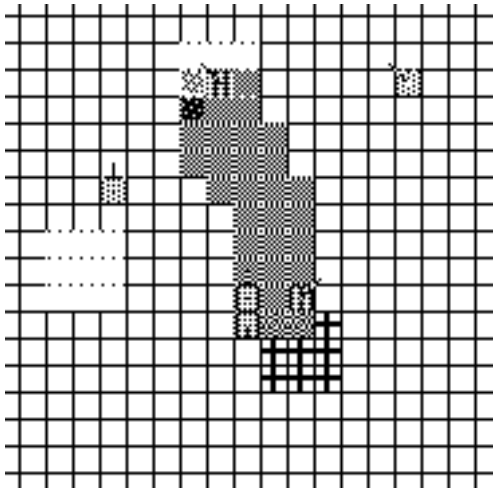
The third phase occurs when the pheromonal clouds have coalesced into recognizable pheromonal trails linking the piles of food with the nest. In the figure below (representing time step 15), the first two (of the 144) food pellets have just reached the nest.



The figure below shows the premature (and temporary) disintegration of the pheromonal trail connecting the northern pile of food with the nest while food still remains in the northern pile. The pheromonal trail connecting the western pile of food with the nest is still intact. 118 of the 144 food pellets have been transported to the nest at this point (representing time step 129).



The fourth phase occurs when the food in a given pile is entirely picked up and its pheromonal trail starts to dissolve. In the figure below (representing time step 152), the western pile has been entirely picked up by the ants and the pheromonal trail connecting it to the nest has already dissolved. The former location of the western pile is shown as the blank white area. 136 of the 144 food pellets have been transported to the nest at this point The pheromone trail connecting the nest and the northern pile (with 8 remaining food pellets) has been reestablished.



Shortly thereafter, the run ends with all 144 food pellets in the nest.

Note that the overall collective behavior of the ant colony observed above is not the result of a central controller orchestrating the activity. Rather, it emerges from the simultaneous execution, in parallel, of a common computer program by a large number of individual ants.

CO-EVOLUTION IN NATURE

The evolutionary process in nature is often described as if one population of individuals is trying to adapt to a fixed environment. This description is, however, only a first order approximation to the actual situation. The environment actually consists of both the physical environment (which is usually relatively unchanging) as well as other independently-acting biological populations of individuals which are simultaneously trying to adapt to "their" environment. The actions of each of these other independently-acting biological populations (species) usually affect all the others. In other words, the environment of a given species includes all the other biological species that contemporaneously occupy the physical environment and which are simultaneously trying to survive. In biology, the term "co-evolution" is sometimes used to reflect the fact that all species are simultaneously co-evolving in a given physical environment.

A biological example presented by Holland illustrates the point (8). A given species of plant may be faced with an environment containing insects that like to eat it. To defend against its predators

(and increase its probability of survival in the environment), the plant may, over a period of time, evolve a tough exterior that makes it difficult for the insect to eat it. But, over a period of time, the insect may retaliate by evolving a stronger jaw so that the insect population can continue to feed on the plant (and increase its probability of survival in the environment). Then, over an additional period of time, the plant may evolve a poison to help defend itself further against the insects. But, then again, over a period of time, the insect may evolve a digestive enzyme that negates the effect of the poison so that the insect population can continue to feed on the plant.

In effect, both the plant and the insects get better and better at their respective defensive and offensive roles in this "biological arms race". Each species changes in response to the actions of the other.

BACKGROUND ON CO-EVOLUTION AND GENETIC ALGORITHMS

In the "genetic algorithm," described by John Holland in his pioneering *Adaptation in Natural and Artificial Systems* (6), a population of individuals attempts to adapt to a fixed "environment." In the basic genetic algorithm as described by Holland in 1975, the individuals in the population are fixed-length character strings (typically binary strings) that are encoded to represent some problem in some way. In the basic "genetic algorithm", the performance of the individuals in the population is measured using a fitness measure which is, in effect, the "environment" for the population. Over a period of many generations, the genetic algorithm causes the individuals in the population to adapt in a direction that is dictated by the fitness measure (its environment).

Holland (8) has incorporated co-evolution and genetic algorithms in his ECHO system for exploring the co-evolution of artificial organisms described by fixed-length character strings (chromosomes) in a "miniature world." In ECHO, there is a single population of artificial organisms. The environment of each organism includes all other organisms.

Miller (12,13) has used co-evolution in a genetic algorithm to evolve a finite automaton as the

strategy for playing the Repeated Prisoner's Dilemma game. Miller's population consisted of strings (chromosomes) of 148 binary digits to represent finite automata with 16 states. Each string in the population represented a complete strategy by which to play the game. That is, it specified what move the player was to make for any sequence of moves by the other player. Miller then used co-evolution to evolve strategies. Miller's co-evolutionary approach to the repeated prisoner's dilemma using genetic algorithms. contrasts with Alexrod's (1) evolutionary approach using genetic algorithms. Axelrod measured performance of a particular strategy by playing it against eight selected superior computer programs submitted in an international programming tournament for the prisoner's dilemma. A best strategy for one player (represented as a 70 bit string with a 3-move look-back) was then evolved with a weighted mix of eight opposing computer programs serving as the environment.

Hillis (5) used co-evolution in genetic algorithms to solve optimization problems.

John Maynard Smith (15) discussed co-evolution in connection with discovering strategies for game.

CO-EVOLUTION AND THE GENETIC PROGRAMMING PARADIGM

In the "hierarchical co-evolution algorithm," there are two (or more) populations of individuals. The environment for the first population consists of the second population. And, conversely, the environment for the second population consists of the first population.

The co-evolutionary process typically starts with both populations being highly unfit (when measured by an absolute fitness measure). Then, the first population tries to adapt to the "environment" created by the second population. Simultaneously, the second population tries to adapt to the "environment" created by the first population.

This process is carried out by testing the performance of each individual in the first population against each individual (or a sampling of individuals) from the second population. We call this performance the "relative fitness" of an individual because it represents the performance of one

individual in one population relative to the environment consisting of the entire second population. Then, each individual in the second population is tested against each individual (or a sampling of individuals) from the first population.

Note that this measurement of relative fitness for an individual in co-evolution is not an absolute measure of fitness against an optimal opponent, but merely a relative measure when the individual is tested against the current opposing population. If one population contains boxers who only throw left punches, then an individual whose defensive repertoire contains only defenses against left punches will have high relative fitness. But, this individual will have only mediocre absolute fitness when tested against an opponent who knows how to throw both left punches and right punches (i.e. an optimal opponent).

Even when both initial populations are initially highly unfit (both relatively and absolutely), the virtually inevitable variation of the initial random population will mean that some individuals have slightly better relative fitness than others. That means that some individuals in each population have somewhat better performance than others in dealing with the current opposing population.

The operation of fitness proportionate reproduction (based on the Darwinian principle of survival and reproduction of the fittest) can then be applied to each population using the relative fitness of each individual currently in each population. In addition, the operation of genetic recombination (crossover) can also be applied to a pair of parents, at least one of which is selected based on its relative fitness.

Over a period of time, both populations of individuals will tend to "co-evolve" and to rise to higher levels of performance as measured in terms of absolute fitness. Both populations do this without the aid of any externally supplied absolute fitness measure serving as the environment. In the limiting case, both populations of individuals can evolve to a level of performance that equals the absolute optimal fitness. Thus, the hierarchical co-evolution algorithm is a self-organizing, mutually-bootstrapping process that is driven only by relative fitness (and not by

absolute fitness).

Co-evolution is especially important in problems from game theory because one almost never has *a priori* access to a minimax strategy for either player. One therefore encounters a "chicken and egg" situation. In trying to develop a minimax strategy for the first player, one does not have the advantage of having a minimax second player against which to test candidate strategies. In checkers or chess, for example, it is difficult for a new player to learn to play well if he does not have the advantage of playing against a highly competent player.

CO-EVOLUTION OF A GAME STRATEGY

We now illustrate the "hierarchical co-evolution algorithm" to discover minimax strategies for both players simultaneously in a simple discrete two-person game represented by a game tree in extensive form.

In the hierarchical co-evolution algorithm, we do not have access to the optimal opponent to train the population. Instead, our objective is to breed two populations simultaneously. Both populations start as random compositions of the available functions and arguments.

Consider the following simple discrete game whose game tree is presented in extensive form in

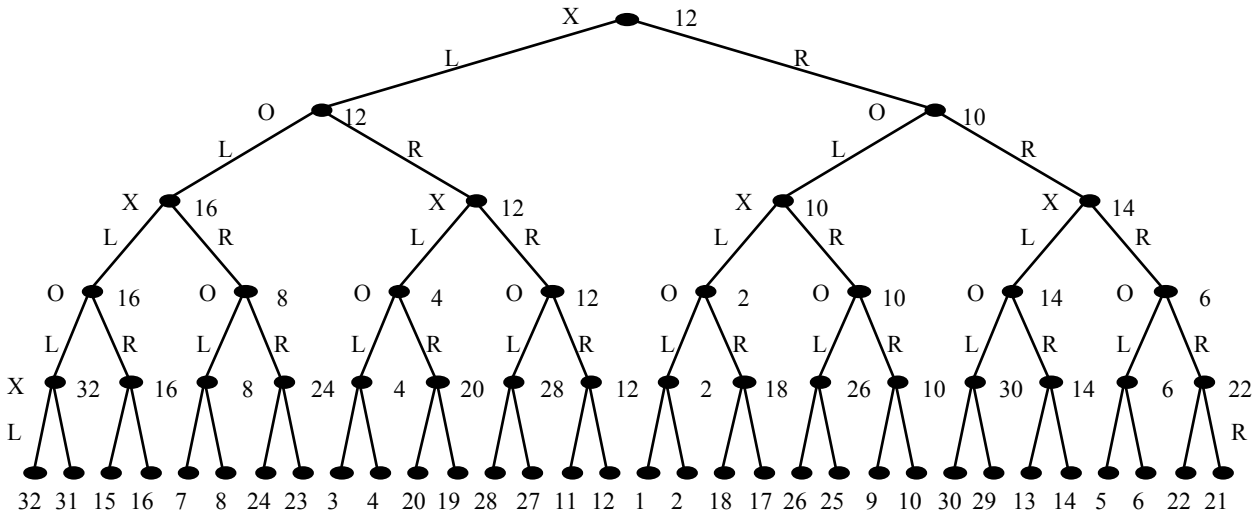


Figure 2 Game Tree with Payoffs

Figure 2. Each internal point of this tree is labeled with the player who must move. Each line is labeled with the choice (either L or R) made by the moving player. Each endpoint of the tree is labeled with the payoff (to player X).

This game is a two-person, competitive, zero-sum game in which the players make alternating moves. On each move, a player can choose to go L (left) or R (right). After player X has made three moves and player O has made two moves, player X receives (and player O pays out) the payoff shown at the particular endpoint of the game tree (1 of 32).

Each player has access to complete information about his opponent's previous moves (and his own previous moves). This historical information is contained in five variables XM1 (X's move 1), OM1 (O's move 1), XM2 (X's move 2), OM2 (O's move 2), and XM3 (X's move 3). These five variables each assume one of three possible values: L (left), R (right), or U (undefined). A variable is undefined prior to the time when the move to which it refers has been made. Thus, at the beginning of the game, all five variables are undefined. The particular variables that are defined and undefined indicate the point to which play has progressed during the play of the game. For example, if both players have moved once, XM1 and OM1 are defined (as either L or R) but the other three variables (XM2, OM2, and XM3) are undefined (have the value U).

A strategy for a particular player in a game specifies which choice that player is to make for every possible situation that may arise for that player. In particular, in this game, a strategy for player X must specify his first move if he happens to be at the beginning of the game. A strategy for player X must also specify his second move if player O has already made one move and it must specify his third move if player O has already made two moves. Since Player X moves first, player X's first move is not conditioned on any previous move. But, player X's second move will depend on Player O's first move (i.e. OM1) and, in general, it will also depend on his own first move (XM1). Similarly, player X's third move will depend on player O's first two moves and, in general, his own first two moves. Similarly, a strategy for player O must specify what choice player O is to make for every possible situation that may arise for player O. A strategy here is a

computer program (i.e. S-expression) whose inputs are the relevant historical variables and whose output is a move (L or R) for the player involved. Thus, the set of terminals is T = {L, R}.

Four testing functions CXM1, COM1, CXM2, and COM2 provide the facility to test each of the historical variables that are relevant to deciding upon a player's move. Each of these functions is a specialized form of the CASE function in LISP. For example, function CXM1 has three arguments and evaluates it first argument if XM1 (X's move 1) is undefined, evaluates its second argument if XM1 is L (Left), and evaluates its third argument if XM1 is R (Right). Functions CXM2, COM1, and COM2 are similarly defined. Thus, the function set for this problem is F = {CXM1, COM1, CXM2, COM2}. Each of these functions takes three arguments.

Our goal is to simultaneously co-evolve strategies for both players of this game.

In co-evolution, the relative fitness of a particular strategy for a particular player in a game is the average of the payoffs received when that strategy is played against the entire population of opposing strategies.

The absolute fitness of a particular strategy for a particular player in a game is the payoff received when that strategy is played against the minimax strategy for the opponent. Note that when we compute the absolute fitness of an X strategy for our descriptive purposes here, we test the X strategy against 4 possible combinations of O moves — that is, O's choice of L or R for his moves 1 and 2. When we compute the absolute fitness of an O strategy, we test it against 8 possible combinations of X moves — that is, X's choice of L or R for his moves 1, 2, and 3. Note that this testing of 4 or 8 combinations does not occur in the computation for relative fitness. When the two minimax strategies are played against each other, the payoff is 12. This score is known as the value of this game. A minimax strategy takes advantage of non-minimax play by the other player.

As previously mentioned, the co-evolution algorithm does not use the minimax strategy of the opponent in any way. We use it in this paper for descriptive purposes only. The co-evolution algorithm uses only relative fitness.

In one run (with population size of 300), the individual strategy for player X in the initial random generation (generation 0) with the best relative fitness was

```
(COM1 L (COM2 (CXM1 (CXM2 R (CXM2 R R R) (CXM2 R L R)) L (CXM2 L R (COM2
R R R))) (COM1 R (COM2 (CXM2 L R L) (COM2 R L L) R) (COM2 (COM1 R R L)
(CXM1 R L R) (CXM1 R L L))) (CXM1 (COM2 (CXM1 R L L) (CXM2 R R L) R) R
(COM2 L R (CXM1 L L L)))) R).
```

This simplifies to

```
(COM1 L (COM2 L L R) R).
```

This individual has relative fitness of 10.08.

The individual in the initial random population (generation 0) for player O with the best relative fitness was an equally complex expression. It simplifies to

```
(CXM2 R (CXM1 # L R) (CXM1 # R L)).
```

Note that, in simplifying this strategy, we inserted the symbol # to indicate that the situation involved can never arise. This individual has relative fitness of 7.57.

Neither the best X individual nor the best O individual from generation 0 reached maximal absolute fitness.

Note that the values of relative fitness for the relative best X individual and the relative best O individual from generation 0 (i.e. 10.08 and the 7.57) are each computed by averaging the payoff from the interaction of the individual involved with all 300 individual strategies in the current opposing population.

In generation 1, the individual strategy for player X with the best relative fitness had relative fitness of 11.28. This individual X strategy is still not a minimax strategy. It does not have the maximal absolute fitness.

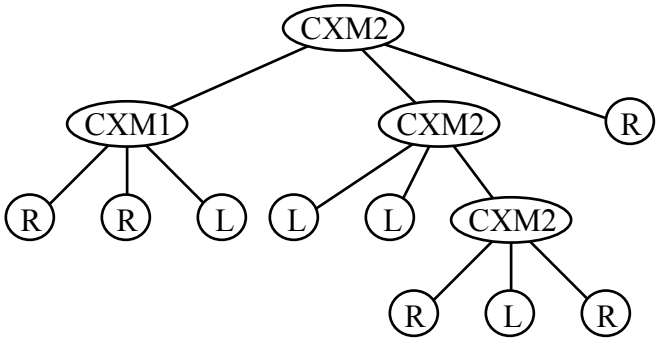
In generation 1, the best individual O strategy attained relative fitness of 7.18. It is shown below:

```
(CXM2 (CXM1 R R L) (CXM2 L L (CXM2 R L R)) R).
```

Although the co-evolution algorithm does not know it, this best single individual O strategy for generation 1 is, in fact, a minimax strategy for player O. It has maximal absolute fitness in this

game. This one O individual was the first such O individual to attain this level of performance during this run. If it were played against the minimax X strategy, it would score 12 (i.e. the value of this game).

This individual O strategy can be graphically depicted as shown below:



This individual O strategy simplifies to

```
(CXM2 (CXM1 # R L) L R).
```

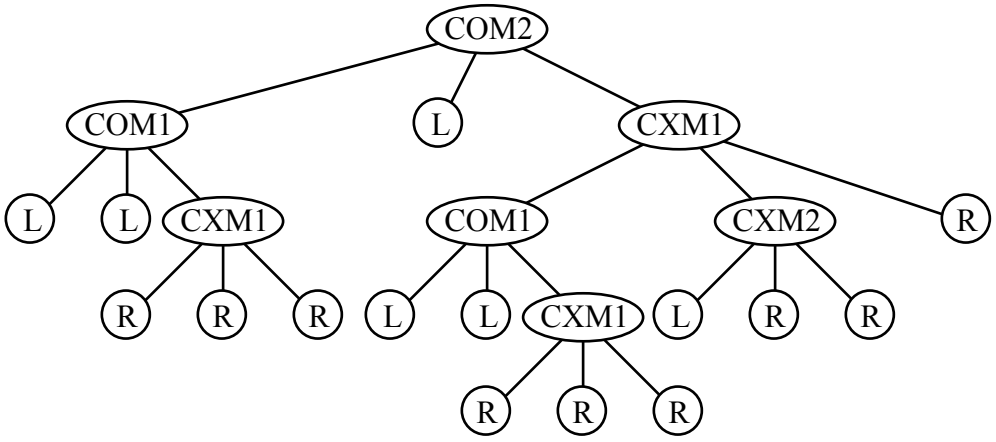
Between generations 2 and 14, the number of individuals in the O population reaching maximal absolute fitness was 2, 7, 17, 28, 35, 40, 50, 64, 73, 83, 93, 98, and 107, respectively. That is, programs equivalent to the minimax O strategy began to dominate the O population.

In generation 14, the individual strategy for player X with the best relative fitness had relative fitness of 18.11. This individual X strategy was

```
(COM2 (COM1 L L (CXM1 R R R)) L (CXM1 (COM1 L L (CXM1 R R R))
      (CXM2 L R R) R)).
```

Although the co-evolution algorithm does not know it, this best single individual X strategy is, in fact, a minimax strategy for player X. This individual X strategy was the first such X individual to attain this level of performance during this run. If it were played against the minimax O strategy, it would score 12 (i.e. the value of this game).

This individual X strategy can be graphically depicted as shown below:



This individual X strategy simplifies to

```
(COM2 (COM1 L L R) L R).
```

Between generations 15 and 29, the number of individuals in the X population reaching maximal absolute fitness was 3, 4, 8, 11, 10, 9, 13, 21, 24, 29, 43, 32, 52, 48, and 50, respectively. That is, programs equivalent to the minimax X strategy began to dominate the X population. Meanwhile, the O population became even more dominated by programs equivalent to the O minimax strategy.

By generation 38, the number of O individuals in the population reaching maximal absolute fitness reached 188 (almost two thirds of the population) and the number of X individuals reaching maximal absolute fitness reached 74 (about a quarter). That is, by generation 38, the minimax strategies for both players were becoming dominant.

Interestingly, these 74 individual X strategies had relative fitness of 19.11 and these 188 individual O strategies had relative fitness of 10.47. Neither of these values equals 12 because the other population is not fully converged to its minimax strategy.

In summary, we genetically bred the minimax strategies for both players of this game without using knowledge of the minimax strategy for either player.

CONCLUSION

In this paper, we have demonstrated the use of the newly developed genetic programming

paradigm to evolve hierarchical computer programs to solve three illustrative problems. These three illustrative problems are only a small subset of the benchmark problems already successfully solved by the genetic programming paradigm (10, 11). These three illustrative problems highlight some of the features of the genetic programming paradigm as compared to other existing paradigms for machine learning and artificial intelligence (such as neural networks and conventional string-based genetic algorithms) that may commend it for future work in the field of artificial life. These features include the following:

- In the genetic programming paradigm, the size and shape of the solution is not specified in advance, but, instead, evolves as the problem is being solved. For many problems, it is difficult, impossible, or unnatural to try to specify (or restrict) the size and shape of the eventual solution in advance. Moreover, advance specification (or restriction) of the size and shape of the solution to a problem narrows the window by which the system views the world and may well preclude finding the solution to the problem. The dynamic variability of the size and shape of the computer programs in the genetic programming paradigm is in marked contrast to both neural network paradigms and conventional string-based genetic algorithms.

- The genetic programming paradigm evolves solutions that are directly expressed in a natural programming structure that overtly contains the functions and arguments naturally arising from the problem domain itself. Solutions expressed in this way are immediately understandable in the terms of the problem domain. Another consequence of this is that the results are relatively easy to audit. This is in marked contrast to solutions produced by, for example, neural network paradigms.

- In the genetic programming paradigm, there is no preprocessing of inputs (in contrast to neural networks, conventional string-based genetic algorithms, and most other machine learning paradigms).

- The genetic programming paradigm works with hierarchical structures at each stage of the process. As a result, the solutions are always hierarchical. Hierarchical structures offer the

possibility of efficiently and understandably representing solutions to problems and also offer the possibility of scaling up well to larger, more significant problems.

In summary, we have shown, by the use the three illustrative problems here (and the wide variety of other seemingly different problems cited from other areas), the power and flexibility of the genetic programming paradigm.

REFERENCES

1 Axelrod, R. "The evolution of strategies in the iterated prisoner's dilemma." In *Genetic Algorithms and Simulated Annealing* , edited by L. Davis. London: Pittman l987.

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

3 De Jong, Kenneth A. "Genetic algorithms: A l0 year perspective." In Grefenstette, J. J.(editor). *Proceedings of an International Conference on Genetic Algorithms and Their Applications.* Hillsdale, NJ: Lawrence Erlbaum Associates l985.

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

5 Hillis, W. Daniel. "Co-Evolving Parasites Improve Simulated Evolution as an Optimization Procedure." In *Emergent Computation: Self-organizing, Collective, and Cooperative Computing Networks*. edited by S. Forrest. Cambridge, MA: MIT Press 1990.

Also in Langton, Christopher G. and Farmer, J. Doyne. (editors) *Proceedings of the Second Conference on Artificial Life*. Redwood City, CA; Addison-Wesley 1990.

6 Holland, J. H. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press 1975.

7 Holland, John H. Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In Michalski, Ryszard S., Carbonell, Jaime G. and Mitchell, Tom M. Machine Learning: An Artificial Intelligence Approach,

Volume II. P. 593-623. Los Altos, CA: Morgan Kaufman l986.

8 Holland, J. H. "ECHO: Explorations of Evolution in a Minature World." In *Proceedings of the Second Conference on Artificial Life*. edited by C. G. Langton, and J. D. Farmer, J. Doyne. Redwood City, CA; Addison-Wesley 1990.

9 Jefferson, David, Collins, Rob, et. al. "The Genesys System: Evolution as a Theme in Artificial Life." In *Proceedings of Second Conference on Artificial Life*, edited by C. G. Langton and D Farmer. Redwood City, CA: Addison-Wesley. 1990.

10 Koza, John R. "Hierarchical Genetic Algorithms Operating on Populations of Computer Programs. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI)*. San Mateo, CA: Morgan Kaufman 1989.

11 Koza, John R. *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems.* Stanford University Computer Science Department Technical Report STAN-CS-90-1314. June 1990. 1990.

12 Miller, J. H. "The Co-evolution of Automata in the Repeated Prisoner's Dilemma." Sante Fe Institute Report 89-003. 1989.

13 Miller, J. H. "The Evolution of Automata in the Repeated Prisoner's Dilemma." In *Two Essays on the Economics of Imperfect Information*. PhD Dissertation, Department of Economics, University of Michigan, 1988.

14 Schaffer , J. D. (editor) *Proceedings of the Third International Conference on Genetic Algorithms*. San Mateo, Ca: Morgan Kaufmann Publishers Inc. 1989.

15 Smith, John Maynard. *Evolutionary Genetics.* Oxford: Oxford University Press. 1989.

16 Smith, Steven F. *A Learning System Based on Genetic Adaptive Algorithms*. PhD Dissertation. Pittsburgh: University of Pittsburgh 1980.