Genetically Breeding Populations of Computer Programs to Solve Problems in Artificial Intelligence

John R. Koza
Computer Science Department
Stanford University
Stanford, CA 94305 USA
Koza@Sunburn.Stanford.Edu
415-941-0336

Abstract

This paper describes the recently developed "genetic programming" paradigm which genetically breeds populations of computer programs to solve problems. In genetic programming, the individuals in the population are hierarchical computer programs of various sizes and shapes. Three applications to problems in artificial intelligence are presented.

Introduction and Overview

This paper describes the recently developed "genetic programming" paradigm which genetically breeds populations of computer programs to solve problems. In genetic programming, the individuals in the population are hierarchical compositions of functions and arguments of various sizes and shapes. Each of these individual computer programs is evaluated for its fitness in handling the problem environment and a simulated evolutionary process is driven by this measure of fitness.

In this paper, the genetic programming paradigm is illustrated with three problems. The first problem involves genetically breeding a population of computer programs to allow an "artificial ant" to traverse an irregular trail. The second problem involves genetically breeding a minimax control strategy in a differential game with an independently-acting pursuer and evader. The third problem involves genetically breeding a minimax strategy for a player of a simple discrete two-person 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 (1) natural genetic operations such as sexual recombination (crossover) and (2) 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 *Adaptation in Natural and Artificial Systems* (Holland 1975). 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. Recent work in genetic algorithms and genetic classifier systems can be surveyed in Goldberg (1989), Davis (1987), and Schaffer (1989).

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 in artificial intelligence 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 (De Jong 1985, De Jong 1987, De Jong 1988).

The structure of the individual mathematical objects that are manipulated by the genetic algorithm can be more complex than the fixed length character strings. Smith (1980) departed from the early fixed-length character strings by introducing variable length strings, including strings whose elements were if-then rules (rather than single characters). Holland's introduction of the classifier system (1986) continued the trend towards increasing the complexity of the structures undergoing adaptation. The classifier system is a cognitive architecture into which the genetic algorithm is embedded so as to allow adaptive modification of a population of string-based if-then rules (whose condition and action parts are fixed length binary strings).

In addition, we have recently shown that entire com-

puter programs can be genetically bred to solve problems in a variety of different areas of artificial intelligence, machine learning, and symbolic processing (Koza 1989, 1990a). In this recently developed "genetic programming" paradigm, the individuals in the population are compositions of functions and terminals appropriate to the particular problem domain. The set of functions used typically includes arithmetic operations, mathematical functions, conditional logical operations, and domain-specific functions. Each function in the function set must 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 that can be recursively composed of the available functions 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 proportionate reproduction and crossover (recombination). Fitness proportionate reproduction is the basic engine of Darwinian reproduction and survival of the fittest and operates for genetic programming paradigms in the same way as it does for conventional genetic algorithms. The crossover operation for genetic programming paradigms is a sexual operation that operates on two parental LISP S-expressions and produces two offspring S-expressions using parts of each parent. In particular, the crossover operation creates new offspring S-expressions by exchanging 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:

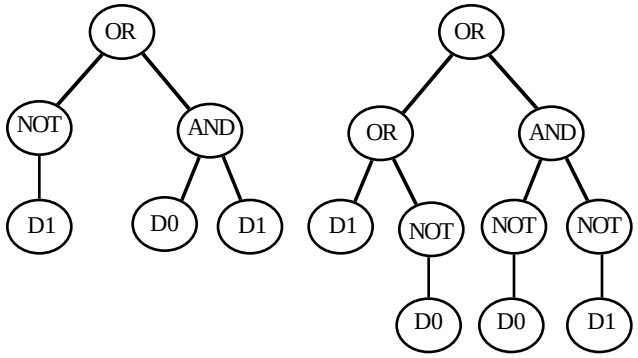
`(OR` **`(NOT D1)`** `(AND D0 D1))`

And, consider the second parental S-expression below:

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

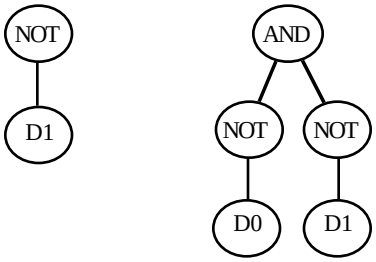
These two LISP S-expressions can be depicted graphically as rooted, point-labeled trees with ordered branches.

The two parental LISP S-expressions are shown below:

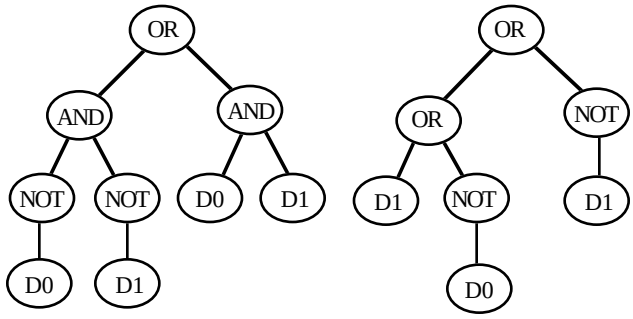


Assume that the points of both trees are numbered in a depth-first way starting at the left. Suppose that the second point (out of 6 points of the first parent) is randomly selected as the crossover point for the first parent and that the sixth point (out of 10 points of the second parent) is randomly 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 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))`.

Details can be found in Koza (1990a).

This new genetic algorithm paradigm has been successfully applied (Koza 1989, 1990a) to example problems in several different areas, including (1) machine learning of functions (e.g. learning the Boolean 11-multiplexer function, (2) planning (e.g. developing a robotic action sequence that can stack an arbitrary initial configuration of blocks into a specified order), (3) automatic programming (e.g. discovering a computational procedure for solving pairs of linear equations, solving quadratic equations for complex roots, and discovering trigonometric identities), (4) sequence induction (e.g. inducing a recursive computational procedure for the Fibonacci and the Hofstadter sequences), (5) pattern recognition (e.g. translation-invariant recognition of a simple one-dimensional shape in a linear retina), (6) 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) (Koza and Keane 1990a, Koza and Keane 1990b), (7) symbolic "data to function" regression, symbolic "data to function" integration, and symbolic "data to function" differentiation, (8) symbolic solution to functional equations (including differential equations with initial conditions, integral equations, and general functional equations), (9) empirical discovery (e.g. rediscovering Kepler's Third Law, rediscovering the well-known econometric "exchange equation" MV = PQ from actual time series data for the money supply, the velocity of money, the price level, and the gross national product of an economy) (Koza 1990b), and (10) simultaneous architectural design and training of neural networks.

Three Examples

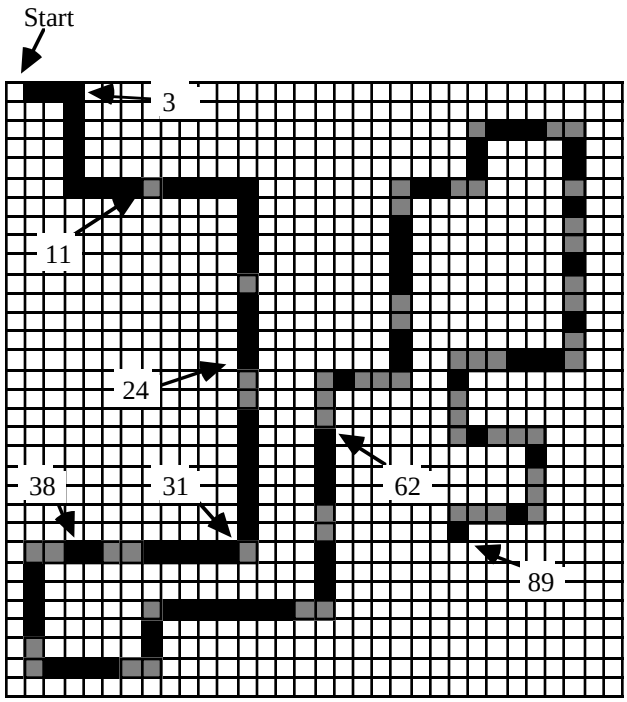
In this section, we present three illustrative examples of the genetic programming paradigm.

The "Artificial Ant"

Jefferson et. al. (1990) devised a planning task involving an “artificial ant” attempting to traverse an irregular trail and successfully used the conventional 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 of stones in 89 of the 1024 cells. The trail has single missing stones, double missing stones, single missing stones at some corners, double missing stones at other corners (knight moves), and triple missing stones at other corners (long knight moves).

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. 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 binary 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 (representing the three actions). 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 (64 substrings of length 7 representing the state transitions plus 5 additional bits representing the initial state of the automaton). They 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), Jefferson, Collins et. al. 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 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). 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.

The initial generation (generation 0) consisted of randomly generated individual S-expressions recursively created using the available functions and available terminals of the problem. For this problem (and each of the other problems described in this paper), 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. Mutation was not used. For practical reasons, 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.

In one run, an individual LISP S-expression scoring 89 out of 89 emerged on the 7th 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 undertake to measure the number of individuals that need to be processed by a genetic algorithm to produce a desired result 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, 8 independent runs are required to assure a 99% probability of solving the problem on at least one of the 8 runs. That is, it is sufficient to process 408,000 individuals. This requires about 6 hours of computing time on the Texas Instruments Explorer II+™ workstation.

The graph below shows that the probability of

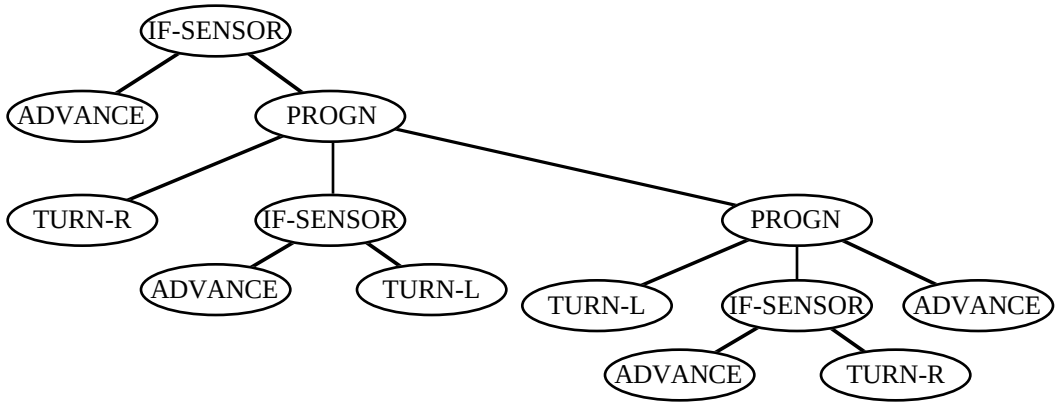
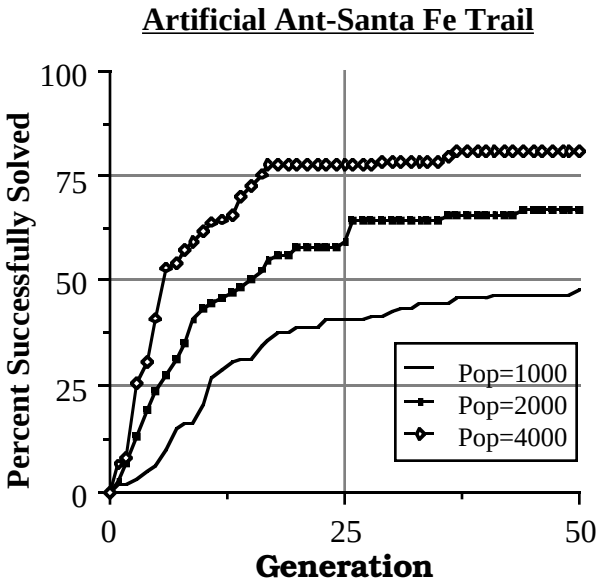


Figure. 1 Artificial Ant Solution

success p_S of a run is 67% for a population size of 2000 with 51 generations. The graph also shows that the probability of success of a run is 81% for a population of 4000 with 51 generations. A population of 2000 requires 4 independent runs (i.e. 408,000 individuals are sufficient) to achieve the desired 99% probability. A population of 4000 requires 3 independent runs (i.e. 612,000 individuals are sufficient).



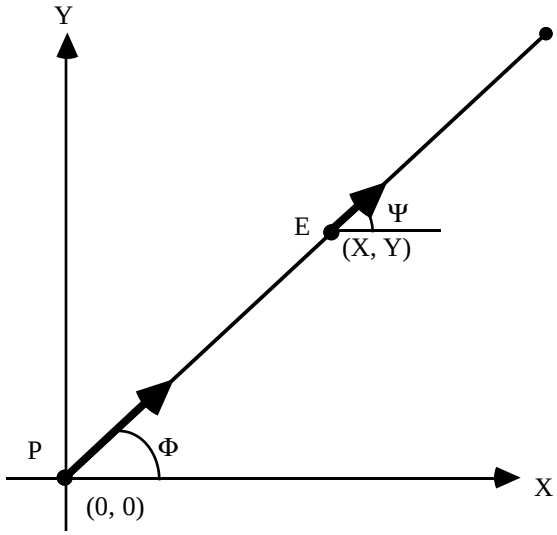
Differential Pursuit Game

As a second illustration of the genetic programming paradigm, we consider a differential pursuer-evader game. Our objective is to find an optimal strategy for one player when the environment (fitness function) consists of an optimal opponent.

In a game, there are two or more independently-acting players who make choices (moves) and receive a payoff based on the choices they make. The differential "game of simple pursuit" is a two-person, competitive, zero-sum, simultaneous-moving, complete-information game in which a fast pursuing player P is trying to capture a slower evading player E. The "choices" available to a player at a given moment consist of choosing a direction (angle) in which to travel. In the simple game of pursuit, the players travel in a plane and both players may instantaneously change direction. Each player travels at a constant speed, and the pursuing player's speed w_p (1.0) is greater than the evading player's speed w_e (0.67).

The state variables of the game are x_p, y_p, x_e, and y_e representing the coordinate positions (x_p, y_p) and (x_e, y_e) of the pursuer P and evader E in the plane.

At each time step, both players know the position (state variables) of both players. The choice for each player is to select a value of their control variable (i.e. the angular direction in which to travel). The pursuer's control variable is the angle ϕ (from 0 to 2π radians) and the evader's control variable is the angle ψ . The players choose their respective control variable simultaneously.



The analysis of this game can be simplified by reducing the number of state variables from four to two (Isaacs 1965). This state reduction is accomplished by simply viewing the pursuer P as being at the origin point (0,0) of a new coordinate system at all times and then viewing the evader E as being at position (x, y) in this new coordinate system. The two numbers x and y representing the position (x, y) of the evader E thus become the two "reduced" state variables of the game. Whenever the pursuer P travels in a particular direction, the coordinate system is immediately adjusted so that the pursuer is repositioned back to the origin (0, 0) and then appropriately adjusting the position (x, y) of the evader to reflect the travel of the pursuer.

The state transition equations for the evader E follow:

$$x(t+1) = x(t) + w_e \text{Cos } \psi - w_p \text{Cos } \phi$$

$$y(t+1) = y(t) + w_e \text{Sin } \psi - w_p \text{Cos } \phi.$$

In order to develop optimal playing strategies, we use a set of random environmental starting condition cases consisting of N_e (= 10) starting positions (x_i, y_i) for the evader E. Each starting value of x_i and y_i is between -5.0 and +5.0. The two players may travel anywhere in the plane. We regard the pursuer P as having captured the evader E when the pursuer gets to within some specified small distance ε= 0.5 of the evader E.

The payoff for a given player is measured by time. The payoff for the pursuer P is the total time it takes to capture the evader E over all of the environmental cases. The pursuer tries to minimize the time to capture. The payoff for the evader E is the total time of survival for E.

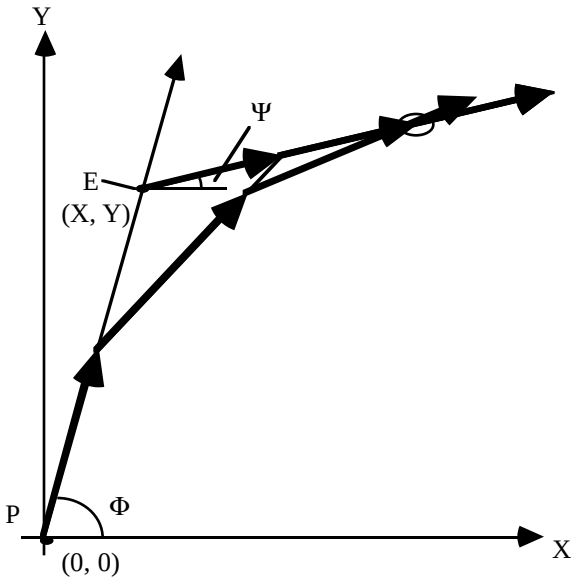
The evader tries to maximize this time of survival.

A maximum "time out" time (100 time steps) is established so that if a particular pursuer strategy has not made the capture within that amount of time, that maximum time becomes the payoff for that particular environmental case and that particular strategy.

The problem is to find the strategy for choosing the control variable of the pursuer so as to minimize the total time to capture for any set of environmental cases when playing against an optimal evader.

For this particular simple game, the best strategy for the pursuer P at any given time step is to chase the evader E in the direction of the straight line currently connecting the pursuer to the evader. And, for this particular simple game, the best strategy for the evader E is to race away from the pursuer in the direction of the straight line connecting the pursuer to the evader.

If the evader chooses some action other than the strategy of racing away from the pursuer in the direction of the straight line connecting the pursuer to the evader (as shown below), the evader will survive for less time than if he follows his best strategy. If the evader initially chooses a sub-optimal direction and then belatedly chooses the optimal direction, his time of survival is less than if he had chosen the optimal direction from the beginning.



The situation is symmetric in that if the pursuer does not chase after the evader E along this same straight line, he fails to minimize the time to capture.

The "value of the game" is the payoff (time) such that, no matter what the evader does, the evader cannot hold out for longer than this amount of time, and, if the evader does anything other than direct fleeing, his survival time is a shorter amount of time. Conversely, no matter what the pursuer does, the pursuer P cannot capture an optimal evader E in less than that amount of time. And, if the pursuer does anything other than direct pursuit, the evader can remain at large for a longer amount of time.

The genetic programming paradigm can be used to solve the differential game of simple pursuit by genetically evolving a population of strategies for the pursuing individuals over a number of generations.

The genetic programming paradigm is especially well suited to solving this kind of problem because the solution takes the form of a mathematical expression whose size and shape may not be known in advance.

The terminal set is T = {X, Y, R}. The two state variables X and Y represent the position of the evader E in the plane in a "reduced" coordinate system where the pursuer is always positioned (or repositioned) to the origin. The terminal R is an ephemeral random constant. Each occurrence of an ephemeral random constant in an individual in the initial random population is assigned a different random floating point value (between -1.0 and +1.0). These ephemeral random constants give the algorithm the building blocks with which to construct the particular numeric constants that may be needed in the solution. Additional details are found in Koza (1990a).

The function set for this problem can be a set of arithmetic and mathematical operations such as addition, subtraction, multiplication, division (using the operation % which returns a zero when division by zero is attempted), and the exponential function EXP. Thus, the function set is F = {+, -, *, %, EXP}.

If the population of individuals represents pursuers and we are attempting to genetically breed an optimal pursuing individual, the environment for this "genetic algorithm" consists of an optimal evading individual. The optimal evader travels with the established constant evader speed w_e in the angular direction specified by the Arctangent function.

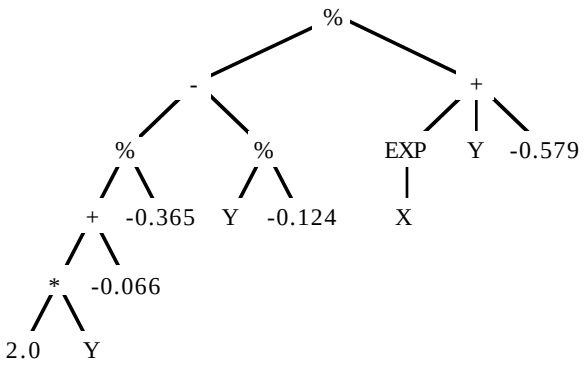
The genetic programming paradigm is successful in genetically breeding optimal pursuing individuals. As one progresses from generation to generation, the population of pursuing individuals typically improves. After several generations, the best pursuing individuals in the population can capture the evader in a small fraction (perhaps 2, 3, or 4) of the 10 environmental cases within a certain amount of time. Then, after additional generations, the population improves and the best pursuing individuals in the population can capture the evader in a larger fraction (perhaps 4, 5, or 6) of the 10 environmental cases within a shorter amount of time. Often, these partially effective pursuers are effective in some identifiable fraction of the plane or at some identifiable range of distances, but ineffective in other parts of the plane or at other distances. However, as more and more generations are run, the population of pursuing individuals typically continues to improve.

In one run, a pursuer strategy emerged in the 17th generation which correctly handled all 10 of the environmental cases. This S-expression is shown below.

```
(% (- (% (+ (* 2.0 Y) -0.066) -0.365)
      (% Y -0.124))
   (+ (EXP X) Y -0.579))
```

When this apparently optimal pursuing individual is re-tested against a much larger set of environmental cases (i.e. 1000), we then find that it also successfully handles 100% of the environmental cases. Thus, this S-expression is an optimal solution to the problem. It is also, as a result, an excellent approximation to the Arctangent function.

This S-expression is depicted graphically below:



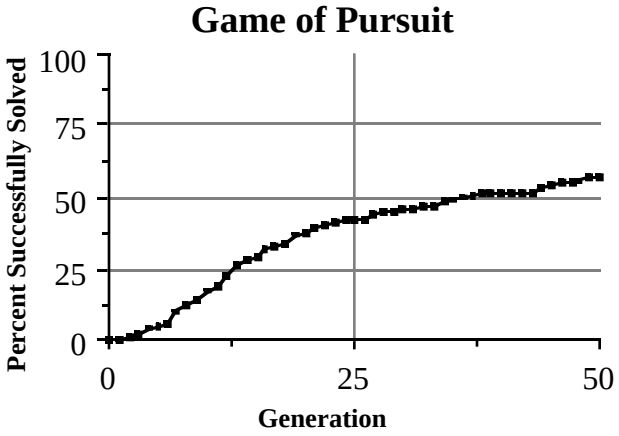
This S-expression is equivalent to

$$\frac{\frac{2y - 0.066}{-0.365} + \frac{y}{0.124}}{e^x + y - 0.579}$$

which in turn is equivalent to

$$\frac{2.58y + 1.81}{e^x + y - 0.579}$$

For example, we ran 111 runs of the differential pursuer-evader game problem with a population size of 500 and found that the probability of success p_S was 55% after 51 generations (see graph below). With a probability of success p_S of 55%, 6 independent runs are required to assure a 99% probability of solving the problem. That is, 153,000 individuals must be processed.



An optimal evader has been similarly evolved using an optimal pursuer (i.e. the Arctangent strategy).

Minimax Strategy for a Game

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

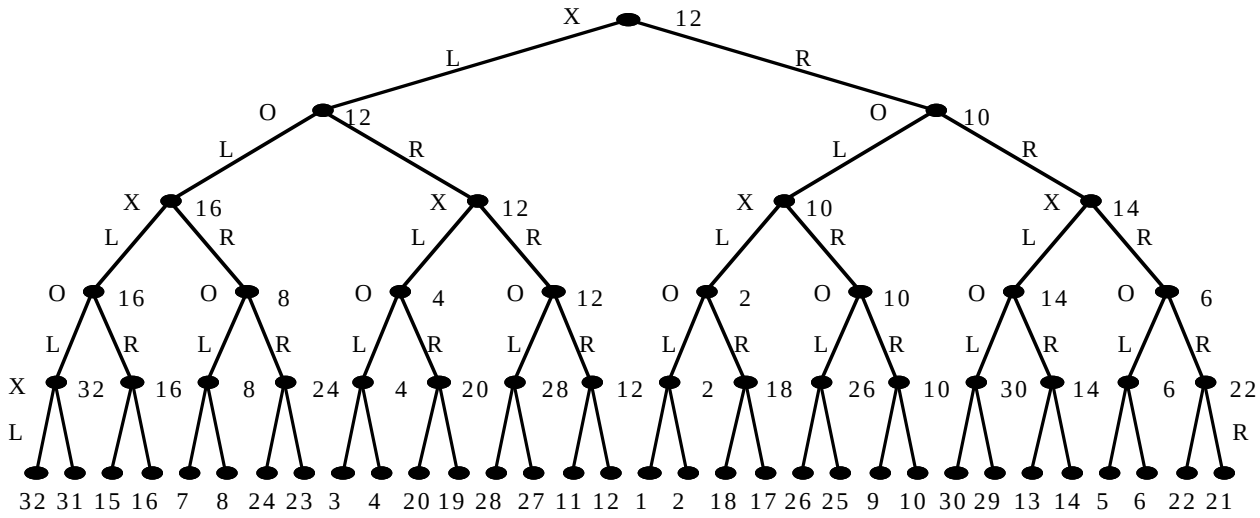


Figure 2 Game Tree with Payoffs

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).

Since this game is a game of complete information, 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 is 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 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.

The 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 fitness of an X strategy, 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 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. When two minimax strategies are played against each other, the payoff is 12, which is the value of this game. A minimax strategy takes advantage of non-minimax play by the other player. A "minimax hit" is a measure of success equal to the number of combinations (out of 4 or 8) where the strategy being tested achieves a payoff at least as high as the minimax strategy.

In one run, the best single individual game-playing strategy for player X in generation 6 had a minimax value of 88 and scored 4 minimax hits. This strategy was

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

This strategy simplifies to

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

If both OM2 (O's move 2) and OM1 (O's move 1) are undefined (U), it must be player X's first move. That is, we are at the beginning of the game (i.e. the root of the game tree). In this situation, the first argument of the COM1 function embedded inside the COM2 function of this strategy specifies that player X is to move L. The left move by player X at the beginning of the game is player X's minimax move because it takes the game to a point with minimax value 12 (to player X) as opposed to a point with only minimax value 10.

If OM2 (O's move 2) is undefined but OM1 is defined, it must be player X's second move. In this situation, this strategy specifies that player X moves L if OM1 (O's move 1) was L and player X moves R if OM1 was R. If OM1 (O's move 1) was L, player O has moved to a point with minimax value 16. Player X should then move L (rather than R) because that move will take the game to a point with minimax value 16 (rather than 8). If OM1 was R, player O has moved to a point with minimax value 12. This move is better for O than moving L. Player X should then move R (rather than L) because that move will take the game to a point with minimax value 12 (rather than 4).

If both OM1 and OM2 are defined, it must be player X's third move. If OM2 was L, player X can either choose between a payoff of 32 or 31 or between a payoff of 28 or 27. In either case, player X moves L. If OM2 was R, player X can choose between a payoff of 15 or 16 or between a payoff of 11 or 12. In either case, player X moves R. In this situation, this S-expression specifies that player X moves L if OM2 (O's move 2) was L and player X moves R if OM2 was R.

If player O has been playing its minimax strategy, this S-expression will cause the game to finish at the endpoint with the payoff of 12 to player X. However, if player O was not playing his minimax strategy, this S-expression will cause the game to finish with a payoff of 32, 16, or 28 for player X. The total of the 12, 32, 16, and 28 is 88 and the attainment of these four values constitutes 4 minimax hits.

We then proceeded to evolve a game-playing strategy for player O for this game. The minimax strategy for player X serves as the environment for evolving game-playing strategies for player O.

In one run, the best single individual strategy for player O in generation 9 had a minimax value of 52 and scored 8 minimax hits. This minimax O strategy was

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

This minimax O strategy simplifies to

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

Note that, in simplifying this strategy, we inserted the symbol # to indicate the situation that can never arise.

Conclusions

We used the genetic programming paradigm to discover a plan allowing an "artificial ant" to traverse an irregular trail, to discover the minimax strategy for a pursuer in the differential game of simple pursuit, and to find the minimax game-playing strategy for a discrete game in extensive form.

Acknowledgments

James P. Rice of the Knowledge Systems Laboratory at Stanford University made numerous contributions in connection with the computer programming of the above.

References

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

De Jong, Kenneth A. Genetic algorithms: A l0 year perspective. *Proceedings of an International Conference on Genetic Algorithms and Their Applications.* Hillsdale, NJ: Lawrence Erlbaum Associates l985.

De Jong, Kenneth A. On using genetic algorithms to search program spaces. *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms.* Hillsdale, NJ: Lawrence Erlbaum Associates l987.

De Jong, Kenneth A. Learning with genetic algorithms: an overview. *Machine Learning*, 3(2), 121-138, 1988.

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

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

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.

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. In press.

Isaacs, Rufus.. *Differential Games*. New York: John Wiley 1965.

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. In Press.

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: Morgan Kaufman 1989.

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. 1990a.

Koza, John R. "A Genetic Approach to Econometric Modeling." Sixth World Congress of the Econometric Society. Barcelona, Spain. August 27, 1990. 1990b.

Koza, John R. and Keane, Martin A. "Cart Centering and Broom Balancing by Genetically Breeding Populations of Control Strategy Programs." In *Proceedings of International Joint Conference on Neural Networks, Washington, January 15-19, 1990*. Volume I. Hillsdale, NJ: Lawrence Erlbaum 1990.

Koza, John R. and Keane, Martin A. "Genetic Breeding of Non-Linear Optimal Control Strategies for Broom Balancing." In *Proceedings of the Ninth International Conference on Analysis and Optimization of Systems. Antibes, June, 1990.* Berlin: Springer-Verlag, 1990.

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

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