Automatic Programming of Robots using Genetic Programming

John R. Koza
Stanford University Computer Science Department
Stanford, CA 94305 USA
E-MAIL: Koza@Sunburn.Stanford.Edu
PHONE: 415-941-0336 FAX: 415-941-9430

James P. Rice
Stanford University Knowledge Systems Laboratory
701 Welch Road
Palo Alto, CA 94304 USA
Rice@Sumex-Aim.Stanford.Edu 415-723-8405

Abstract

The goal in automatic programming is to get a computer to perform a task by telling it what needs to be done, rather than by explicitly programming it. This paper considers the task of automatically generating a computer program to enable an autonomous mobile robot to perform the task of moving a box from the middle of an irregular shaped room to the wall. We compare the ability of the recently developed genetic programming paradigm to produce such a program to the reported ability of reinforcement learning techniques, such as Q learning, to produce such a program in the style of the subsumption architecture. The computational requirements of reinforcement learning necessitates considerable human knowledge and intervention, whereas genetic programming comes much closer to achieving the goal of getting the computer to perform the task without explicitly programming it. The solution produced by genetic programming emerges as a result of Darwinian natural selection and genetic crossover (sexual recombination) in a population of computer programs. The process is driven by a fitness measure which communicates the nature of the task to the computer and its learning paradigm.

Introduction and Overview

In the 1950s, Arthur Samuel identified the goal of getting a computer to perform a task without being explicitly programmed as one of the central goals in the fields of computer science and artificial intelligence. Such automatic programming of a computer involves merely telling the computer what is to be done, rather than explicitly telling it, step-by-step, how to perform the desired task.

In an AAAI-91 paper entitled "Automatic Programming of Behavior-Based Robots using Reinforcement Learning" Mahadevan and Connell (1991) reported on using reinforcement learning techniques, such as Q learning (Watkins 1989), in producing a program to control an autonomous mobile robot in the style of the subsumption architecture (Brooks 1986, Connell 1990, Mataric 1990). In particular, the program produced by reinforcement learning techniques enabled an autonomous mobile robot to perform the task of moving a box from the middle of a room to the wall. In this paper, we will show that the onerous computational requirements imposed by reinforcement learning techniques necessitated that a considerable amount of human knowledge be supplied in order to achieve the reported "automatic programming" of the box moving task.

In this paper, we present an alternative method for automatically generating a computer program to perform the box moving task using the recently developed genetic programming paradigm. In genetic programming, populations of computer programs are genetically bred in order to solve the problem. The solution produced by genetic programming emerges as a result of Darwinian natural selection and genetic crossover (sexual recombination) in a population of computer programs. The process is driven by a fitness measure which communicates the nature of the task to the computer and its learning paradigm.

We demonstrate that genetic programming comes much closer than reinforcement learning techniques to achieving the goal of getting the computer to perform the task without explicitly programming it.

Background on Genetic Algorithms

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

Recent work in genetic algorithms can be surveyed in Goldberg 1989 and Davis (1987, 1991).

Background on Genetic Programming

For many problems, the most natural representation for solutions are computer programs. The size, shape, and contents of the computer program to solve the problem is generally not known in advance. The computer program that solves a given problem is typically a hierarchical composition of various terminals and primitive function appropriate to the problem domain. In robotics problems, the computer program that solves a given problem typically takes the sensor inputs or state variables of the system as its input and produces an external action as its output. It is unnatural and difficult to represent program hierarchies of unknown (and possibly dynamically varying) size and shape with the fixed length character strings generally used in the conventional genetic algorithm.

In genetic programming, one recasts the search for the solution to the problem at hand as a search of the space of all possible compositions of functions and terminals (i.e. computer programs) that can be recursively composed from the available functions and terminals. Depending on the particular problem at hand, the set of functions used may include arithmetic operations, primitive robotic actions, conditional branching operations, mathematical functions, or domain-specific functions. For robotic problems, the primitive functions are typically the external actions which the robot can execute; the terminals are typically the sensor inputs or state variables of the system.

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.

Genetic programming, like the conventional genetic algorithm, is a domain independent method. It proceeds by genetically breeding populations of computer programs to solve problems by executing the following three steps:

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

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

  (a) Execute each program in the population and assign it a fitness value according to how well it solves the problem.

  (b) Create a new population of computer programs by applying the following two primary operations. The operations are applied to computer program(s) in the population chosen with a probability based on fitness.

    (i) *Reproduction*: Copy existing computer programs to the new population.

    (ii) *Crossover*: Create new computer programs by genetically recombining randomly chosen parts of two existing programs.

(3) The single best computer program in the population at the time of termination is designated as the result of the run of genetic programming. This result may be a solution (or approximate solution) to the problem.

The basic genetic operations for genetic programming are fitness proportionate reproduction and crossover (recombination).

The reproduction operation copies individuals in the genetic population into the next generation of the population in proportion to their fitness in grappling with the problem environment. This operation is the basic engine of Darwinian survival and reproduction of the fittest.

The crossover operation is a sexual operation that operates on two parental LISP S-expressions and produces two offspring S-expressions using parts of each parent. 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 crossover operation always produces syntactically and semantically valid LISP S-expressions as offspring regardless of the crossover points.

For example, consider the two parental S-expressions:

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

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

Figure 1 graphically depicts these two S-expressions as rooted, point-labeled trees with ordered branches. The numbers on the tree are for reference only.

Assume that the points of both trees are numbered in a depth-first way starting at the left. Suppose that point 2 (out of 6 points of the first parent) is randomly selected as the crossover point for the first parent and that point 6 (out of 10 points of the second parent) is randomly selected as the crossover point of the second parent. The crossover points in the trees above are therefore the NOT in the first parent and the AND in the second parent.

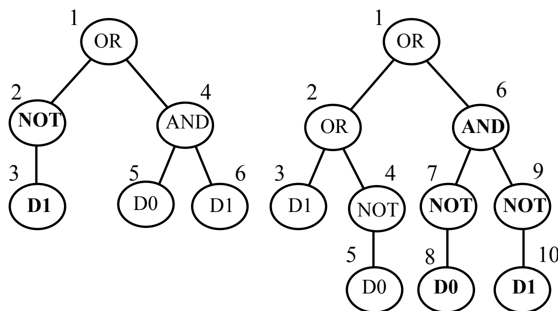

Figure 1 Two parental computer programs shown as trees with ordered branches. Internal points of the tree correspond to functions (i.e. operations) and external points correspond to terminals (i.e. input data).

Figure 2 shows the two crossover fragments are two sub-trees. These two crossover fragments correspond to the bold sub-expressions (sub-lists) in the two parental LISP S-expressions shown above. Figure 3 shows the two offspring.

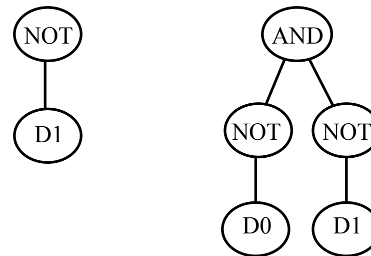

Figure 2 The two crossover fragments.

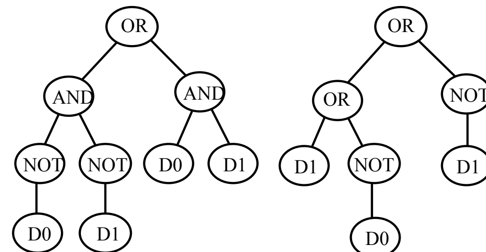

Figure 3 Offspring resulting from crossover.

When the crossover operation is applied to two individual parental computer programs chosen from the population in proportion to fitness, it effectively directs the future population towards parts of the search space containing computer programs that bear some resemblance to their parents. If the parents exhibited relatively high fitness because of certain structural features, their offspring may exhibit even higher fitness after a recombination of features.

Although one might think that computer programs are so brittle and epistatic that they could only be genetically bred in a few especially congenial problem domains, we have shows that computer programs can be genetically bred to solve a surprising variety of problems, including

- discovering inverse kinematic equations (e.g., to move a robot arm to designated target points) [Koza 1992a],
- planning (e.g., navigating an artificial ant along a trail),
- optimal control (e.g., balancing a broom and backing up a truck) [Koza and Keane 1990, Koza 1992d],
- Boolean function learning [Koza 1989, 1992c], and
- classification and pattern recognition (e.g., two intertwined spirals) [Koza 1992e].

A videotape visualization of the application of genetic programming to planning, emergent behavior, empirical discovery, inverse kinematics, and game playing can be found in the *Artificial Life II Video Proceedings* [Koza and Rice 1991].

Box Moving Robot

In the box moving problem, an autonomous mobile robot must find a box located in the middle of an irregularly shaped room and move it to the edge of the room within a reasonable amount of time.

The robot is able to move forward, turn right, and turn left. After the robots finds the box, it can move the box by

pushing against it. However, this sub-task may prove difficult because if the robot applies force not coaxial with the center of gravity of the box, the box will start to rotate. The robot will then lose contact with the box and will probably then fail to push the box to the wall in a reasonable amount of time.

The robot is considered successful if any part of the box touches any wall within the allotted amount of time.

The robot has 12 sonar sensors which report the distance to the the nearest object (whether wall or box) as a floating point number in feet. The twelve sonar sensors (each covering 30°) together provide 360° coverage around the robot.

The robot is capable of executing three primitive motor functions, namely, moving forward by a constant distance, turning right by 30°, and turning left by 30°. The three primitive motor functions `MF`, `TR`, and `TL` each take one time step (i.e., 1.0 seconds) to execute. All sonar distances are dynamically recomputed after each execution of a move or turn. The function `TR` (Turn Right) turns the robot 30° to the right (i.e., clockwise). The function `TL` (Turn Left) turns the robot 30° to the left (i.e., counter-clockwise). The function `MF` (Move Forward) causes the robot to move 1.0 feet forward in the direction it is currently facing in one time step. If the robot applies its force orthogonally to the midpoint of an edge of the box, it will move the box about 0.33 feet per time step.

The robot has a `BUMP` and a `STUCK` detector. We used a 2.5 foot wide box. The north (top) wall and west (left) wall of the irregularly shaped room are each 27.6 feet long.

The sonar sensors, the two binary sensors, and the three primitive motor functions are not labeled, ordered, or interpreted in any way. The robot does not know *a priori* what the sensors mean nor what the primitive motor functions do. Note that the robot does not operate on a cellular grid; its state variables assume a continuum of different values.

There are five major steps in preparing to use genetic programming, namely, determining (1) the set of terminals, (2) the set of functions, (3) the fitness measure, (4) the parameters and variables for controlling the run, and (5) the criteria for designating a result and for terminating a run.

The first major step in preparing to use genetic programming is to identify the set of terminals. We include the 12 sonar sensors and the three primitive motor functions (each taking no arguments) in the terminal set. Thus, the terminal set T for this problem is T = {`S00`, `S01`, `S02`, `S03`, ... , `S11`, `SS`, `(MF)`, `(TR)`, `(TL)`}.

The second major step in preparing to use genetic programming is to identify a sufficient set of primitive functions for the problem. The function set F consists of F = {`IFBMP`, `IFSTK`, `IFLTE`, `PROGN2`}.

The functions `IFBMP` and `IFSTK` are based on the `BUMP` detector and the `STUCK` detector defined by Mahadevan and Connell. Both of these functions take two arguments and evaluate their first argument if the detector is on and otherwise evaluates their second argument.

The `IFLTE` (If-Less-Than-or-Equal) function is a four-argument comparative branching operator that executes its third argument if its first argument is less than its second (i.e., then) argument and, otherwise, executes the fourth (i.e., else) argument. The operator `IFLTE` is implemented as a macro in LISP so that only either the third or fourth argument is evaluated depending on the outcome of the test involving the first and second argument. Since the terminals in this problem take on floating point values, this function is used to compare values of the terminals. `IFLTE` allows alternative actions to be executed based on comparisons of observation from the robot's environment. `IFLTE` allows, among other things, a particular action to be executed if the robot's environment is applicable and allows one action to suppress another. It also allows for the computation of the minimum of a subset of two or more sensors. `IFBMP` and `IFSTK` are similarly defined as macros

The connective function `PROGN2` taking two arguments evaluates both of its arguments, in sequence, and returns the value of its second argument.

Although the main functionality of the moving and turning functions lies in their side effects on the state of the robot, it is necessary, in order to have closure of the function set and terminal set, that these functions return

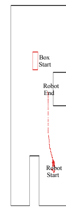

Figure 5 Typical random robot trajectory from generation 0.

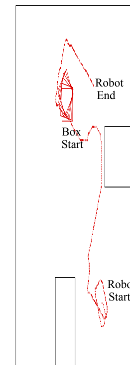

Figure 6 Trajectory of the best-of-generation individual for generation 0.

some numerical value. For Version 1 only, we decided that each of the moving and turning functions would return the minimum of the two distances reported by the two sensors that look forward. Also, for Version 1 only, we added one derived value, namely the terminal `SS` (Shortest Sonar) which is the minimum of the 12 sonar distances `S0`, `S1`, ... , `S11`, in the terminal set T.

The third major step in preparing to use genetic programming is the identification of the fitness function for evaluating how good a given computer program is at solving the problem at hand. A human programmer, of course, uses human intelligence to write a computer program. Intelligence is the guiding force that creates the program. In contrast, genetic programming is guided by the pressure exerted by the fitness measure and natural selection.

The fitness of an individual S-expression is computed using fitness cases in which the robot starts at various different positions in the room. The fitness measure for this problem is the sum of the distances, taken over four fitness cases, between the wall and the point on the box that is closest to the nearest wall at the time of termination of the fitness case.

A fitness case terminates upon execution of 350 time steps or when any part of the box touches a wall. If, for example, the box remains at its starting position for all four fitness cases, the fitness is 26.5 feet. If, for all four fitness cases, the box ends up touching the wall prior to timing out for all four fitness cases, the raw fitness is zero (and minimal).

The fourth major step in preparing to use genetic programming is selecting the values of certain parameters. The major parameters for controlling a run of genetic programming is the population size and the number of generations to be run. The population size is 500 here. A maximum of 51 generations (i.e. an initial random generation 0 plus 50 additional generations) is established for a run. Our choice of population size reflected an estimate on our part as to the likely complexity of the solution to this problem; however, we have used a population of 500 on about half of the numerous other problems on which we have applied genetic programming (Koza 1992a, 1992b, 1992c). In addition, each run is controlled by a number of minor parameters. The values of these minor parameters were chosen in the same way as described in the above references and were not chosen especially for this problem.

Finally, the fifth major step in preparing to use genetic programming is the selection of the criteria for terminating a run and accepting a result. We will terminate a given run when either (i) genetic programming produces a computer program which scores a fitness of zero, or (ii) 51 generations have been run. The best-so-far individual obtained in any generation will be designated as the result of the run.

Learning, in general, requires some experimentation in order to obtain the information needed by the learning algorithm to find the solution to the problem. Since learning here would require that the robot perform the overall task a certain large number of times, we planned to simulate the activities of the robot, rather than use a physical robot.

Version 1

Figure 5 shows the irregular room, the starting position of the box, and the starting position of the robot for the

Figure 7 Trajectory of the best-of-run individual with the robot starting in the southeast.

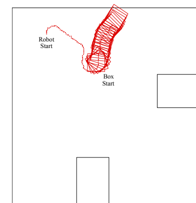

Figure 8 Trajectory of the best-of-run individual with the robot starting in the northwest .

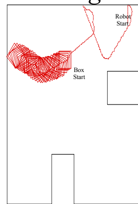

Figure 9 Trajectory of the best-of-run individual with the robot starting in the northeast.

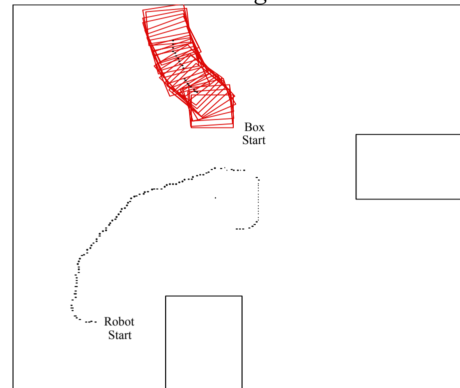

Figure 10 Trajectory of the best-of-run individual with the robot starting in the southwest.

particular fitness case in which the robot starts in the southeast part of the room. The raw fitness of a majority of the individual S-expressions from generation 0 is 26.5 (i.e., the sum, over the four fitness cases, of the distances to the nearest wall) since they cause the robot to stand still, to wander around aimlessly without ever finding the box, or, in the case of the individual program shown in the figure, to move toward the box without reaching it.

Even in generation 0, some individuals are better than others. Figure 6 shows the trajectory of the best-of-generation individual from generation 0 from one run. This individual containing 213 points finds the box and moves it a short distance for one of the four fitness cases, thereby scoring a raw fitness of 24.5.

The Darwinian operation of fitness proportionate reproduction and genetic crossover (sexual recombination) is now applied to the population, and a new generation of 500 S-expressions is produced. Fitness progressively improved between generations 1 and 6.

In generation 7, the best-of-generation individual succeeded in moving the box to the wall for one of the four fitness cases (i.e., it scored one hit). Its fitness was 21.52 and it had 59 points (i.e. functions and terminals) in its program tree.

By generation 22, the fitness of the best-of-generation individual improved to 17.55. Curiously, this individual, unlike many earlier individuals, did not succeed in actually moving the box to a wall for any of the fitness cases.

By generation 35, the best-of-generation individual had 259 points and a fitness of 10.77.

By generation 45 of the run, the best-of-generation individual computer program was successful, for all four fitness cases, in finding the box and pushing it to the wall within the available amount of time. Its fitness was zero. This best-of-run individual had 305 points.

Note that we did not pre-specify the size and shape of the solution to the problem. As we proceeded from generation to generation, the size and shape of the best-of-generation individuals changed. The number of points in the best-of-generation individual was 213 in generation 0, 59 in generation 7, 259 in generation 35, and 305 in generation 45. The structure of the S-expression emerged as a result of the selective pressure exerted by the fitness measure.

Figure 7 shows the trajectory of the robot and the box for the 305-point best-of-run individual from generation 45 for the fitness case where the robot starts in the southeast part of the room. For this fitness case, the robot moves more or less directly toward the box and then pushes the box almost flush to the wall.

Figure 8 shows the trajectory of the robot and the box for the fitness case where the robot starts in the northwest part of the room. Note that the robot clips the southwest corner of the box and thereby causes it to rotate in a counter clockwise direction until the box is moving almost north and the robot is at the midpoint of the south edge of the box.

Figure 9 shows the trajectory of the robot and the box for the fitness case where the robot starts in the northeast part of the room. For this fitness case, the robot's trajectory to reach the box is somewhat inefficient. However, once the robot reaches the box, the robot pushes the box more of less directly toward the west wall.

Figure 10 shows the trajectory of the robot and the box for the fitness case where the robot starts in the southwest part of the room.

Version 2

In the foregoing discussion of the box moving problem, the solution was facilitated by the presence of the sensor `SS` in the terminal set and the fact that the functions `MF`, `TL`, and `TR` returned a numerical value equal to the minimum of several designated sensors. This is in fact the way we solved it the first time.

This problem can, however, also be solved without the terminal `SS` being in the terminal set and with the three functions each returning a constant value of zero. We call these three new functions `MF0`, `TL0`, and `TR0`. The new function set is F_0 = {`MF0`, `TR0`, `TL0`, `IFLTE`, `PROGN2`}. We raised the population size from 500 to 2,000 in the belief that version 2 of this problem would be much more difficult to solve.

In our first (and only) run of Version 2 of this problem, an 100%-correct S-expression containing 207 points with a fitness of 0.0 emerged on generation 20:

```
(IFSTK (IFLTE (IFBMP (IFSTK (PROGN2 S02
S09) (IFSTK S10 S07)) (IFBMP (IFSTK (IFLTE
(MF0) (TR0) S05 S09) (IFBMP S09 S08))
(IFSTK S07 S11))) (IFBMP (IFBMP (PROGN2 S07
(TL0)) (PROGN2 (TL0) S03)) (IFBMP (PROGN2
(TL0) S03) (IFLTE S05 (TR0) (MF0) S00)))
(IFLTE (IFBMP S04 S00) (PROGN2 (IFLTE S08
S06 S07 S11) (IFLTE S07 S09 S10 S02))
(IFBMP (IFLTE (TL0) S08 S07 S02) (IFLTE S10
S00 (MF0) S08)) (IFBMP (PROGN2 S02 S09)
(IFBMP S08 S02))) (IFSTK (PROGN2 (PROGN2
S04 S06) (IFBMP (MF0) S03)) (PROGN2 (IFSTK
S05 (MF0)) (IFBMP (IFLTE (TR0) S08 (IFBMP
S07 S06) S02) (IFLTE S10 (IFBMP S10 S08)
(MF0) S08))))) (IFLTE (PROGN2 S04 S06)
(PROGN2 (IFSTK (IFBMP (MF0) S09) (IFLTE S10
S03 S03 S06)) (IFSTK (IFSTK S05 S01) (IFBMP
(MF0) S07))) (PROGN2 (IFLTE (IFSTK S01
(TR0)) (PROGN2 S06 (MF0)) (IFLTE S05 S00
(MF0) S08) (PROGN2 S11 S09)) (IFBMP (MF0)
(IFSTK S05 (IFBMP (PROGN2 (IFSTK (PROGN2
S07 S04) (IFLTE S00 S07 S06 S07)) (PROGN2
S04 S06)) (IFSTK (IFSTK (IFBMP S00 (PROGN2
S06 S10)) (IFSTK (MF0) S10)) (IFBMP (PROGN2
S08 S02) (IFSTK S09 S09))))))) (IFLTE
(IFBMP (PROGN2 S11 S09) (IFBMP S08 S11))
(PROGN2 (PROGN2 S06 S03) (IFBMP (IFBMP S08
S02) (MF0))) (IFSTK (IFLTE (MF0) (TR0) S05
S09) (IFBMP (PROGN2 (TL0) S02) S08)) (IFSTK
(PROGN2 S02 S03) (PROGN2 S01 S04)))))
```

Figure 11 shows the hits histogram for generations 0, 15, and 20. Note the left-to-right undulating movement of the center of mass of the histogram and the high point of the histogram. This "slinky" movement reflects the improvement of the population as a whole.

We have also employed genetic programming to evolve a computer program to control a wall following robot using the subsumption architecture (Koza 1992b) based on impressive work successfully done by Mataric (1990) in programming an autonomous mobile robot called TOTO.

Comparison with Reinforcement Learning

We now analyze the amount of preparation involved in what Mahadevan and Connell describe as "automatic programming" of an autonomous mobile robot to do box moving using reinforcement learning techniques, such as the Q learning technique developed by Watkins (1989).

Reinforcement learning techniques, in general, require that a discounted estimate of the expected future payoffs be calculated for each state. Q learning, in particular, requires that an expected payoff be calculated for each combination of state and action. These expected payoffs then guide the choices that are to be made by the system. Any calculation of expected payoff requires a statistically significant number of trials. Thus, reinforcement learning requires a large number of trials to be made over a large number of combinations of possibilities.

Before any reinforcement learning and "automatic programming" can take place, Mahadevan and Connell make the following 13 decisions:

First, Mahadevan and Connell began by deciding that the solution to this problem in the subsumption architecture requires precisely three task-achieving behaviors.

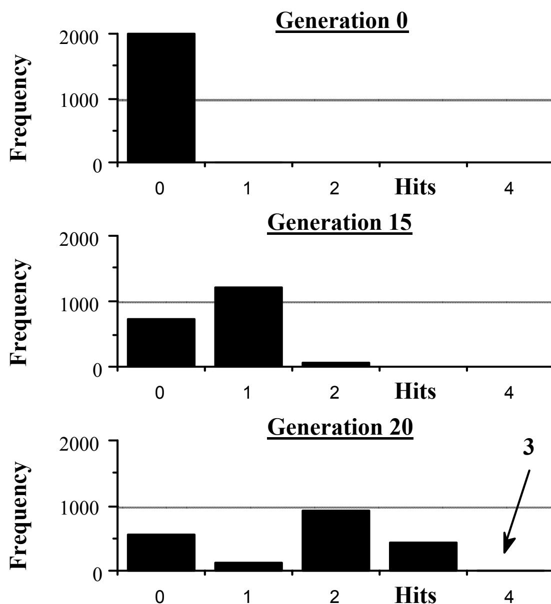

Figure 11 Hits histogram for generations 0, 15, and 20 for Version 2.

Second, they decided that the three task-achieving behaviors are finding the box, pushing the box across the room, and recovering from stalls where the robot has wedged itself or the box into a corner.

Third, they decided that the behavior of finding the box has the lowest priority, that the behavior of pushing the box across the room is of intermediate importance, and that the unwedging behavior has highest priority. That is, they decide upon the conflict-resolution procedure.

Fourth, they decided on the applicability predicate of each of the three task-achieving behaviors.

Fifth, they decided upon an unequal concentration of the sonar sensors around the periphery of the robot. In particular, they decided that half of the sensors will look forward, and, after a gap in sonar coverage, an additional quarter will look to the left and the remaining quarter will look to the right. They decided that none of the sonar sensors will look back.

Sixth, they decided to preprocess the real-valued distance information reported by the sonar sensors in a highly nonlinear problem-specific way. In particular, they condensed the real-valued distance information from each of the eight sensors into only two bits per sensor. The first bit associated with a given sensor (called the "`NEAR`" bit) is on if any object appears within 9 – 18 inches. The second bit (called the "`FAR`" bit) is on if any object appears within 18 – 30 inches. The sonar is an echo sonar that independently and simultaneously reports on both the 9 – 18 inch region and the 18 – 30 region. Thus, it is possible for both bits to be on at once. Note that the sizes of the two regions are unequal and that inputs in the range of less than 9 inches and greater than 30 inches are ignored. This massive state reduction and nonlinear preprocessing are apparently necessary because reinforcement learning requires calculation of a large number of trials for a large number of state-action combinations. In any event, eight real-valued variables are mapped into 16 bits in a highly nonlinear and problem-specific way. This reduces the input space from 12 floating-point values and two binary values to a relatively modest 2^{18} = 262,144 possible states. An indirect effect of ignoring inputs in the range of less than 9 inches and greater than 30 inches is that the robot must resort to blind random search if it is started from a point that is not within between 9 and 30 inches a box (i.e., the vast majority of potential starting points in the room).

Seventh, as a result of having destroyed the information contained in the eight real-valued sonar sensor inputs by condensing it into only 16 bits, they now decided to explicitly create a particular problem-specific matched filter to restore the robot's ability to move in the direction of an object. The filter senses (i) if the robot has just moved forward, and (ii) if any of four particular sonar sensors were off on the previous time step but any of these sensors are now on. The real-valued values reported by the sonar sensors contained sufficient information to locate the box. This especially tailored temporal filter is

required only because of the earlier decision to condense the information contained in the eight real-valued sonar sensor inputs into only 16 bits.

Eighth, they then decided upon a particular non-linear scale (–1, 0, +3) of three reinforcements based on the output of the matched filter just described.

Ninth, they then decided upon a different scale (+1 and –3) of two reinforcements based on an especially-defined composite event, namely the robot "continu[ing] to be bumped and going forward" versus "loses contact with the box."

Tenth, they defined a particular timer for the applicability predicate for the pushing behavior and the unwedging behavior (but not the finding behavior) lasting precisely five time steps. This timer keeps the behavior on for five time steps even if the applicability predicate is no longer satisfied by the state of the world, provided the behavior involved was once turned on.

Eleventh, they decided upon the scale (+1 and –3) for reinforcement of another especially defined composite event connected with the unwedging behavior involving "no longer [being] stalled and is able to go forward once again."

Twelfth, after reducing the total input space to 2^{18} = 262,144 possible states, they then further reduce the input space by consolidating input states that are within a specified Hamming distance. The Hamming distance is applied in a particular nonlinear way (i.e., weighting the NEAR bits by 5, the FAR bits by 2, the BUMPED bit by 1, and the STUCK bit by 1). This consolidation of the input space reduces it from size 2^{18} = 262,144 to size 2^9 = 512. This final consolidation to only 512 is again apparently necessitated because the reinforcement learning technique requires calculation of a large number of trials for a large number of state-action combinations.

Thirteenth, they then perform the reinforcement on each of the three behaviors separately.

The first four of the above 13 decisions constitute the bulk of the difficult definitional problem associated with the subsumption architecture. After these decisions are made, only the content of the three behavioral actions remains to be learned. The last nine of the decisions constitute the bulk of the difficulty associated with the reinforcement learning and executing the task. It is difficult to discern what "automatic programming" or "learning" remains to be done after these 13 decisions have been made. It is probably fair to say that the 13 preparatory decisions, taken together, probably require more analysis, intelligence, cleverness, and effort than programming the robot by hand.

In contrast, in preparing to use genetic programming on this problem, the five preparatory steps included determining the terminal set, determining the set of primitive functions, determining the fitness measure, determining the control parameters, and determining termination criterion and method of result designation. The terminal set and function set were obtained directly from the definition of the problem as stated by Mahadevan and Connell (with the minor changes already noted). Our effort in this area was no greater or less than that of Mahadevan and Connell. Defining the domain-specific fitness measure for this problem required a little thought, but, in fact, flows directly from the nature of the problem. Our choice of population size required the exercise of some judgment, but we used our usual default values for the minor control parameters. We used our usual termination criterion, and method of result designation.

In fact, determination of the fitness measure was the critical step in applying genetic programming to the problem. Genetic programming allows us to generate a complex structures (i.e., a computer program) from a fitness measure. As in nature, fitness begets structure.

References

Brooks, Rodney. 1986. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*. 2(1) March 1986.

Connell, Jonanthan. 1990. *Minimalist Mobile Robotics*. Boston, MA: Academic Press 1990.

Davis, Lawrence (editor). 1987. *Genetic Algorithms and Simulated Annealing*. London: Pittman l987.

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

Goldberg, David E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley l989.

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

Koza, John R. 1989. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*. 768-774. San Mateo, CA: Morgan Kaufman 1989.

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

Koza, John R. 1992b. Evolution of subsumption using genetic programming. In Bourgine, Paul and Varela, Francisco (editors). *Proceedings of European Conference on Artificial Life*. Cambridge, MA: MIT Press 1992.

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

Koza, John R. 1992d. A genetic approach to finding a controller to back up a tractor-trailer truck. In *Proceedings of the 1992 American Control Conference*. American Automatic Control Council 1992.

Koza, John R. 1992e. A genetic approach to the truck backer upper problem and the inter-twined spirals problem. In *Proceedings of International Joint*

Conference on Neural Networks, *Washington, June 1992.* IEEE Press.

Koza, John R. and Keane, Martin A. 1990. 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,France, June, 1990.* 47-56. Berlin: Springer-Verlag, 1990.

Koza, John R. and Rice, James P. 1991. A genetic approach to artificial intelligence. In Langton, C. G. (editor). *Artificial Life II Video Proceedings*. Redwood City, CA: Addison-Wesley 1991.

Mahadevan, Sridhar and Connell, Jonanthan. 1990. *Automatic Programming of Behavior-based Robots using Reinforcement Learning*. IBM Technical Report RC 16359. IBM Research Division 1990.

Mahadevan, Sridhar and Connell, Jonanthan. 1991. Automatic programming of behavior-based robots using reinforcement learning. In *Proceedings of Ninth National Conference on Artificial Intelligence*. 768-773. Volume 2. Menlo Park, CA: AAAI Press / MIT Press 1991.

Mataric, Maja J. 1990. *A Distributed Model for Mobile Robot Environment-Learning and Navigation*. MIT Artificial Intelligence Lab report AI-TR-1228. May 1990.

Watkins, Christopher. 1989. *Learning from Delayed Rewards*. PhD Thesis. King's College, Cambridge 1989.