Evolution of Subsumption Using Genetic Programming

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

The recently developed genetic programming paradigm is used to evolve emergent wall following behavior for an autonomous mobile robot using the subsumption architecture.

1. INTRODUCTION AND OVERVIEW

The repetitive application of seemingly simple rules can lead to complex overall emergent behavior. Emergent functionality means that overall functionality is not achieved in the conventional tightly coupled, centrally controlled way, but, instead, indirectly by the interaction of relatively primitive components with the world and among themselves [Steels 1991]. Emergent functionality is one of the main themes of research in artificial life [Langton 1989].

In this paper, we use the genetic programming paradigm to evolve a computer program that exhibits emergent behavior and enables an autonomous mobile robot to follow the walls of an irregularly shaped room. The evolutionary process is driven only by the fitness of the programs in solving the problem.

2. 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.

Current work in the field of genetic algorithms is reviewed in Goldberg [1989], Belew and Booker [1991], Davis [1987, 1991], Rawlins [1991] and Meyer and Wilson [1991].

3. BACKGROUND ON GENETIC PROGRAMMING

For many problems, the most natural representation for solutions are computer programs whose size, shape, and content have not been determined in advance. It is unnatural and difficult to represent computer programs of dynamically varying size and shape with fixed length character strings.

Although one might think that computer programs are so epistatic that they could only be genetically bred in a few especially congenial problem domains, we have shown that computer programs can be genetically bred to solve a surprising variety of problems in many different areas [Koza 1992], including

- emergent behavior (e.g. discovering a computer program which, when executed by all the ants in an ant colony, enables the ants to locate food, pick it up, carry it to the nest, and drop pheromones along the way so as to produce cooperative emergent behavior) [Koza 1991a],
- planning (e.g. navigating an artificial ant

along an irregular trail) [Koza 1990b],

- finding minimax strategies for games (e.g. differential pursuer-evader games; discrete games in extensive form) by both evolution and co-evolution [Koza 1991b],
- optimal control (e.g. centering a cart and balancing a broom in minimal time by applying a bang-bang force to the cart) (Koza and Keane 1990a, 1990b],
- machine learning of functions (e.g. learning the Boolean 11-multiplexer function) [Koza 1991d],
- generation of random numbers (using entropy as fitness) [Koza 1991c],
- symbolic regression, integration, differentiation, and symbolic solution to general functional equations for a solution in the form of a function (including differential equations with initial conditions, and integral equations) [Koza 1990], and
- simultaneous architectural design and training of neural nets [Koza and Rice 1991a].

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 1991b].

3.1. OBJECTS IN GENETIC PROGRAMMING

In genetic programming, 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. The set of terminals used typically includes inputs (sensors) appropriate to the problem domain and possibly various constants. Each function in the function set should be well defined for any combination of elements from the range of every function that it may encounter and every terminal that it may encounter.

One can now view the search for a solution to the problem as a search in the hyperspace of all possible compositions of functions and terminals (i.e. computer programs) 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.

3.2. OPERATIONS IN GENETIC PROGRAMMING

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

The reproduction operation copies an individual in the population into the new population for the next generation.

The crossover (recombination) 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.

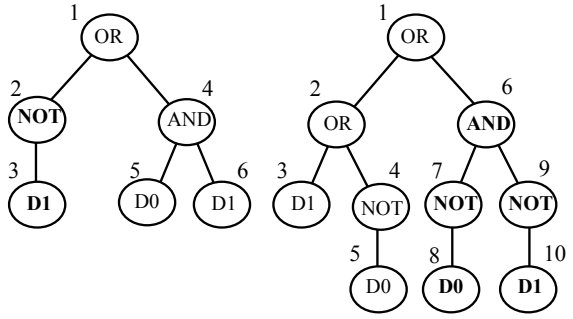

Figure 1: Two parental computer programs shown as trees with ordered branches. Internal points of the tree correspond to functions (i.e. operations) and external points correspond to terminals (i.e. input data).

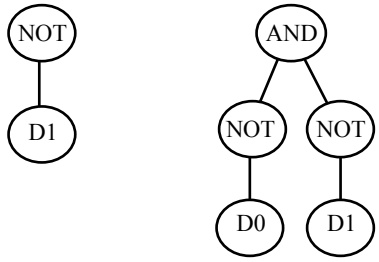

Figure 2: The two crossover fragments

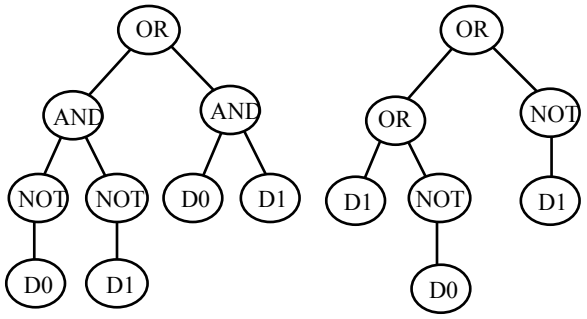

Figure 3: Offspring resulting from crossover

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 points of 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 no. 2 (out of 6 points of the first parent) is randomly selected as the crossover point for the first parent and that point no. 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 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 resulting from the crossover.

Note that the first offspring in Figure 3 is an S-expression for the Boolean even-parity (i.e. equal) function, namely

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

3.3. EXECUTION OF GENETIC PROGRAMMING

The genetic programming paradigm, 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 two 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 genetic programming paradigm. This result may be a solution (or approximate solution) to the problem.

4. THE WALL FOLLOWING PROBLEM

Mataric [1990] described the problem of controlling an autonomous mobile robot to perform the task of following the walls of an irregular room.

The robot is capable of executing the following five primitive motor functions: moving forward by a constant distance, moving backward by a constant distance, turning right by 30 degrees, turning left by 30 degrees, and

stopping.

The robot has 12 sonar sensors which report the distance to the nearest wall. Each sonar sensor covers a 30 degree sector around the robot. In addition, there was a sensor for the STOPPED condition of the robot.

Figure 4 shows an irregularly shaped room and the distances reported by the 12 sonar sensors. The robot is shown at point (12, 16) near the center of the room. The north (top) wall and west (left) wall are each 27.6 feet long.

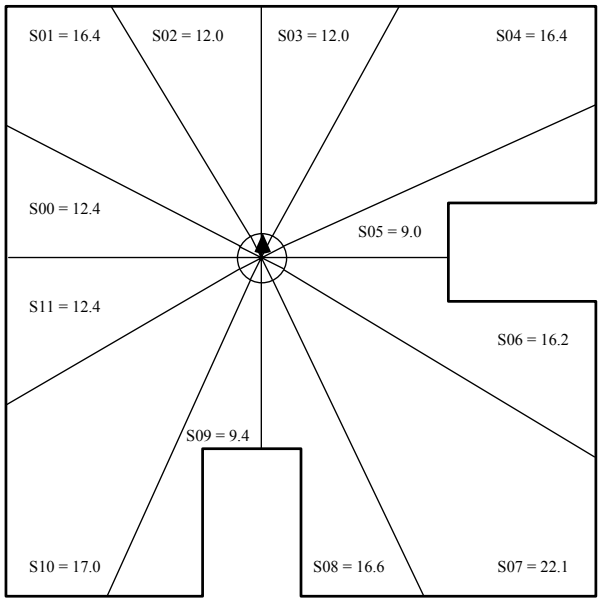

Figure 4: Irregular room with robot with 12 sonar sensors located near middle of the room

One can envision solving the wall following problem in one of three approaches, namely

(1) the conventional approach to building control systems for for autonomous mobile robots,
(2) the subsumption architecture, and
(3) genetic programming.

Regardless of which of these three approaches is used to solve the wall following problem, the 13 sensors can be viewed as the input to an as-yet unwritten computer program in a style appropriate to the three approaches. This as-yet unwritten program will process these inputs and will cause the activation, in some order, of the various primitive motor functions. This as-yet-unwritten program will be a composition of the five available primitive motor functions and the 13 available sensors.

Note that the 13 sensors and five primitive motor functions are the starting point of all three approaches. They are given as part of the statement of the problem.

5. THE CONVENTIONAL ROBOTIC APPROACH

The conventional approach to building control systems for autonomous mobile robots is to decompose the overall problem into a series of functional units that perform functions such as perception, modeling, planning, task execution, and motor control. A central control system then executes each functional unit in this decomposition and then passes the results on to the next functional unit in an orderly, closely coupled, and synchronized manner. For example, the perception unit senses the world and the results of this sensing are then passed to a modeling module which attempts to build an internal model of the perceived world. The internal model resulting from this modeling is then passed on to a planning unit which computes a plan. The plan might be devised by a consistent and logically sound technique (e.g. resolution and unification) or it might be devised by one of the many heuristic techniques of symbolic artificial intelligence. In any event, the resulting plan is passed on to the task execution unit which then executes the plan by calling on the motor control unit. The motor control unit then acts directly on the external world.

In this conventional approach, only a few of the functional units (e.g. the perception unit and motor control unit) typically are in direct communication with the world. All functional units must typically be executed in their intended orderly, closely coupled, and synchronized manner in order to make the robot do anything.

6. THE SUBSUMPTION ARCHITECTURE

The subsumption architecture decomposes the problem into a set of asynchronous task achieving behaviors [Brooks 1986, 1989]. The task achieving behaviors for an autonomous mobile robot might include behaviors such as avoiding objects, wandering, exploring,

identifying objects, building maps, planning changes to the world, monitoring changes, and reasoning about behavior of the objects. The task achieving behaviors operate locally and asychronously and are only loosely coupled to one another. In contrast to the conventional approach, each of the task achieving behaviors is typically in direct communication with the world (and each other). The task achieving behaviors in the subsumption architecture are typically much more primitive than the functional units of the conventional approach.

In the subsumption architecture, various subsets of the task achieving behaviors typically exhibit some partial competence in solving a simpler version of the overall problem. Thus, the solution to a more complex version of a problem can potentially be built up by incrementally adding new independent acting parts to existing parts. In addition, the system may be fault tolerant in the sense that the failure of one part does not cause complete failure, but, instead, causes a gracefully degradation of performance to some lower level. In contrast, in the conventional approach, the various functional units have no functionality when operating separately and there is a complete suspension of all performance when one functional unit fails.

In the subsumption architecture, the task achieving behaviors each consist of an applicability predicate, a gate, and a behavioral action. If the current environment satisfies the applicability predicate of a particular behavior, the gate allows the output of the behavioral action to feed out onto the output line of that behavior. Potential conflicts among behavioral actions are resolved by a hierarchical arrangement of suppressor nodes. As a simple example, suppose that there are three task achieving behaviors with strictly decreasing priority. The applicability predicates and the suppressor nodes of these three behaviors are equivalent to the following composition of ordinary IF conditional functions:

```
(IF A-P-1 BEHAVIOR-1
          (IF A-P-2 BEHAVIOR-2
                    (IF A-P-3 BEHAVIOR-3)
```

In particular, if the first applicability predicate (A-P-1) is satisfied, then BEHAVIOR-1 is executed. Otherwise, if A-P-2 is satisfied, BEHAVIOR-2 is executed. Otherwise, the lowest priority behavior (i.e. BEHAVIOR-3) is executed.

Mataric (1990) has implemented the subsumption architecture for controlling an autonomous mobile robot by conceiving and writing a set of four programs for performing four task achieving behaviors. The four behaviors together enable a mobile robot called TOTO to follow the walls in an irregular room.

Starting with the five primitive motor functions and the 13 sensors that are part of the definition of the problem, Mataric applied her intelligence and ingenuity and conceived of a set of four task achieving behaviors which together enable a mobile robot to follow the walls in an irregular room. As a matter of preference, Mataric specifically selected her four task achieving behaviors so that their applicability predicates were mutually exclusive (thus eliminating the need for a conflict resolution architecture allowing one task achieving behavior to suppress the behavior of another).

Mataric then wrote a set of four LISP programs for performing the four task achieving behaviors. Mataric's four LISP programs corresponded to the four task achieving behaviors and were called STROLL, AVOID, ALIGN, and CORRECT. Each of these four task achieving behaviors interacted directly with the world and each other.

Various subsets of Mataric's four behaviors exhibited some partial competence in solving part of the overall problem. For example, the robot became capable of collission free wandering with only the STROLL and AVOID behaviors. The robot became capable of tracing convex boundaries with only the addition of only the ALIGN behavior to these first two behaviors. Finally, the robot became capable of general boundary tracing with the further addition of the CORRECT behavioral unit.

Mataric's four LISP programs included nine LISP functions (namely, COND, AND, NOT, IF, >, >=, =, <=, and >). In addition, her four programs internally made use of three constant

parameters (defining an edging distance EDG, minimum safe distance MSD, and danger zone DZ), the minimum of all 12 sonar distances (called "Shortest Sonar" or SS), and eight other internally defined variables representing the minimum of various thoughtfully chosen subsets of the 12 sonar distances (e.g. the dynamically computed minimum of a particular three forward facing sensors).

In total, Mataric's four LISP programs consisted a composition of 151 functions and terminals.

The fact that Mataric was able to write four programs enabling an autonomous robot to perform the task of following the wall of an irregular room is evidence (based on this particular problem) for one of the claims of the subsumption architecture, namely, that it is possible to build a control system for an autonomous mobile robot using loosely coupled, asynchronous task achieving behaviors.

Note that if Mataric had wanted to write a computer program for wall following using conventional coupled synchronous robotic techniques, her program would have taken in the same 13 sensors as inputs and caused the activation, in some order, of the same five primitive motor functions as the output of the program.

The conception and design of suitable task achieving behaviors for the subsumption architecture requires considerable ingenuity and skill on the part of the human programmer.

7. APPLICATION OF GENETIC PROGRAMMING TO THE WALL FOLLOWING PROBLEM

The question arises as to whether an autonomous mobile robot can learn to perform wall following in an evolutionary way, and, in particular, by using genetic programming. This learning would include learning both the necessary task achieving behaviors (including the applicability predicates and behavioral actions) and the conflict resolution hierarchy.

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

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

The first major step in preparing to use genetic programming is to identify the set of terminals. The genetic programming paradigm genetically creates a computer program that takes certain inputs and produces outputs in order to successfully perform a specified task. The inputs to this program usually come from the statement of the problem. For the wall following problem, the potential inputs to the computer program consist of the 13 available sensors. These are the same 13 sensors which one would use if one were attempting to perform wall following with the conventional robotic approach or the subsumption architecture.

In reviewing the 13 sensors, we concluded that we had no use for the STOPPED sensor since our simulated robot could not be damaged by running into a wall in the course of a computer simulation. Moreover, we did not want our simulated robot to ever stop. Thus, we deleted the STOPPED sensor, the STOP primitive function, and the constant parameter for the danger zone DZ.

We retained Mataric's other two constant numerical parameters (i.e. the edging distance EDG and the minimum safe distance MSD). We retained Mataric's overall minimum sensor SS. However, we did not use any of her eight derived values representing specific subsets of sonar sensors. Human programmers find it convenient to create and refer to such intermediate variables in their programs.

Thus, our terminal set consisted of 15 items, namely,

```
T = {S00, S01, S02, S03,
     ..., S11, SS, MSD, EDG}
```

In other words, at each time step of the simulation, our simulated robot will have access to these 15 floating point values.

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

We start with the five given primitive motor functions that are part of the statement of this problem. As previously mentioned, we had no use for the STOP function. Since we want to evolve a subsumption architecture and we observed above that the subsumption architecture can be viewed as a composition of ordinary IF conditional functions, we included a single simple decision making function (IFLTE) in the function set. The function IFLTE (If-Less-Than-Or-Equal) takes four arguments. If the value of the first argument is less than or equal the value of the second argument, the third argument is evaluated and returned. Otherwise, the fourth argument is evaluated and returned.

We also included a connective function (PROGN2) in our function set. The connective function PROGN2 taking two arguments evaluates both of its arguments, in order, and returns the result of evaluating its second argument.

Thus, the function set F for this problem consists of four of the five given primitive motor functions (i.e. TR, TL, MF, and MB as described below), the decision function IFLTE, and the connective PROGN2. That is, the function set F is

```
F = {TR, TL, MF, MB, IFLTE,
     PROGN2}
```

The function TR (Turn Right) turns the robot 30 degrees to the right (i.e. clockwise).

The function TL (Turn Left) turns the robot 30 degrees to the left (i.e. counter-clockwise).

We achieved the same effect as the STOP function by letting our robot push up against the wall, and, if no change of state occurs after one time step, the robot is viewed as having stopped. Because we were not concerned with physically damaging our simulated robot, we did not include Mataric's primitive motor function STOP for stopping the robot (e.g. when it is about to invade the danger zone DZ and possibly damage itself).

The function MF (Move Forward) causes the robot to move 1.0 feet forward in the direction it is currently facing. If any of the six forward looking sonar sensors (i.e. S00 through S05) report a distance to any wall of less than 110% of the distance to be moved, no movement occurs.

The function MB (Move Backward) causes the robot to move 1.3 feet backwards. If any of the six backward looking sonar sensors (i.e. S06 though S11) report a distance to any wall of less than 110% of the distance to be moved, no movement occurs.

All sonar distances are dynamically recomputed after each execution of a move or turn. Each of the moving and turning functions returns the minimum of the two distances reported by the two sensors (i.e. S02 and S03) that look in the direction of forward movement (i.e. S02 representing the 11:30 o'clock direction and S03 representing the 12:30 o'clock direction).

The functions MF, MB, TR, and TL each take one time step (i.e. 1.0 seconds) to execute.

The third major step in preparing to use genetic programming is identification of the fitness function for evaluating how good a given computer program is at solving the problem at hand.

A wall following robot may be viewed as a robot that travels along the entire perimeter of the irregularly shaped room. Noting that the edging distance is 2.3 feet, we proceed to define the fitness measure for this problem by placing 2.3 foot square tiles along the perimeter of the room. Twelve such tiles fit along the 27.6 foot north wall and 12 such tiles fit along the 27.6 foot west wall. A total of 56 tiles are required to cover the entire periphery of the room.

Figure 5 shows the room with the 56 tiles (each with a filled circle at its center). The robot is shown in the middle of the room at its starting position (12, 16) facing in its starting direction (i.e. south).

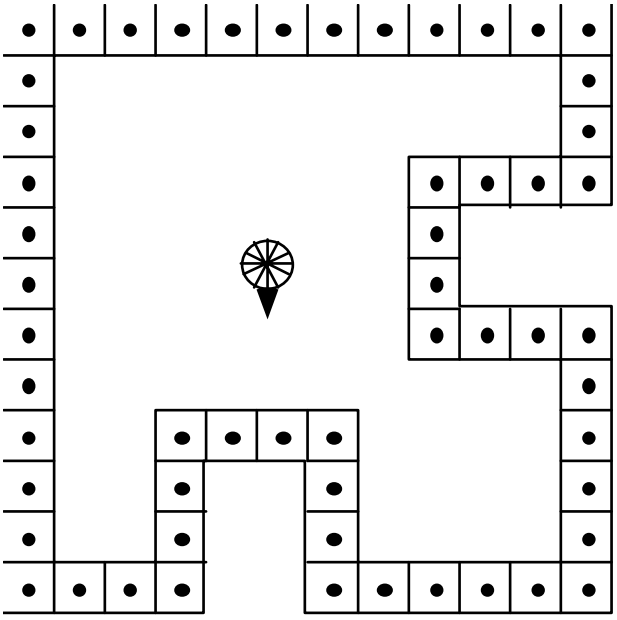

Figure 5: Room with 56 tiles on periphery showing robot at its starting position (12,16) facing south.

We defined the fitness of an individual S-expression in the population to be the number of tiles (from 0 to 56) that are touched by the robot within the allotted period of time (i.e. 400 time steps).

The fourth major step in preparing to use genetic programming is selecting the values of certain parameters. The population size is 1000 here. Each new generation is 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 both parents selected with a probability proportionate to fitness). In selecting crossover points, 90% were internal (function) points of the tree and 10% were external (terminal) points of the tree. For the practical reason of conserving computer time, the depth of initial random S-expressions was limited to 4 and the depth of S-expressions created by crossover was limited to 15.

Finally, the fifth major step in preparing to use genetic programming is the selection of the criterion for terminating a run and accepting a result. We will terminate a given run when either (i) genetic programming produces a computer program which achieves the maximal value for fitness (i.e. 56 out of 56), or (ii) 101 generations have been run.

Note that in performing these five preparatory steps, we made use only of the information provided in the basic statement of the problem (with the modifications needed because we did not intend to allow our simulated robot ever to stop during the course of our computer simulations). We *did not* use any of Mataric's thoughtfully chosen subsets of sensors nor did we use any knowledge about the four task achieving behaviors which Mataric conceived and defined. We *did*, however, define a way to measure fitness in performing wall following. We *did* use the 12 sonar sensors, two of the three constant numerical parameters, and four of the five primitive motor functions that were part of the statement of the problem.

8. RESULTS

In one run of the genetic programming paradigm on this problem, 57% of the individuals in the population in the initial random generation (i.e. generation 0) scored a fitness of zero (out of a possible 56). Many of these zero-scoring S-expressions merely caused the robot to turn without ever moving while others caused the robot to wander aimlessly in circles in the middle of the room. About 20% of the individuals from generation 0 were wall-bangers which scored precisely one because they headed for a wall and continued to push up against it.

The best single individual from generation 0 scored 17 (out of 56). This S-expression consists of 17 points (i.e. functions and terminals) and is shown below:

```
(IFLTE (PROGN2 MSD (TL))
       (IFLTE S06 S03 EDG (MF))
       (IFLTE MSD EDG S05 S06)
       (PROGN2 MSD (MF)))
```

Figure 6 shows the looping trajectory of the robot while executing this best-of-generation program for generation 0. The 39 filled circles along the periphery of the room represent the 39 of the 56 tiles that were not touched by the robot before it timed out. As can be seen, this individual starts in the middle of the room and circles on itself three times. It then begins a series of 11 loops which cause the robot to repeatedly hit the wall at irregular intervals.

This looping leaves many intervening points along the wall untouched. This individual time outs on the west wall after 400 time steps. By generation 2, the best-of-generation individual scored 27. This S-expression consisted of 57 points.

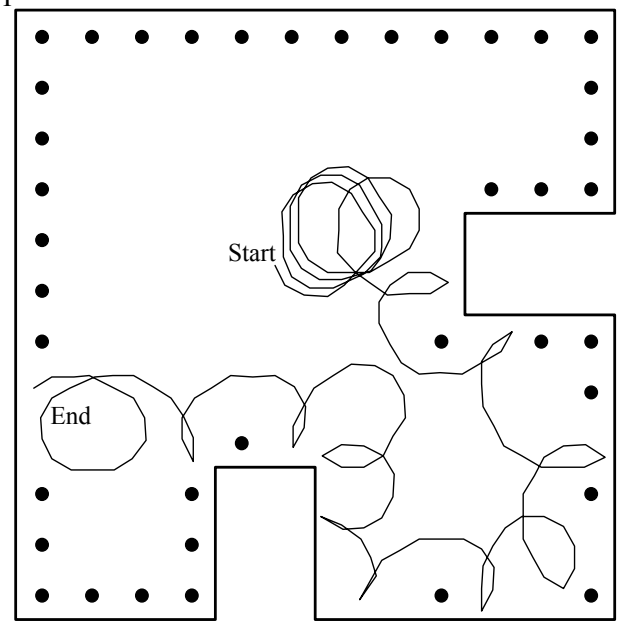

Figure 6: Looping trajectory from generation 0 of the best-of-generation individual (scoring 17).

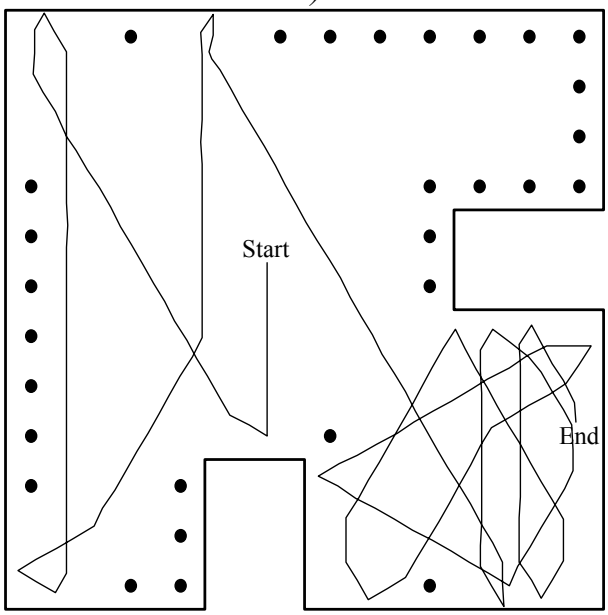

Figure 7: Ricocheting trajectory from generation 2 of the best-of-generation individual (scoring 27).

Figure 7 shows the ricocheting trajectory of the robot while executing this best-of-generation program for generation 2. As can be seen, this individual causes the robot to touch occasional points on the periphery of the room as the robot ricochets around the room 16 times.

Although this ricocheting individual from generation 2 is far from perfect, it is considerably better than the static and aimless wandering individuals (both scoring zero) from generation 0, the wall-banging individuals (scoring one) from generation 0, and the best-of-generation looping individual from generation 0 (scoring 17).

By generation 14, the best-of-generation S-expression scored 49 and consisted of 45 points. Figure 8 shows the trajectory of the robot while executing this best-of-generation program for generation 14. After once reaching a wall, this individual slithers in broad snake like motions along the walls and never returns to the middle of the room. In scoring 49 out of 56, it misses five corners and two points in the middle of walls.

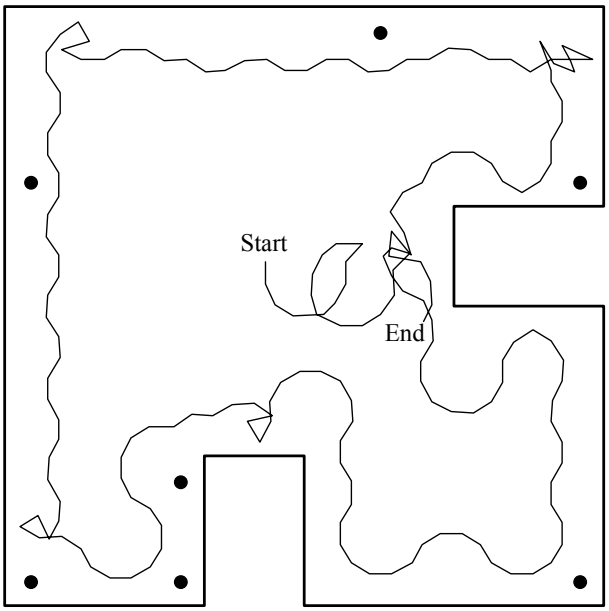

Figure 8: Broad snake like trajectory from generation 14 of best-of-generation individual (scoring 49 out of 56).

Finally, in generation 57, the best-of-generation S-expression scored a perfect 56 out of 56 This S-expression consisted of 145 points and is shown below:

```
(IFLTE (IFLTE S10 S05 S02 S05) (IFLTE
(PROGN2 S11 S07) (PROGN2 (PROGN2 (PROGN2
S11 S05) (PROGN2 (PROGN2 S11 S05) (PROGN2
(MF) EDG))) SS) (PROGN2 (PROGN2 (IFLTE S02
(PROGN2 S11 S07) S04 (PROGN2 S11 S05))
(TL)) (MB)) (IFLTE S01 EDG (TR) (TL)))
```

```
(PROGN2 SS S08) (IFLTE (IFLTE (PROGN2 S11
S07) (PROGN2 (PROGN2 (PROGN2 S10 S05)
(PROGN2 (PROGN2 S11 S05) (PROGN2 (MF)
EDG))) SS) (PROGN2 (PROGN2 S01 (PROGN2
(IFLTE S07 (IFLTE S02 (PROGN2 (IFLTE SS EDG
(TR) (TL)) (MB)) S04 S10) S04 S10) (TL)))
(MB)) (IFLTE S01 EDG (TR) (TL))) (PROGN2
S05 SS) (PROGN2 (PROGN2 MSD (PROGN2 S11
S05)) (PROGN2 (IFLTE (PROGN2 (TR) (TR))
(PROGN2 S01 (PROGN2 (IFLTE S02 (TL) S04
(MB)) (TL))) (PROGN2 S07 (PROGN2 (PROGN2
(MF) EDG) EDG)) (IFLTE SS EDG (PROGN2
(PROGN2 (PROGN2 S02 S05) (PROGN2 (PROGN2
S11 S05) (PROGN2 (IFLTE S02 (TL) S04 S10)
EDG))) SS) (TL))) S08)) (IFLTE SS EDG (TR)
(TL))))
```

This program consists of a composition of conditional statements which test various sensors from the environment and invoke various given primitive motor functions of the robot in order to perform wall following. In other words, this program is a program in the subsumption architecture.

We can simplify this S-expression to the following S-expression containing 59 points:

```
(IFLTE (IFLTE S10 S05 S02 S05)
       (IFLTE S07 (PROGN2 (MF) SS)
          (PROGN2 (TL) (MB))
          (IFLTE S01 EDG (TR) (TL)))
   *
   (IFLTE (IFLTE S07 (PROGN2 (MF) SS)
              (PROGN (IFLTE SS EDG (TR)
(TL))
                     (MB) (TL) (MB))
              (IFLTE S01 EDG (TR) (TL)))
           SS
      (IFLTE (PROGN2 (TR) (TR))
             (PROGN2 (IFLTE S02 (TL) *
(MB))
                     (TL))
         (MF)
         (TL))
      (IFLTE SS EDG (TR) (TL))))
```

In this S-expression, the asterisks indicate subexpressions that are free of side-effects and which are just returned as the value of the expression (i.e. are executed, but are inconsequential).

Figure 9 shows the trajectory of the robot while executing this best-of-generation program for generation 57. This individual starts by briefly moving at random in the middle of the room. However, as soon as it reaches the wall, it moves along the wall and stays close to the wall. It touches l00% of the 56 tiles along the periphery of the room.

Note that the progressive change in size and shape of the individuals in the population is a characteristic of genetic programming. The size (i.e. 145 points) and particular hierarchical structure of the best-of-generation individual from generation 57 was not specified in advance. Instead, the entire structure evolved as a result of reproduction, crossover, and the relentless pressure of fitness. That is, fitness caused the development of the structure.

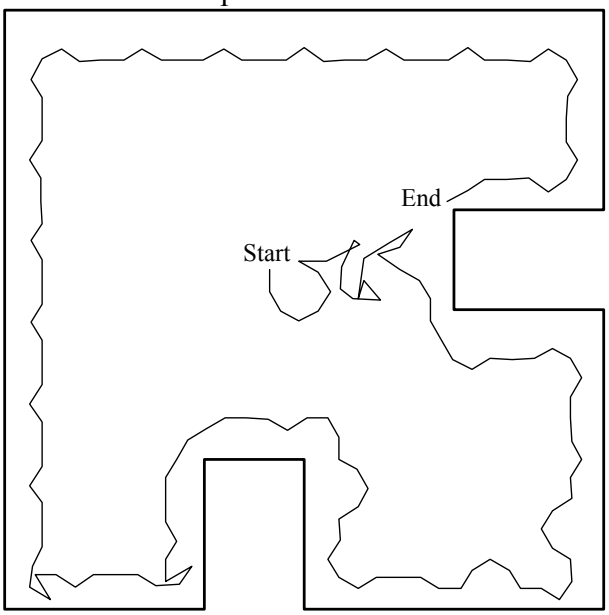

Figure 9: Wall following trajectory of the best-of-generation individual (scoring 56 out of 56) from generation 57.

Although a program written by a human programmer cannot be directly compared to the program generated using genetic programming, it is, nonetheless, interesting to note that the 145 points of this S-expression is similar to the 151 points in Mataric's four LISP programs.

We have obtained similar results on other runs of this problem.

9. CONCLUSIONS

We demonstrated that it is possible to use the genetic programming paradigm to breed a computer program to enable a robot to follow the wall of an irregular room.

The program we discovered consisted of a composition of conditional statements which tested various sensors from the environment and invoked various given primitive motor functions of the robot in order to perform wall following. In other words, this program is a program in the subsumption architecture. Thus, we have demonstrated the evolution of a program in the

subsumption architecture using an evolutionary process that evolves structures guided only by a fitness measure.

The fact that it is possible to evolve a subsumption architecture to solve a particular problem suggests that this approach to decomposing problems may be useful in building up solutions to difficult problems by aggregating task achieving behaviors until the problem is solved.

10. ACKNOWLEDGMENTS

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

11. REFERENCES

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

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

Brooks, Rodney. A robot that walks: emergent behaviors from a carefully evolved network. *Neural Computation* 1(2), 253-262. 1989.

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

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

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

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

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

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

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

Koza, John R. Evolution and co-evolution of computer programs to control independent-acting agents. In Meyer and Wilson below. 1991b.

Koza, John R. Evolving a computer program to generate random numbers using the genetic programming paradigm. In Belew and Booker above. 1991c.

Koza, John R. A hierarchical approach to learning the Boolean multiplexer function. In Rawlins below. 1991d.

Koza, John R. *Genetic Programming*. Cambridge, MA: MIT Press, 1992 (forthcoming).

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.* Berlin: Springer-Verlag, 1990a.

Koza, John R. and Keane, Martin. Cart centering and broom balancing by genetically breeding populations of control strategy programs. In *Proceedings of International Joint Conference on Neural Networks*, *Washington, January, 1990*. Volume I. Hillsdale, NJ: Lawrence Erlbaum 1990b.

Koza, John R. and Rice, James P. Genetic generation of both the weights and architecture for a neural network. In *Proceedings of International Joint Conference on Neural Networks*, *Seattle, July 1991*. 1991a

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

Meyer, Jean-Arcady and Wilson, Stewart W. *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior.* Paris. September 24-28, 1990. MIT Press, Cambridge, MA, 1991.

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

Langton, Christopher G. *Artificial Life, Santa Fe Institute Studies in the Sciences of Complexity*. Volume VI. Redwood City, CA: Addison-Wesley. 1989.

Rawlins, Gregory (editor). *Proceedings of Workshop on the Foundations of Genetic Algorithms and Classifier Systems*. *Bloomington, Indiana. July 15-18, 1990*. San Mateo, CA: Morgan Kaufmann 1991.

Steels, Luc. Towards a theory of emergent functionality. In Meyer, Jean-Arcady and Wilson, Stewart W. *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior.* Paris. September 24-28, 1990. Cambridge, MA: MIT Press 1991.