Simultaneous Discovery of Detectors and a Way of Using the Detectors via Genetic Programming

John R. Koza
Computer Science Department
Margaret Jacks Hall
Stanford University
Stanford, California 94305
Koza@cs.stanford.edu
415-941-0336

Abstract—Conventional approaches to problems of pattern recognition and machine learning usually require that the user hand-craft detectors for key features in the problem environment. Conventional approaches often additionally require the user to specify in advance the size and shape of the eventual way of combining the detectors into a compete solution. This paper describes a general approach for simultaneously discovering detectors and a way of combining the detectors to solve a problem via genetic programming with automatic function definition. Genetic programming provides a way to genetically breed a computer program to solve a wide variety of problems. Automatic function definition automatically and dynamically enables genetic programming to define potentially useful functions dynamically during a run and to facilitate a solution of a problem by automatically and dynamically decomposing the problem into simpler subproblems. This approach is illustrated with a problem of letter recognition.

I Introduction and Overview

Conventional approaches to pattern recognition and machine learning usually require that the user hand-craft detectors for key features in the problem environment. Conventional approaches often additionally require the user to specify in advance the size and shape of the eventual way of combining the detectors into a compete solution. However, in many instances, finding the detectors and a way of combining the detectors in order to solve the problem really is *the problem*. Indeed, the necessity for pre-identification of the particular components of solutions and the necessity for pre-determination of a way of combining these components has been recognized as a bane of machine learning starting with Samuel's ground-breaking work in machine learning involving learning to play the game of checkers [Samuel [1959].

In Samuel's checker player, learning consisted of progressively adjusting numerical coefficients in an algebraic expression of a predetermined functional form (specifically, a polynomial of specified order). Each component term of the polynomial represented a hand-crafted detector reflecting some aspect of the current state of the board (e.g., number of pieces, center control, etc.). The polynomial weighted each detector with a numerical coefficient and thereby assigned a single numerical value of a board to the player. If a polynomial were good at assigning values to boards, the polynomial could be used to compare the boards that would arise if the player were to make various alternative moves – thus permitting the best move to be selected from among the alternatives on the basis of the polynomial. In Samuel's learning system, the numerical coefficients of the polynomial were adjusted with experience, so that the predictive quality of the polynomial progressively improved. Samuel predetermined the way the detectors would be combined to solve the problem by selecting the functional form of the polynomial. Samuel recognized, from the beginning, the importance of enabling learning to occur without predetermining the size and shape of the solution and of "[getting] the program to generate its own parameters [detectors] for the evaluation polynomial."

This paper describes a general approach for simultaneously discovering detectors and a way of combining the detectors to solve a problem. The approach involves using genetic programming with automatic function definition to evolve a solution to the problem.

Genetic programming provides a way to search the space of all possible programs composed of certain terminals and primitive functions to find a function which solves, or approximately solves, a problem.

Automatic function definition enables genetic programming to define potentially useful functions automatically and dynamically during a run and also to combine these defined functions dynamically during a run in order to solve a problem.

Since the metaphor of a detector is especially apt in the field of pattern recognition, our general approach will be illustrated by means of a problem from the field of pattern recognition.

Sections I, III, and VI of the article by Koza, Keane, and Rice [1993] in these ICNN-93 proceedings provide necessary background on genetic programming and automatic function definition for this article**.**

Section II of this article states the illustrative pattern recognition problem. Section III details the preparatory steps for applying genetic programming with automatic function definition to the problem. Section IV describes a solution to the problem.

II Letter Recognition Problem

Since the metaphor of a detector is especially apt in the field of pattern recongition, we will illustrate our approach to the problem of finding detectors and finding a way to combine them with a problem of letter recognition from the field of pattern recognition.

Fig. 1 shows the letters I and L, each presented in a 6 by 4 pixel grid of binary (ON or OFF) values.

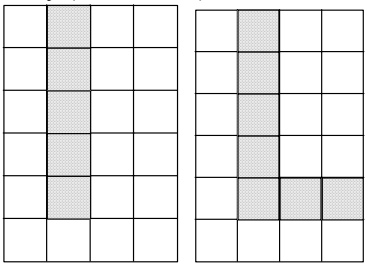

Fig. 1 The letters I and L

The goal of pattern recognition is to discover a computer program that can take any of the 2^{24} possible patterns of bits as its input and produce a correct identification I, L, or `NIL` (i.e., not the letter I or the letter L) for the pattern as its output.

Note that the correct identification of a pattern of pixels requires not only establishing that all the specific pixels that must be ON are indeed ON, but also inspecting other pixels on the grid to exclude the possiblity of an imperfect letter or another letter.

Fig. 2 shows two patterns that are neither the letter I nor the letter L. The pattern on the left has a misplaced ON pixel in the lower right hand corner of the grid instead of as the rightmost pixel of the bottom of the L. The second pattern has a missing ON pixel in an otherwise correct letter L.

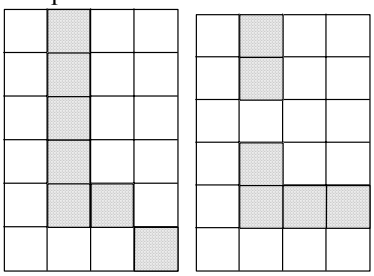

Fig. 2 Two patterns that should not be recognized as the letter L

There are, of course, many different ways to structure a computer program to perform the task of letter recognition. The particular approach presented in this paper was chosen to illustrate and emphasize the discovery of *local* detectors and *hierarchical* combinations of the detectors to solve a problem.

If one were trying to verbally describe the letter L to someone who was not familiar with the Roman alphabet, one might give a dynamic description involving drawing a vertical line of five pixels downwards (south) from some specified starting location in the 6 by 4 grid and then drawing a horizontal line of two pixels to the right (east). This dynamic description of the pattern contains both locally-acting and hierarchical aspects. This can be implemented using a slow-moving turtle with limited vision. The turtle starts at a designated starting location on the grid and can move one step at a time to the north (up), south (down), east (right), or west (left). The turtle's vision is limited to its immediate neighborhood consisting of the nine pixels centered at its current location. The one pixel where the turtle is currently located is called "`X`" (center) and the eight neighboring pixels at distance one are called `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, and `NW`.

One way to structure a computer program to recognize letters is for the "main" result-producing branch to be a decision tree that produces a return value consisting of an identification of the pattern (I, L, or `NIL`). The use of a decision tree structure in the result-producing branch permits easy handling of arbitrary numbers of possible outcomes. The result-producing branch will activate various alternative actions and, ultimately, return an identification of the pattern. The desired bias toward local inspection and hierarchical combination can be attained by specifying that the result-producing branch of the program will be capable of moving the turtle but only capable of sensing the single pixel where the turtle is currently located. The result-producing branch will not have direct access to the any other pixel sensors; however, it may call on the detectors. The result-producing branch will contain logical predicates consisting of compositions of Boolean conjunctions, disjunctions, and negations operating on the values returned by the detectors.

The function-defining branches define detectors that examine the entire nine-pixel local neighborhood of the turtle and evaluate logical predicates involving what the turtle sees. Thus, the function definitions will be compositions of the Boolean conjunctions, disjunctions, and negations and the nine pixel sensors (`X`, `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, and `NW`). For example, the function definition

```
(defun vertical-line ()
  (values (AND N X S (NOT SW) (NOT W) (NOT NW)
               (NOT SE) (NOT E) (NOT NE))))
```

is a detector that returns `T` only if the turtle is currently located at the midpoint of a vertical line segment consisting of three ON pixels surrounded on both sides by three OFF pixels. Note that there are no dummy variables in this `defun` since the nine pixel sensors are global variables established by virtue of the turtle's current location on the grid.

III Preparatory Steps for Using Genetic Programming

In what follows, we apply genetic programming with automatic function definition to the problem of letter recognition described above.

We decided that each individual overall S-expression in the population will consist of five function-defining branches (defining detectors called `ADF0` through `ADF4`) and a final (rightmost) result-producing branch.

Genetic programming will evolve function definitions in the five function-defining branches of each overall S-expression and then, at its discretion, it will call some, all, or

none of the defined functions (detectors) in the result-producing branch. The structures of both the function-defining branches and the result-producing branch are determined by the combined effect, over many generations, of the selective pressure exerted by the convolution-based fitness measure and by the effects of the operations of Darwinian fitness proportionate reproduction and crossover.

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

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

The terminal sets and function sets are different for the function-defining branches and the result-producing branch of each individual S-expression in the population.

We first consider the five function-defining branches (i.e., the detectors).

Since the function-defining branches are to define detectors which are capable of analyzing what the turtle sees at its current location on the grid, the terminal set $\mathcal{T}_{fd}$ for each function-defining branch consists of the nine pixel sensors, so that

$$\mathcal{T}_{fd} = \{X, N, NE, E, SE, S, SW, W, NW\}.$$

The function set $\mathcal{F}_{fd}$ for the function-defining branch is

$$\mathcal{F}_{fd} = \{AND, OR, NOT\},$$

taking 2, 2, and 1 argument, respectively. Note that there are no side-effecting functions in the function-defining branches.

Each function-defining branch is a composition of primitive functions from the function set $\mathcal{F}_{fd}$ and terminals from the terminal set $\mathcal{T}_{fd}$.

We now consider the result-producing branch.

We envision that the result-producing branch of each program will be a decision tree which consists of compositions of decision-making functions to return the identification I, L, or NIL. Thus, the terminal set $\mathcal{T}_{rp}$ for the result-producing branch is

$$\mathcal{T}_{rp} = \{I, L, NIL\}.$$

Since the result-producing branch is to be a decision tree, its function set will include the three-argument conditional decision-making "IF-THEN-ELSE" operator. This operator is implemented as a macro as described in Koza [1992a].

The function set of the result-producing branch will also contain eight operators that can move the turtle one step in any one of the eight possible directions from its current location. For example, the side-effecting operator (GO-N) would move the turtle north (up) one step in the 6 by 4 grid. As the turtle moves, the values of the nine pixel sensors (X, N, NE, E, SE, S, SW, W, and NW) are dynamically redefined. For simplicity, the grid is toroidal. These eight operators take no arguments. Each operator returns the value (T or NIL) of the pixel to which the turtle moves (i.e., it returns the new X).

The one-argument HOMING operator evaluates its argument and, in addition, has the side effect of rubber-banding the turtle to its position at the start of the evaluation of the HOMING. HOMING is equivalent to the brackets in a Lindenmayer system.

The Boolean functions AND, OR, and NOT are included in the function set to enable the result-producing branch to define logical predicates.

Finally, the five automatically defined functions (ADF0 through ADF0) that constitute the detectors are included in the function set of the result-producing branch. These five defined functions take no arguments.

Thus, the function set $\mathcal{F}_{rp}$ for the result-producing branch is

$$\mathcal{F}_{rp} = \{IF, AND, OR, NOT, HOMING, ADF0, ADF1, ADF2, ADF3, ADF4, GO\text{-}N, GO\text{-}NE, GO\text{-}E, GO\text{-}SE, GO\text{-}S, GO\text{-}SW, GO\text{-}W, GO\text{-}NE\},$$

The first five functions above take 3, 2, 2, 1, and 1 argument, respectively, and the remaining functions take no arguments (and could, optionally, if desired, have been treated as terminals).

The result-producing branch is a composition of primitive functions from the function set $\mathcal{F}_{rp}$ and terminals from the terminal set $\mathcal{T}_{rp}$.

Note that the above terminal sets and function sets satisfy the closure property in that each primitive function in the function set is well defined for any combination of arguments from the range of values returned by every primitive function that it may encounter and the value of every terminal that it may encounter.

Since each individual S-expression in the population consists of five function-defining branches and one result-producing branch, we must create the initial random generation so that every individual S-expression in the population has this particular constrained syntactic structure. Specifically, every individual S-expression must have the invariant structure represented by the six points of types 1 through 6 described in Section VI of Koza, Keane and Rice [1993]. Each of the five function-defining branches is a random composition of functions from the function set $\mathcal{F}_{fd}$ and terminals from the terminal set $\mathcal{T}_{fd}$. Each function and terminal in each function-defining branch is of type 7. The result-producing branch is a random composition of functions from the function set $\mathcal{F}_{rp}$ and terminals from the terminal set $\mathcal{T}_{rp}$. Each function and terminal in the result-producing branch is of type 8.

Moreover, since a constrained syntactic structure is involved, we must perform crossover so as to preserve the syntactic validity of all offspring as the run proceeds from generation to generation. Since each S-expression must have the invariant structure represented by the six points of types 1 through 6, crossover is limited to points of types 7 and 8. Structure-preserving crossover is implemented by allowing any point of type 7 or 8 in any branch of the overall S-expression to be the crossover point in the first parent.

However, once the crossover point in the first parent has been selected, the crossover point of the second parent must be of the same type. In practice, this means that crossover will only exchange a sub-tree from a function-defining branch with a sub-tree from another function-defining branch or that crossover will exchange a sub-tree from a result-producing branch with a sub-tree from another result-producing branch. This restriction on the selection of the crossover point of the second parent ensures the syntactic validity of the offspring.

Because of the complexity of the solutions evolved by genetic programming for this problem, we discovered, after initially solving this problem, that imposing an additional constrained syntactic structure on the result-producing branch would greatly enhance our ability to understand the solution. Specifically, we constrained the first (antecedent) argument of each IF operator to be a composition of the three Boolean functions (`AND`, `OR`, `NOT`), the five automatically defined functions (`ADF0` through `ADF4`), the eight turtle-moving operators (e.g., `GO-N`), and the `HOMING` function. In addition, we constrained the two consequent arguments of each IF operator to be compositions of the IF operator and the terminals (I, L, and `NIL`). These additional syntactic contraints had the helpful effect of clearly isolating and highlighting the structure of the decision tree within the result-producing branch. Thus, the original type 8 was subdivided into types 8a and 8b. As before, the initial random population was randomly generated in comformity with these new constraints and structure-preserving crossover was performed to preserve the new syntactic structure (now involving types 7, 8a, or 8b).

The third major step in preparing to use genetic programming is the identification of the fitness measure for evaluating the goodness of each pattern-recognizing individual in the population. The fitness cases for genetic programming are chosen to represent a sufficient variety of situations so that the program is likely to generalize to handle all possible combinations of inputs. In this regard, the fitness cases are similar to the necessarily small, finite number of combinations of inputs used to test and debug computer programs.

Each individual in the population is tested against an environment consisting of N_{fc} = 78 fitness cases, each consisting of a 6 by 4 pixel pattern and the correct identification (I, L, or `NIL`) for that pattern. The set of fitness cases included the two letters (i.e., the positive cases) and 76 different negative fitness cases. The negative cases included every version of the letters I and L with one ON pixel deleted; every version of the letters I and L with one extraneous ON pixel added adjacent to the correct pixels; checkerboard patterns; the all-ON and all-OFF pattern; various patterns bearing some resemblance to I, L, or other letters; and various random patterns bearing no resemblance to I, L, or other letters.

When an individual S-expression in the population is tested against a particular fitness case, the result can be a true-positive (i.e., the individual correctly identifies an I as an I or an L as an L), a true-negative (i.e., a pattern that is not an I or L is correctly identified as `NIL`), a false-positive (i.e., a non-letter is identified as a letter), a false-negative (i.e., a letter is identified as a non-letter), or a wrong-positive (i.e., an I is identified as an L or an L is identified as an I).

For this problem, fitness is the sum, over the fitness cases, of the weighted errors produced by the S-expression. The smaller the sum of weighted errors, the better. A 100% correct pattern-recognizer would have a fitness of zero. True-postives and true-negatives contribute zero to this sum. False-positives contribute 38 (in order to give the two positive cases equal weight with the 76 negative cases). Wrong-positives contribute one. False-negatives contribute 23 (i.e., the number of pixels minus one) to maintain consistency with work not described herein involving translation-invariant pattern recognition and cellular automata.

The fourth major step in preparing to use genetic programming is the selection of values for certain parameters. Our choice of 8,000 as the population size and our choice of 101 as the maximum number of generations to be run reflect an estimate on our part as to the likely difficulty of this problem and the practical limitations on available computer time and memory. Our choice of values for the various secondary parameters that control a run of genetic programming are the same default values as we have consistently used on numerous other problems [Koza 1992a], except that we continue our recently adopted practice of using tournament selection (with a group size of seven) as the selection method.

Finally, the fifth major step in preparing to use genetic programming is the selection of the criterion for terminating a run and the selection of the method for designating a result. We will terminate a given run if we encounter a 100% correct individual or after 101 generations. We designate the best individual obtained during the run (the best-so-far individual) as the result of the run.

IV Results for One Run

A review of one particular successful run will serve to illustrate how genetic programming simultaneously evolves the detectors and a way of combining the detectors for the problem of letter recognition.

A Discovery of Solution for One Run

The 8,000 randomly generated individuals found in the initial generation of the population (generation 0) are, as one would expect, not very good. The fitness of the worst individual pattern-recognizer in the population for generation 0 has 44 points (i.e., functions and terminals in the body of its five function-defining branches and its result-returning branch) and has the enormous unfavorable fitness (error) of 3,535. However, even in a randomly created population of programs, some individuals are better than others. For example, an individual at the 33rd percentile of generation 0 has fitness of 1,771.

The best individual from generation 0 has 186 points, has a fitness of 53, and is shown below:

```
(PROGN (DEFUN ADF0 ()
         (VALUES (OR (OR (AND (OR S X) (OR N
         SW)) (OR (OR N E) (AND S NW))) (AND
         (OR (NOT S) (AND S SE)) (OR (AND W
         SE) (NOT N)))))))
       (DEFUN ADF1 ()
         (VALUES (AND (AND (NOT (NOT X)) (NOT
         (OR S X))) (OR (AND (AND SW NW) (NOT
         SW)) (NOT (AND N SW))))))
       (DEFUN ADF2 ()
         (VALUES (OR (AND (NOT (AND W E)) (OR
         (AND NW W) (NOT NW))) (OR (OR (AND N
         E) (AND S SE)) (OR (AND W SE) (OR SE
         NE))))))
       (DEFUN ADF3 ()
         (VALUES (AND (NOT (AND (NOT SE) (OR W
         SW))) (OR (NOT (OR NW NE)) (AND (NOT
         S) (NOT NW))))))
       (DEFUN ADF4 ()
         (VALUES (AND (NOT (OR (OR W SW) (OR
         NW NW))) (AND (AND (AND X N) (NOT
         NE)) (OR (OR N SE) (OR X E))))))
       (VALUES
         (IF (OR (NOT (AND (GO-S) (GO-S)))
         (AND (NOT (ADF0)) (HOMING (GO-N))))
         (IF (HOMING (AND (GO-S) (ADF0))) (IF
         (OR (GO-N) (ADF2)) (IF (ADF1) L I)
         (IF (ADF1) I NIL)) (IF (HOMING (GO-
         S)) (IF (ADF1) L I) (IF (GO-W) L
         NIL))) (IF (OR (OR (GO-E) (ADF3))
         (AND (ADF3) (ADF3))) (IF (HOMING (GO-
         N)) (IF (ADF3) L NIL) (IF (GO-S) NIL
         L)) (IF (NOT (ADF1)) (IF (GO-E) NIL
         I) (IF (ADF1) L L))))))).
```

The result-producing branch of this best-of-generation individual makes multiple references to four of its five detectors, contains numerous movements in various directions of the turtle, contains numerous Boolean functions, contains numerous IF operators, and is capable of arriving at all three possible conclusions for a given pattern. The detectors (defined functions) each look at numerous pixels and performs various logical functions on what is seen.

Fig. 3 shows, by generation, the fitness of the best-of-generation individual. As can be seen, the average fitness of the population as a whole tends to improve (i.e., drop) from generation to generation. Specifically, the fitness of the best-of-generation individual drops to 42, 31, 22, 14, 10, 7, 6, 5, and 3 for generations 5, 10, 15, 20, 25, 30, 35, 40, and 45, respectively. The fitness of the best-of-generation individual drops to 2 for generations 47, 48, and 49.

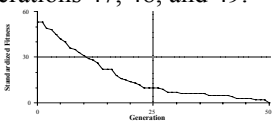

Fig. 3 Fitness curves

The hits histogram is a useful monitoring tool for visualizing the progressive learning of the population as a whole during a run. The horizontal axis of the hits histogram represents the number of hits (0 to 78) while the vertical axis represents the number of individuals in the population (0 to 8,000) scoring that number of hits.

Fig. 4 shows the hits histograms for generations 0, 15, 30, and 50 of this run. Notice the left-to-right undulating movement of both the high point and the center of mass of these three histograms. This "slinky" movement reflects the improvement of the population as a whole. Four solutions to the problem emerge at generation 50.

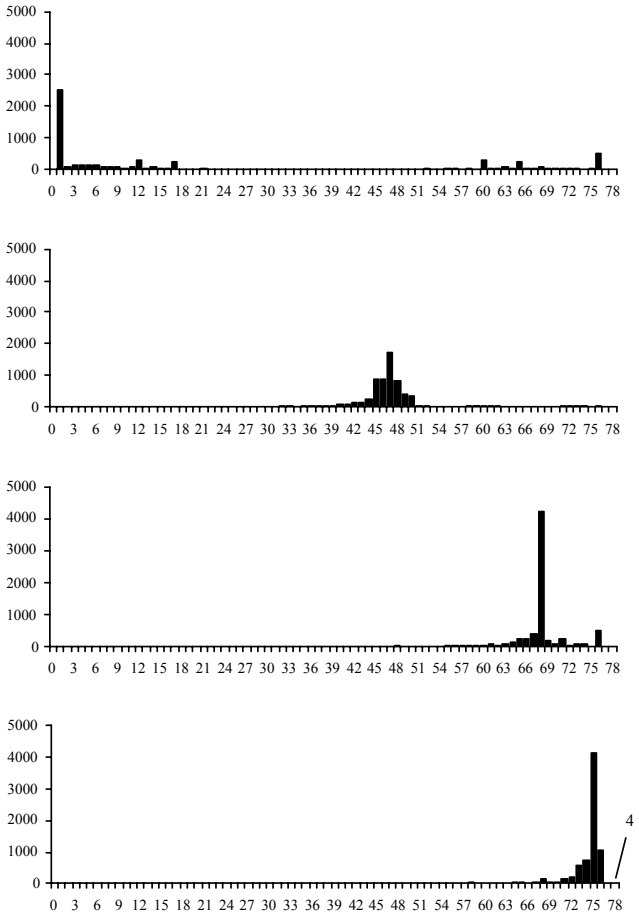

Fig. 4 Hits histograms for generations 0, 15, 30, and 50

By generation 50, the best-of-generation individual has 312 points (of which 149 are in the result-producing branch) and has the perfect fitness value of zero. This best-of-run individual is shown below:

```
(PROGN (DEFUN ADF0 ()
         (VALUES (OR (OR (AND W SE) (OR (AND
         (NOT (OR SW SW)) (NOT (AND X SW)))
         (NOT (OR (NOT S) (AND X NW))))) (AND
         (OR (NOT S) (AND W SE)) (OR (OR S X)
         (NOT N))))))
       (DEFUN ADF1 ()
         (VALUES (AND (AND (NOT (NOT X)) (NOT
         (OR S X))) (NOT (OR S X)))))
       (DEFUN ADF2 ()
         (VALUES (OR (AND (NOT (AND W E)) (OR
         (AND NW W) (NOT NW))) (OR (OR (AND N
         E) (AND S SE)) (OR (AND W (NOT NW))
         (OR SE NE))))))
       (DEFUN ADF3 ()
         (VALUES (AND (NOT (AND (NOT SE) (OR W
         SW))) (OR (NOT (OR NW (NOT NW))) (AND
         (NOT S) (AND (NOT (AND (NOT SE) (OR W
         SW))) (OR (NOT (OR NW (NOT (AND (NOT
```

```
  SE) (OR W SW))))) (AND (NOT S) (OR
  (NOT (NOT NW)) (NOT SE))))))))))
(DEFUN ADF4 ()
  (VALUES (AND (NOT (OR (OR W SW) (OR
  NW NW))) (AND (AND (AND X N) (NOT
  NE)) (AND (NOT (OR (OR E SW) (OR NW
  NW))) (AND (AND (AND (AND X N) (NOT
  NE)) (NOT NE)) (OR (OR N SE) (OR X
  E))))))))
(VALUES
  (IF (OR (NOT (ADF4)) (AND (OR (NOT
  (AND (GO-S) (GO-S))) (AND (OR (NOT
  (AND (GO-S) (GO-S))) (AND (NOT (AND
  (ADF3) (ADF3))) (HOMING (GO-S)))) (OR
  (NOT (AND (GO-S) (GO-S))) (AND
  (HOMING (GO-N)) (HOMING (GO-N))))))
  (OR (NOT (ADF4)) (AND (OR (NOT (AND
  (GO-S) (GO-S))) (AND (OR (NOT (AND
  (GO-S) (GO-S))) (AND (NOT (AND (ADF3)
  (ADF3))) (HOMING (GO-N)))) (OR (NOT
  (AND (GO-S) (GO-S))) (AND (HOMING
  (GO-N)) (HOMING (GO-N))))))) (OR (NOT
  (AND (GO-S) (GO-S))) (AND (NOT (AND
  (GO-S) (ADF3))) (HOMING (GO-
  N)))))))); antecedent of outermost IF
      (IF (HOMING (AND (GO-S) (ADF0)))
  (IF (GO-S) NIL L) (IF (HOMING (GO-S))
  (IF (ADF1) L I) (IF (ADF1) L NIL)));
  then-part of outermost IF
      (IF (OR (OR (GO-E) (ADF3)) (AND
  (OR (NOT (ADF4)) (AND (NOT (ADF3))
  (OR (NOT (AND (GO-S) (GO-S))) (AND
  (NOT (GO-S)) (AND (ADF3) (ADF3))))))
  (HOMING (GO-N)))) (IF (ADF2) (IF (GO-
  S) L NIL) (IF (GO-S) (IF (ADF3) L
  NIL) (IF (GO-S) NIL L))) (IF (NOT
  (ADF1)) (IF (GO-E) NIL I) (IF (ADF1)
  L L))); else-part of outwemost IF
  ))).
```

B. Analysis of the Solution for One Run

The performance of the 100% correct best-of-run individual from generation 50 from the run described above can be understood by first considering how this program successfully recognizes the pixel pattern for the letter L. To aid in this process, Fig. 5 identifies the seven pixels that must be ON for the pattern to be the letter L with the Roman numerals I through VII and identifies the 14 adjacent pixels that must be OFF for the letter L with the lower-case letters a through n.

a	I	n	
b	II	m	
c	III	l	
d	IV	k	j
e	V	VI	VII
f	g	h	i

Fig. 5 The seven ON and 14 OFF pixels constituting the letter L

When the best-of-run individual from generation 50 encounters an L, it moves the turtle 25 times. The turtle always starts at pixel III for any pattern. Fig. 6 shows the 25 steps in this trajectory, with the turtle starting at pixel III and ending at pixel n. The figure omits certain numbers where the turtle repetitively moves back and forth over the same two adjacent pixels.

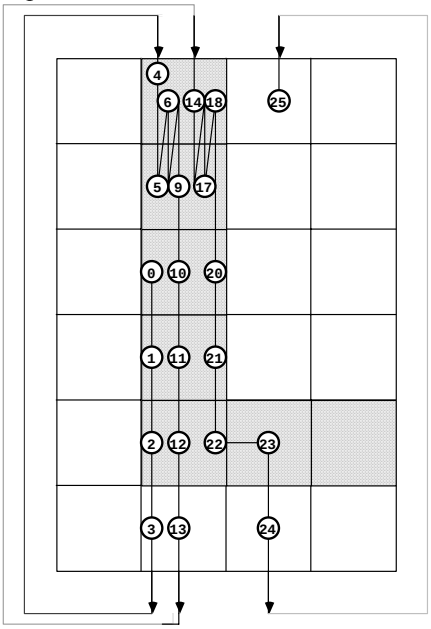

Fig. 6 Trajectory of the turtle for identifying an L for best-of-run individual from generation 50

Since the result-producing branch begins with `(IF (OR (NOT (ADF4))` ..., the defined function `ADF4` is evaluated with the turtle being at its starting location (pixel III) at turtle step 0. Detector `ADF4` examines seven of the nine pixels within view.

Fig. 7 shows the seven pixels values required to cause the function definition for `ADF4` to return a value of `T`. `ADF4` depends on seven pixels, lacks any reference to pixel `S`, and effectively ignores pixel `SE`.

Off	On	**Off**
Off	On	**Off**
Off		

Fig. 7 Arrangement of pixels required to cause ADF4 to return `T`

When the turtle is located at pixel III, pixels II and III are ON, and pixels b, c, d, m, and l are OFF, so `ADF4` returns `T`. (see Fig. 8) These latter five pixels all lie adjacent to the vertical segment of the L. Thus, when the turtle is at pixel III, `ADF4` acts as a detector for two of the three vertically stacked pixels of a potential L being ON and as a detector for five of the six pixels adjacent to the potential L being OFF. `ADF4` is an incomplete detector for a vertical line segment.

Since `ADF4` returns `T` when the turtle is located at pixel III, `(NOT (ADF4))` is `NIL`, so the second clause of the first `OR` must be evaluated. This second clause begins with `(AND (OR (NOT (AND (GO-S) (GO-S)))`... . When the first argument of the inner `AND`, namely `(GO-S)`, is evaluated, the turtle moves south (down the vertical segment of the L) from pixel III to pixel IV. Since pixel IV is ON after this turtle step 1, the `(GO-S)` operator returns `T`, thus

necessitating evaluation of the second argument of this inner AND. This second argument, which consists of another (GO-S) operator, moves the turtle south again from pixel IV to pixel V. Since pixel V is also ON after turtle step 2, the inner AND returns T. However, the NOT necessitates evaluation of the second argument of the OR, namely

```
(AND (OR (NOT (AND (GO-S) ... )) ... ))
```

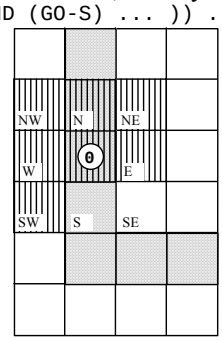

Fig. 8 Detector ADF4 applied at pixel III at turtle step 0

The (GO-S) operator (the first argument to the inner AND above) now moves the turtle south to pixel g. Pixel g is OFF at turtle step 3, since it is below the vertical segment of the L. Note that when LISP evaluates the two-argument Boolean AND function, it skips evaluation of the second argument if the first argument evaluates to NIL. (Similarly, evaluation of the second argument of the OR function is skipped if the first argument evaluates to T). The second (GO-S) argument to the AND is replaced with an ellipsis, since it will not be evaluated. This illustrates that when side-effecting operators are contained in arguments to Booolean functions, the operators are conditionally executed in LISP depending on the context. Since the NOT negates the NIL returned by the AND, the second argument to the OR containing seven points is also skipped and replaced with a second ellipsis.

Now two (GO-S) operators move the turtle to pixels I and II (since the grid is toroidal) at turtle steps 4 and 5. The (GO-N) moves the turtle back to pixel I at turtle step 6, but the HOMING rubber bands the turtle back to pixel II at turtle step 7. This sequence is repeated at turtle steps 8 and 9, leaving the turtle at pixel II.

Detector ADF4 is now applied at pixel II. Fig. 9 shows that when the turtle is located at pixel II, ADF4 returns T when pixels I and II are ON and when pixels a, b, c, n, and m are OFF.

Detector ADF4 returns T and thereby provides the new information that pixels a and n are OFF. Thus, by turtle step 9, seven pixels (a, b, c, d, n, m, and l) have been verified as being OFF via two applications of detector ADF4 and one additional pixel (g) has been verified as being OFF via the (GO-S) operator. In addition, five pixels (I, II, III, IV, and V) have been verified (often repetitively) as being ON. Several pixels have been verfied by more than one action by turtle step 9.

Between turtle steps 10 and 21, the turtle repetitively moves up and down (via several HOMINGs) along the vertical segment of the L, but provides no information that is not already known.

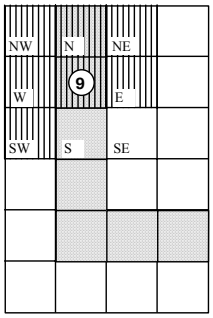

Fig. 9 Detector ADF4 applied at pixel II at turtle step 9

The turtle arrives at pixel IV at turtle step 21 and begins evaluation of the last six points of the antecedent clause of the outermost IF of the value-returning branch, namely

```
(AND (NOT (AND (GO-S) (ADF3)))
     (HOMING (GO-N))))))))).
```

The (GO-S) operator moves the turtle to pixel V (the junction of the L, which is ON) and executes detector ADF3.

Fig. 10 shows the pixels values required to cause ADF3 to return a value of T. As can be seen, ADF3 examines five pixels (NW, W, SW, S, and SE) to see if they are all OFF.

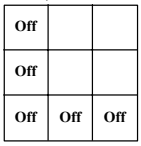

Fig. 10 Arrangement of pixels required to cause ADF3 to return a value of T

Fig. 11 shows that when the turtle is located at pixel V (the junction of the L), ADF3 returns T when pixels d, e, f, g, and h are OFF. These five pixels all lie adjacent to the junction of the L, so ADF3 acts as a detector for emptiness adjacent to the junction of an L.

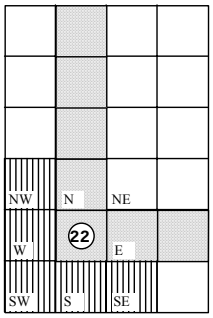

Fig. 11 Detector ADF3 applied at pixel V at turtle step 22

Since the NOT negates the return value of the AND, the (HOMING (GO-N)) is not executed.

For the letter L, the antecedent part of the outermost IF of the value-returning branch of the best-of-run individual of generation 50 evaluates to NIL, so that the second (then) argument of the IF is skipped and the third (else) argument is executed. The (GO-E) operator moves the turtle east to pixel VI (which is ON) and causes detector ADF2 to be evaluated for turtle step 23.

Fig. 12 shows that ADF2 examines six pixels. ADF2 returns a value of NIL when NW, W, and E are ON and when N, NE, and SE are OFF.

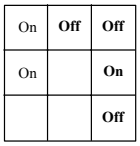

Fig. 12 Arrangement of pixels required to cause ADF2 to return a value of NIL

Fig. 13 shows that when the turtle is located at pixel VI at turtle step 23, ADF2 returns a value of NIL when pixels IV, V, and VII are ON and when pixels i, j, and k are OFF.

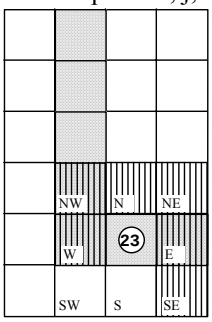

Fig. 13 Detector ADF2 applied at pixel VI at turtle step 23

The result now depends on the expression below involving detector ADF2:

```
1   (IF (ADF2)
2       (IF (GO-S) L NIL)
3       (IF (GO-S)
4           (IF (ADF3) L NIL)
5           (IF (GO-S) NIL L)) ... ).
```

For the letter L, ADF2 evaluates to NIL on line 1. As a result, line 2 is skipped and the (GO-S) operator on line 3 moves the turtle south to pixel h at turtle step 24. Since pixel h is OFF, line 4 is skipped and the (GO-S) operator on line 5 moves the turtle toroidally to pixel n for turtle step 25. Since pixel n is OFF for the L, the result-producing branch and the S-expression as a whole therefore returns L, which is indeed the correct identification of the pattern.

Notice that if detector ADF2 returns T on line 1 above, this would mean that either pixel i, j or k was ON or that pixel VII was OFF (since pixels IV and V have been previously established as being ON). Any of these four possibilities would mean that the pattern is a flawed pattern for which NIL (rather than L or I) should be returned. For example, if pixel VII were OFF and pixel i were ON (as shown in the left half of Fig. 2) or if pixel VII were ON and pixel i were ON, the pattern would be a flawed L. In these situations, the (GO-S) operator on line 2 would be executed, thereby moving the turtle to pixel h. Since pixel h is already known to be OFF, the result-producing branch and the S-expression as a whole would return NIL, which, under the circumstances, would be correct identification of the pattern. Similarly, the result-producing branch and the S-expression as a whole returns the value NIL for the 14 fitness cases for which there is an extraneous ON pixel adjacent to an L (in locations a through n) and the seven fitness cases for which there is a missing pixel within an L (in locations I through VII).

In summary, the best-of-run individual from generation 50 applies detectors at turtle steps 0, 9, 22, and 23 and considers direct input from the turtle over 25 steps in order to determine that all seven pixels (I through IV) that should be ON for an L are indeed ON and that all 14 pixels that should be OFF for an L are indeed OFF, as shown in Fig. 14.

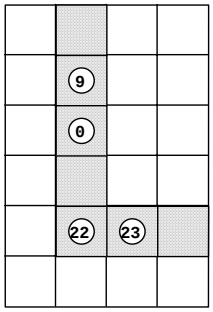

Fig. 14 Turtle steps 0, 9, 22, and 23 where detectors ADF2, ADF3, and ADF3 are applied

In identifying the letter I, the turtle moves up and down the vertical column consisting of pixels I through V and pixel g for the first 22 turtle steps much as in the identification of the L. However, when the turtle moves east on turtle step 23, pixel VI is OFF for the I.

Notice that detector ADF0 is used for determining that certain patterns should be classified as NIL, although it not used in classifying the positive cases. Also, notice that although ADF1 is merely the constant function NIL, it does appear, and is used, in the result-producing branch.

V Conclusions

We have demonstrated the use of genetic programming with automatic function definition to simultaneously discover the detectors and a way of ssing the detectors to perform a letter recognition task.

ACKNOWLEDGEMENTS

James P. Rice of the Knowledge Systems Laboratory at Stanford University programmed the above on a Texas Instrument Explorer II$^+$ computer. Martin Keane and Simon Handley made helpful comments on this paper.

REFERENCES

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

Koza, John R. Hierarchical automatic function definition in genetic programming. In Whitley, Darrell (editor). *Proceedings of Workshop on the Foundations of Genetic Algorithms and Classifier Systems*, *Vail, Colorado 1992.* San Mateo, CA: Morgan Kaufmann Publishers Inc. 1992. 1992b.

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

Koza, John R., Martin A. Keane, and Rice, James P. Performance improvement of machine learning via automatic discovery of facilitating functions as applied to a problem of symbolic system identification. In these ICNN-93 Conference Proceedings. 1993.

Samuel, Arthur L. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development,* 3(3): 210–229. July 1959.