Evolving Visual Routines

Michael Patrick Johnson, Pattie Maes and **Trevor Darrell**
MIT Media Laboratory
Cambridge, Mass.
aries/pattie/trevor@media.mit.edu

Abstract

Traditional machine vision assumes that the vision system recovers a a complete, labeled description of the world [Marr, 1982]. Recently, several researchers have criticized this model and proposed an alternative model which considers perception as a distributed collection of task-specific, task-driven visual routines [Aloimonos, 1993, Ullman, 1987]. Some of these researchers have argued that in natural living systems these visual routines are the product of natural selection [Ramachandran, 1985]. So far, researchers have hand-coded task-specific visual routines for actual implementations (e.g. [Chapman, 1993]). In this paper we propose an alternative approach in which visual routines for simple tasks are evolved using an artificial evolution approach. We present results from a series of runs on actual camera images, in which simple routines were evolved using Genetic Programming techniques [Koza, 1992]. The results obtained are promising: the evolved routines are able to correctly classify up to 93% of the images, which is better than the best algorithm we were able to write by hand.

1 Introduction

One of the hardest problems to be solved when building a real robotic autonomous agent is the perception problem. Traditional machine vision assumes that the vision system produces a labeled, perfectly resolved model of the world, distinguishing and representing all objects [Marr, 1982]. This turns out to be an extremely hard, if not impossible, problem to solve.

Active vision is a new paradigm for doing machine vision that has received a lot of attention recently [Ballard, 1989, Aloimonos, 1993]. The insight that active vision brings to the problem is that almost all of the time an intelligent agent doesn't even *need* all that information, since it is involved in a particular task that only requires specific knowledge of certain objects. Several researchers have proposed ways of allowing the central control system to actively control what is processed by the vision system in a *task-dependent* manner (e.g. [Ballard and Whitehead, 1990]). For example, if I want to pick up the coffee cup on my desk, I don't need to process and represent all the paper clips, papers, every character on every paper, and all the rest of the clutter usually on a desk. I just want information on my hand, the cup and anything that might be in the way of my hand and the cup. Furthermore, that information will be different information than I might extract if I were intent on drawing the cup. Thus, active vision requires more work on the part of the central system.

To explain how the internal active processing could be implemented, Shimon Ullman proposes the *visual routines* model [Ullman, 1987], which describes a set of primitive routines that can be applied to an input image in order to find certain spatial relations between objects as well as other useful information. He suggests that given a specific task by the central system, the visual routines processor compiles an appropriate visual routine and applies it to a base representation, perhaps changing it in some way or returning a result. It may be asked to process something else in response to that answer, and so on. Ullman does not go into detail on how routines are developed in the first place, how they are stored or how they are chosen and applied. An obvious answer might be that the central system does some intelligent reasoning in order to determine their application.

Ramachandran, a human vision researcher, disagrees with this solution and suggests the "utilitarian theory" of perception [Ramachandran, 1985]. He argues that since many other systems in the human body are collections of *ad hoc* pieces that all function in their own way but tend to work together, perception is likely to be the same. He argues that this is the manner in which natural selection works — anything that works will be used whenever it works. Hence, he concludes, perception is like a "bag of tricks" selected by evolution to solve various perceptual subproblems. Whichever trick has proven to work for a particular problem will be the one used for that problem in the future.

In creating a computer vision system, we have at our disposal a powerful alternative — the programmer. This person can design by hand visual routines that solve a given task, test them and refine them, hopefully reaching an optimal solution. This may be a reasonable approach in limited cases, such as the Sonja system by David Chapman [Chapman, 1993], which has a set of routines designed by Chapman for solving different visual tasks involved in playing a particular video game. The set of routines is fairly small and fixed, allowing a

programmer to work them out by hand.

Another similar example (and the one which we will focus on) is the simple vision system used by the ALIVE (Artificial Life Interactive Video Environment) virtual environment project [Maes *et al.*, 1994, Maes *et al.*, 1993]. In this system, a user can interact in real-time with a computer graphics creature using gestures which are interpreted by a vision system. This system employed a set of hand-coded low-level heuristics for solving specific simple visual tasks involved in efficiently processing camera input of a person interacting with the system. These tasks included perception problems such as "Find the hands" and "Is the person sitting?" and "Is the person pointing?".

The problem with this method is that it ties up a lot of programmer-hours fiddling with knobs and parameters and conditions, trying to get something that works well all the time. If a new problem is approached, a new set of routines has to be created. Also, it is difficult for the programmer to anticipate all cases in which the program should work.

This paper describes a way of automatically producing a visual routine appropriate for a given task by using simulated evolution of computer programs in the style of John Koza [Koza, 1992]. The paper is organized as follows. Section 2 discusses previous work on visual routines. Section 3 describes the particular perception task that we concentrated on, namely finding the left and right hand of a person in a black and white silhouette of the person, which is extracted from a real camera image. Section 4 discusses the actual Genetic Programming (GP) implementation used for the experiments. Section 5 discusses results of the GP runs. Section 6 lists possible future directions. Finally, section 7 gives a summary.

2 Visual Routines

Ullman [Ullman, 1987] breaks the visual system into three main areas: the base representation, the visual routines processor and the higher level components. The base representation is the result of initial, parallel processing of the retinal image. It is bottom-up and uniform in the sense that the same processing occurs across the whole image (which he calls spatial parallelism). This could involve resolution edge detection, color processing, etc. The visual routines processor performs the tasks that require a more directed, specific or inherently sequential set of processes. The higher level components consist of recognition memory and task formulation.

Ullman makes the distinction between universal routines and specific routines. Universal routines are routines that are processed automatically in order to form a basis for deciding which specific routines must be used. In the absence of a specific task, universal routines operate. They might, he suggests, isolate prominent areas of the image, do simple spatial relations tasks, and perform crude color and shape characterizations. An example might be the process that leads to a saccade toward a pop-out area of the visual field; that is, an area that is significantly different from its surroundings due to motion, color, orientation or depth.

Finally, Ullman proposes a set of specific routines which he admits is not comprehensive, but seem to be useful for certain problems.

- *Indexing* involves shifting the focus of attention to important areas of the visual field. For example, shifting focus to "pop-out" areas or in order to peform a visual search for specific objects.
- *Marking* allows the system to remember a location or describe areas of processing for other operations.
- *Ray intersection* involves tracing along a ray between two points (one may be "at infinity," or out of the critical processing area) and count or place markers at intersections with boundaries. This is useful for inside-outside decision tasks (for example, deciding if an indicated point lies inside or outside a closed contour).
- *Bounded Activation* or *coloring* fills in an area surrounding a marked point, stopping at a boundary. What constitutes a boundary is a function of the task, so must be a parameter to the operation. For example, consider again the inside-outside task. People can still tell if a point is inside a circle with a dashed line for a border (see [Ullman, 1987]). Thus, the boundary need not be solid.
- *Boundary tracing* is another sequential operation that traces a boundary (with the same parameter as above to define a boundary), searching for various facts, such as whether the curve is closed or not, or whether there are objects along it.

Other routines can be applied to the results of these computations. For example, a point could be marked and the area around it colored. Checking whether the color spread to some point known to be outside the curve (using a bounding box of the curve, say), will tell us whether the point was inside a closed contour or not.

David Chapman applied similar operations to the domain of visual attention in his Sonja video game playing system [Chapman, 1993]. Sonja can find gaps in a boundary (ostensibly doors), track moving pop-out objects (such as the player's icon) with a marker, color a region and cast rays, for instance[1]. In the game, characters move around a two-dimensional world full of impenetrable walls. To move to a certain marked point (say there is something desirable there) a ray is cast from the current location to the desired location to detect any intersections with walls. If an intersection point is found, a path around the obstacle is constructed. Chapman hand-coded his visual routines to solve these problems using a circuit specification language. A large amount of programming effort was required to get these to work

[1] However, Chapman does not provide a model of "low-level" vision for Sonja; his primitives are actually implemented by directly referencing the models which underly the screen representation. This is appropriate for a closed video game system, but ignores issues such as how the primitives will perform in noisy and/or unexpected situations that occur in real-world environments.

correctly. An automatic technique for finding visual routines is thus quite desireable.

This is exactly the problem which we focus on in this paper. Inspired by the way in which nature may have evolved a range of special-purpose visual routines [Ramachandran, 1985], we try to automatically develop visual routines for particular tasks that are relevant to a system using artificial evolution. This work is related to the research described in [Cliff D. T. and I., 1993], which also evolves visually-guided behaviors. However, in contrast to that research, the work presented here uses actual camera images, rather than simulated ones.

3 The Problem — Find the Hands

The simple image processing application we focus on for the preliminary study is that of finding the hands in the bitmap silhouette of a person. A solution to this problem was actually necessary for the ALIVE virtual environment project [Maes *et al.*, 1994] (see Section 3.1). For ALIVE, a visual routine was written by hand for this task, applied when hand location was desired. We restrict the problem slightly for this study, to see if genetic programming can solve subgoals of the overall task. Specifically, we ask that the genetic programming system evolve a program for finding the "left" or "right" hand in the silhouette when the hand is below or at the shoulder line and the person is facing mostly toward the camera. The "left" hand is defined to be the one on the left side of the bitmap, since we don't have enough information to be sure the person isn't facing the wrong way. We define the "right" hand in like manner. Furthermore, our evolved solution need not find both hands at once — we evolved programs for finding the left hand and right hand separately. It is probably equivalent to learning to find just one side and use a mirrored image to find the other, but by doing them separate we can try to get around problems such as noise caused by inconsistent lighting. The evolution for one side might find a different solution better for the hardware than using the mirrored version of the other.

3.1 Fitness Case Generation

The ALIVE system allows a user to interact via gestures with a computer graphics behavior-based agent. The user's video image (taken from a video camera) is *superimposed* on the real-time graphical world. The composite image is then displayed on a life-size projection screen in front of the user. The effect is as if the user were looking into a magical mirror — they see a mirror image of themselves (their live video image) superimposed on the graphics world. They interact with the creatures in the virtual world by gestures. For example, they can point and the creature will walk in the indicated direction. They can wave and it will return.

ALIVE uses single color background subtraction, a method used ubiquitously for special effects. The user stands in front of a monochromatic blue wall with a camera pointed at them. The algorithm then replaces anything of the shade of blue in the background with computer graphics output and leaves the rest as live video. The effect is that the user is superimposed on the graphics. Technically, it returns a binary image with bits on where the video should go and bits off where the graphics should go. This output is used directly as input for the vision system and is the source of our fitness case silhouettes (see Figure 1). We used a total of 46 fitness cases in the experiments described below (and used 46 mirrored silhouettes as evaluation test cases).

The fitness cases were produced from the larger, noisier original images produced by the vision system by finding the largest connected component and returning its bounding box and centroid. For our purpose, we presume some other universal routine has already performed this task for us. Notice that some of the cases are extremely difficult as part of the hand is in front of the body or not easily locatable. A person labelled the "correct" hand locations. For the difficult cases, a best guess was given as a location near the waist. It is probably not proper to ask the program to find the hands in these cases; indeed, the original ALIVE system will only look for hands when it is likely that they are visible in the silhouette.

4 Genetic Programming Implementation

Based on Koza's original Genetic Programming (GP) algorithm, we implemented a typed GP system for solving the restricted hands problem. We refrain from a full discussion of GP here and give only salient points. Koza discusses this system and variations of it at great length in his book [Koza, 1992]. The system used here is similar to Koza's except for the addition of the type specification described below and several other small improvements.

The genetic operations used in this work are copying, crossover and mutation. Copying involves copying a certain percentage of individuals to the new generation roughly in proportion to their fitness (Koza calls this "fitness-proportionate reproduction"). Crossover chooses two individuals at random, again roughly proportional to fitness, and swaps randomly chosen subtrees. Mutation picks a random subtree and replaces it with a newly generated subtree. Better results were obtained using the tournament selection method described in [Koza, 1992], which picks k individuals at random and chooses the one with the best fitness (effectively a tournament between the individuals, hence tournament selection). This can be used for copying and for selecting the two individuals for crossover. Tournament selection seems to keep the system from converging prematurely due to one individual swamping the population.

4.1 The Typed Genetic Program

Our implementation is typed; the programmer must specify a type for each terminal and a *signature* for each function. A signature is comprised of a return type and a list of types for the arguments. Only forms consistent with this type specification will be allowed. This makes the space of unmeaningful programs smaller, as well as allowing the primitives to be more specialized by not having to deal with arbitrary typed input. This in turn speeds up fitness evaluation. Koza does have several examples of a typed system in his book, but they seem less general than the one used here.

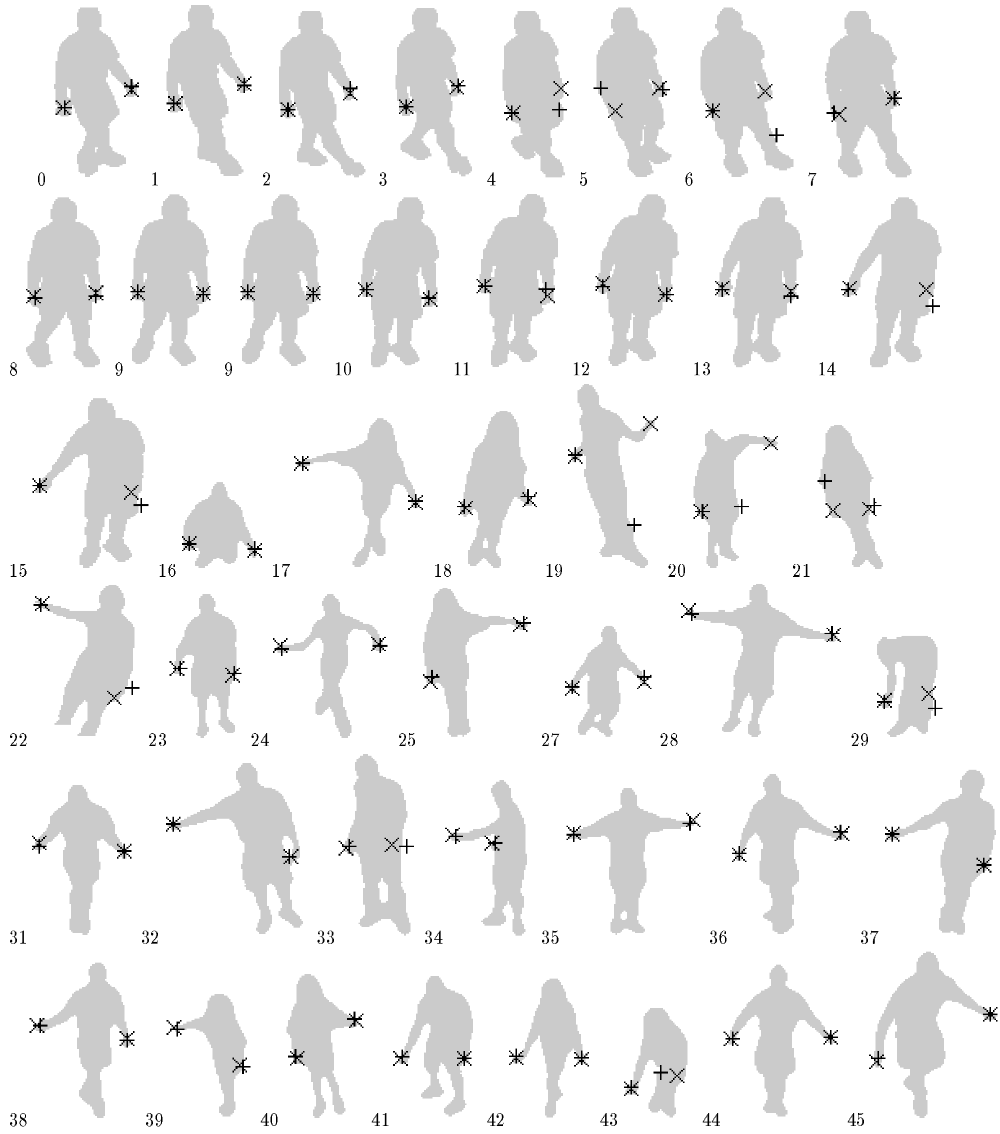

Figure 1: Fitness Cases. X shows the desired location. + shows the best program's result.

The programmer specifies the top level return type, eliminating the problems of forms returning incorrect data types. For example, in our experiments the top level type is a point. The GP system chooses randomly among the primitives of the appropriate type to fill in the argument slots recursively, starting at the root node. The maximum depth of the original programs is a parameter.

Types make crossover a little harder. One of the two mating programs is picked at random. Then we pick a random subtree in it and find the return type. We then choose randomly among the subtrees with appropriate root type in the other mate. If there are none, no crossover occurs and the individuals are copied. Also, if crossover will produce a program deeper than the maximum allowable depth of a program (another parameter) the crossover does not occur and the parents are copied.

Mutation chooses a node at random and replaces it with a randomly generated subtree of the same type as the original node. The depth of the created subtree is again limited by a parameter.

One drawback of the typed GP used is that overloading is not allowed — programs can have one and only one signature (return type and arg list) to make the type of a given primitive static. If it could have multiple types, when it is picked to do crossover, the system would have to find out what type it is returning in the given program. To do this it has to traverse the entire subtree. Thus, if a function can take two types of arguments, the programmer must make a unique function for each case. This was not a problem in our experiment.

A big advantage of the typed GP is that we can constrain the system in many useful ways. By having a function that returns a special type that exists no where else in the terminal and function set and specifying it as the root type, we can force the system to put that function at the root. This can be used to optimize the arguments of a particular top-level function. Another advantage is that we can create special terminal types that only certain functions use. If we want a function to take integers in a certain range, we make a set of terminal variables of a special type, say "smallint" and set them to constant integers. Thus, if larger integers are used elsewhere, the protected function won't get them. This aspect of the typed system was invaluable in making the *percent* terminal, described below.

4.2 The Primitives

Two alternatives were available. One was to evolve a sequential set of actions that used side effects on an internal representation and left a marker on the answer. The other was to try and evolve a form that would evaluate and return the answer purely functionally. We decided it would be interesting to try the latter first, since this was essentially how the ALIVE solution worked and it would allow us to sanity check the GP system with a known hand-coded solution. Future research will experiment with the side-effecting primitives, which is closer in spirit to Ullman's work.

In a GP system with a limited number of fitness cases and sufficiently powerful primitives, it is likely that solutions will be too specific. For example, if there is some way to use the primitives to recognize each fitness case and return the fixed answer for that case, GP will tend to exploit it. Such solutions are often fragile and will not generalize to new cases. Since we had a small number of fitness cases, we limited the power of our primitives. The primitives were carefully chosen to be "weak" in the sense that they had little observational power. For instance, there are no conditionals and no observers for the values of the terminals that could be used in conditionals. We discuss generalization results in Section 5.

Terminals In our implementation, the terminals actually hide a fair amount of preprocessing. We assume that the set of universal routines as proposed by Ullman has already operated and that we have found the moving object, subtracted the background, and found the bounding box and centroid of the silhouette. Our terminal set is:

- Centroid point of the silhouette (*centroid*)
- Bounding Box Top Left point (*bb-tl*)
- Bounding Box Bottom Right point (*bb-br*)
- Percentages (0%, 10%, 20%, ..., 100%)

We defined a *point* type as an ordered pair of integers. The first three terminals are of this type. Percentages are floating point values, 0.0, 0.1, 0.2, ..., 1.0.

Functions We chose a set of primitive functions that was similar to that used in the original ALIVE code. These were the following:

- Point Operators (*pt+, pt-, point-between*)
- Feature Detectors (*find-bottom-edge, find-top-edge*)
- Point List Filters (*leftmost-point, rightmost-point, average-point*)

The first two operators add and subtract points. The third, *point-between*, takes two points (which define a rectangle) and two percents and returns the point in the rectangle specified in normalized coordinates by the percents.

The edge detectors look for a fixed size edge in a rectangular window specified by two points. One detector finds a transition from dark to light as the y-value increases, the other a transition from light to dark. These are implemented as matched filters. A fixed-size edge template (five pixels wide by four high, with half the pixels on) is convolved around the image to produce the number of matching points at each location. The set of points with the maximum correlation is then calculated and returned. It is hoped that the system will discover a small enough window of the image to apply the filters on so that the result set is small and the points near the desired solution.

The point list filters do the obvious things — they return the point with the smallest x-coordinate, point with the largest x-coordinate, and average all the points in the list, respectively. These are necessary for dealing with arms held out to the sides, where each edge detector would return a long list of matches along the edge.

Since the routine supplies the window arguments to the edge detector, they can be invalid for several reasons. For example, they can be outside the area of the image. For these degenerate calls, we just return the top left window point arbitrarily. It might be more intelligent to only look at the part of the window within the image, but we prefer that a proper program exhibits a finer control on the windows chosen.

4.3 The Fitness Measure and Fitness Cases

Koza describes the concept of a "hit." When a program gets the answer on a fitness case (henceforth *cases*), it is considered to get a hit. The GP system stops running when it gets hits on all the cases or reaches a maximum generation. We define a hit to be getting within three pixels of the hand position. The hands are about five pixels wide, so this error margin is still usually within the area of the hand.

The fitness of a routine ρ where $\rho(i)$ is the point returned by the routine on fitness case i, is the number of missed cases plus the logarithm of the error (Euclidean distance) between $\rho(i)$ and the actual hand location summed over fitness cases, plus some other constraints:

$$Fitness(\rho) = (N - Hits) + \gamma \; Error(\rho) + \alpha \; Size(\rho)$$

where

$$Error(\rho) = \left(\sum_{i \in Cases} \log(||\rho(i) - actual_i|| \; + \; 1.0) \right)$$

The length term was a parsimony constraint added to satisfy the Occam's Razor impulse for simple-looking solutions. Also, simple solutions are often more general, which we desire. The *size* of a form is the total number of nodes (internal and leaves) in it. We chose a coefficient that would only make length matter when the answer was "close" to a fitness of zero (specifically, $\alpha = 0.1$). Thus, of two programs with the same error, the shorter would be more fit.

We chose a γ of 1.0 in the trials reported below, making misses greater than around ten pixels be considered exponentially less important. Since we are trying to optimize hits, we also add the number of misses to the function.

Early runs used just a sum of absolute errors instead of the logarithmic sum. This tended to reduce overall fitness, but didn't optimize the number of hits — that is, it found only average results on all cases rather than trying to optimize accuracy. It seemed that the system was missing potentially very good solutions because it happened to be *way* off on just a few bad cases. In order to maximize hits, we first tried replacing the sum of absolute errors with the number of misses. This fared poorly, probably due to lack of gradient information on whether one solution was better than another, even if both had the same number of hits. A noticable improvement was discovered by adding the logarithmic sum, which effectively considered points way off or very far off to be about the same fitness. This has the effect of giving gradient information for near misses, but just some bad score if it wasn't close at all. Several suggestions for ways to avoid this "sensitivity to fitness function," which seems to be a problem in GP, are discussed in Section 6.

5 Results

First, we note that finding the "perfect" answer to every fitness case isn't necessary in the ALIVE environment, since it involves a time series of silhouettes. A post-processing part of the system can remove outliers, use Kalman filtering or other methods to clean up errors. Also, unless the hands are doing something interesting, like moving around, we don't care where they are, like just hanging to the side of the person. So we don't mind if it misses hard cases, but care a lot if it misses easy ones.

We did multiple runs with a population size of 500. Copy percent was zero, crossover was 85%, and mutation was 15%. The effective copy rate was still about 10% due to failed crossovers[2], however. The initial program maximum depth was five, the maximum program depth after crossover or mutation was eleven, and the maximum mutation subtree depth was four. A worst-case combinatorial analysis on the size of this search space with these depth restrictions is simple. Given a maximum depth of d and noticing that the branching factor is less than two (those functions with more arguments only have two that can recurse), we compute that there are a maximum of 2^{d-1} internal nodes (where functions can be added) and 2^{d-1} terminal nodes. Note that there can be more than 2^{d-1} terminal nodes due to the percent terminals for the point-between functions. If every internal node was a point-between, we'd have another 2^{d-1} terminals, for a total of 2^d. If there are α terminals and β functions to choose for each point, the total number of maximal trees of depth d is $\alpha^{2^d}\beta^{2^{d-1}}$. Since all size trees of depth $1, \ldots, d$ are allowable, we have a search space of size:

$$\sum_{i=1}^{d} \alpha^{2^i}\beta^{2^{d-1}}$$

For our parameters, our search space is approximately 10^{2166}. This is astronomically huge. Of course, not all of the programs in this space will be distinct, but this gives us an idea of the area we are searching for good solutions. Also, the parsimony constraint will tend to make us look at smaller programs, unless the really good ones are larger, in which case it will have to search larger programs.

Results for the left hand runs are summarized in Figures 2 and 3. These figures represent the best of generation individual's hits and fitness, averaged over twelve

[2] When crossover cannot be completed due to size restraints, it fails and the parents are copied.

separate *tabula rasa* runs. The error bars show a standard deviation from the mean. The best left-hand program found across all runs was:

```
(POINT-BETWEEN
  (LEFTMOST-POINT
    (FIND-BOTTOM-EDGE
      TL
      (LEFTMOST-POINT
        (FIND-BOTTOM-EDGE
          (POINT-BETWEEN TL BR 20% 70%)
          (LEFTMOST-POINT
            (FIND-BOTTOM-EDGE
              (POINT-BETWEEN TL CENTROID 0% 90%)
              (POINT-BETWEEN TL BR 0% 80%)))))))
  TL 30% 10%)
```

which received 43 hits out of the possible 46, for an accuracy of 93%. Its predictions are shown in Figure 1.

The first hand-coded solution we created received only 4 hits and a standardized fitness of 78.6. It was:

```
(LEFTMOST-POINT
  (FIND-BOTTOM-EDGE
    TL
    (POINT-BETWEEN TL BR 20% 90%)))
```

This just looks for a bottom edge in a window on the left side of the image. It returns the leftmost maximal correlation point. Since the correct locations are defined as the center of the hand, this doesn't often get close enough for a hit. The next hand-coded program was better:

```
(POINT-BETWEEN
  (LEFTMOST-POINT
    (FIND-TOP-EDGE TL (POINT-BETWEEN TL BR 20% 90%)))
  (LEFTMOST-POINT
    (FIND-BOTTOM-EDGE TL (POINT-BETWEEN TL BR 20% 90%)))
  50%
  50%)
```

This finds a top and bottom edge and averages them, hoping to get the center of the hand. This gets 12 hits and a fitness of 79.12. The best hand-coded left-hand program, which took about an hour of tweaking, was:

```
(PT- (LEFTMOST-POINT
       (FIND-BOTTOM-EDGE TL (POINT-BETWEEN TL BR 20% 90%)))
     (PT- (POINT-BETWEEN TL CENTROID 0% 10%) TL))
```

This gets 40 hits (87% accuracy) and a fitness of 26.64. It finds the bottom edge of the hand and moves the answer up a little. It is encouraging that we can find a solution as good or better than this one automatically in about the same amount of time.

Clearly most runs had converged by 100 generations. Since this is equivalent to evaluating 50,000 individuals, (500 per generation times 100 generations) we decided to compare results against 50,000 randomly created programs (with no evolution) to see if GP was more efficient. These results are summarized in Figures 4 and 5. Note that the y-axis is logarithmic scale. Most random programs get zero hits. The best are about 15 hits. This pales in comparison to the consistent good results using GP, with similar computation expenditure.

Results for the right-hand runs are summarized in Figures 6– 9, similarly to the left hand runs. These results are less good than those obtained for the left hand problem. Most notably, the GP algorithm seems to converge more slowly and with a wider variance than the left-hand cases. This is probably due to the fact that the right hand cases are much harder or more unfair. For instance, in many of these cases the hands are somewhere in front of the body where our weak primitives will not work. The position given as the "answer" by a human was a guess as to where the hand probably was, using knowledge of human anatomy, etc. Probably it is not right to ask that it find the hand in these cases since its primitives cannot locate it here. Even so, it manages to find the hands in almost all the cases where we would care where the hand is. One subset of cases it misses are when the hands are raised above the shoulder since this subset of problems requires a very different solution than the majority of the other cases. These results seem to suggest that it is harder to converge on a solution when there are fitness cases that are unsolvable.

The best right-hand program evolved across all runs was:

```
(RIGHTMOST-POINT
 (FIND-TOP-EDGE
  (POINT-BETWEEN
   (RIGHTMOST-POINT
    (FIND-TOP-EDGE
     (RIGHTMOST-POINT
      (FIND-BOTTOM-EDGE
       (POINT-BETWEEN BR CENTROID 0% 60%)
       (POINT-BETWEEN
        (POINT-BETWEEN BR CENTROID 0% 60%)
        CENTROID
        80% 90%)))
     (POINT-BETWEEN
      (LEFTMOST-POINT
       (FIND-BOTTOM-EDGE
        (POINT-BETWEEN TL CENTROID 80% 60%)
        (POINT-BETWEEN BR CENTROID 0% 60%)))
      (POINT-BETWEEN TL TL 80% 10%)
      0% 40%)))
  (RIGHTMOST-POINT
   (FIND-BOTTOM-EDGE
    (POINT-BETWEEN BR CENTROID 0% 60%)
    (POINT-BETWEEN TL CENTROID 80% 90%)))
  100% 10%)
 (POINT-BETWEEN
  CENTROID
  (RIGHTMOST-POINT
   (FIND-BOTTOM-EDGE
    (POINT-BETWEEN TL BR 80% 40%)
    (POINT-BETWEEN
     CENTROID
     (POINT-BETWEEN BR BR 60% 30%)
     100% 80%)))
  100% 70%)))
```

which received 32 hits, for an accuracy of 70%. Its predictions are shown in Figure 1. A surprise was that it managed to get case 34 correct, where both hands are on the same side of the body. It also failed to get seemingly easy ones, such as 6 and 25.

The best hand-written right-hand program was:

```
(PT-
  (RIGHTMOST-POINT
     (FIND-BOTTOM-EDGE (POINT-BETWEEN TL BR 80% 0%) BR))
  (PT- (POINT-BETWEEN TL CENTROID 0% 10%) TL))
```

which received 26 hits (57%) and a fitness of 54.67. This does the equivalent of the left-hand program, looking in the right 20% of the bounding box for the rightmost bottom edge and moving it up a little.

We also tested whether these solutions would work on other fitness cases than those in the training set. We created 46 "new" fitness cases by mirroring the existing cases through the vertical axis. In this manner, we could also compare how the programs evolved for that particular side compared to the one evolved for the other side. Testing the best left-hand program on these 46 flipped cases produced 26 hits (57% accuracy) and a fitness of 53.3. This is not as good as the evolved version for the right side; however, the best right-hand program gets 32 hits (70%), which is not far from that. Therefore, it does generalize to some degree.

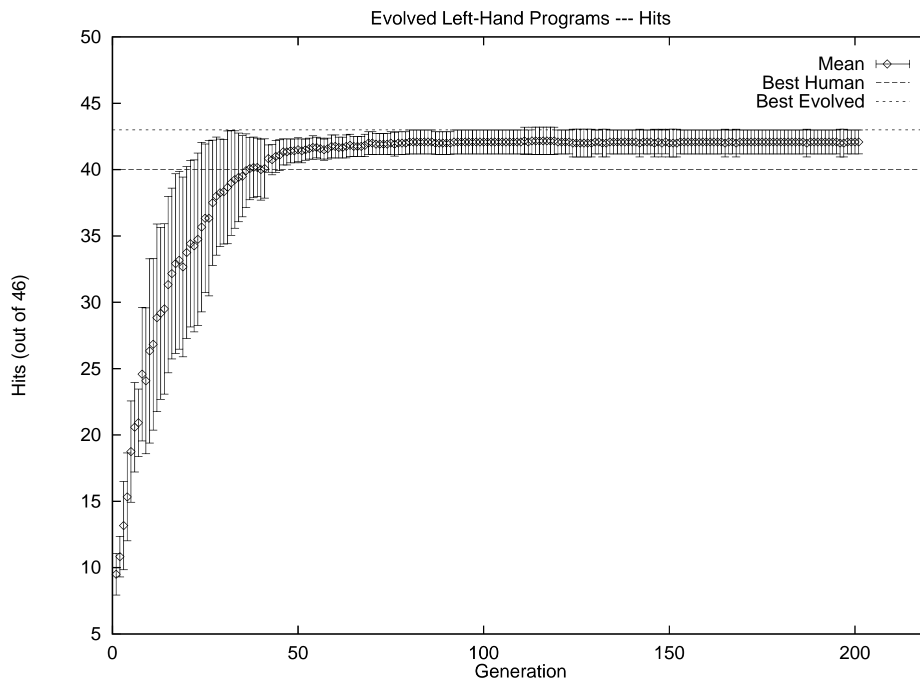

Figure 2: Population: 500. Copy: 0%. Crossover: 85% Mutate: 15%. 46 fitness cases. Mean and Std. Dev. over 12 trials.

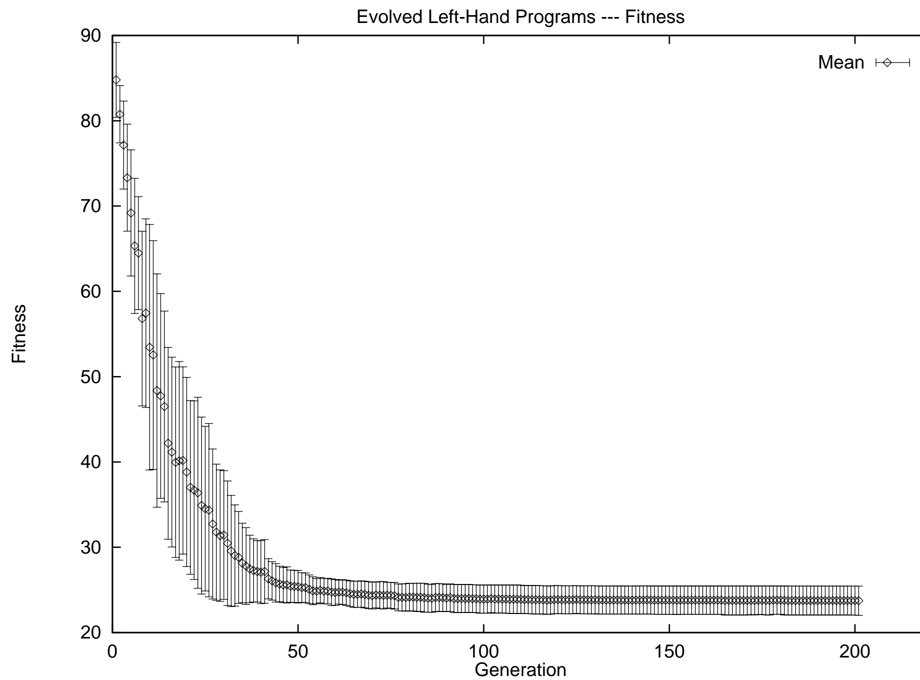

Figure 3: Population: 500. Copy: 0%. Crossover: 85% Mutate: 15%. 46 fitness cases. Mean and Std. Dev. over 12 trials.

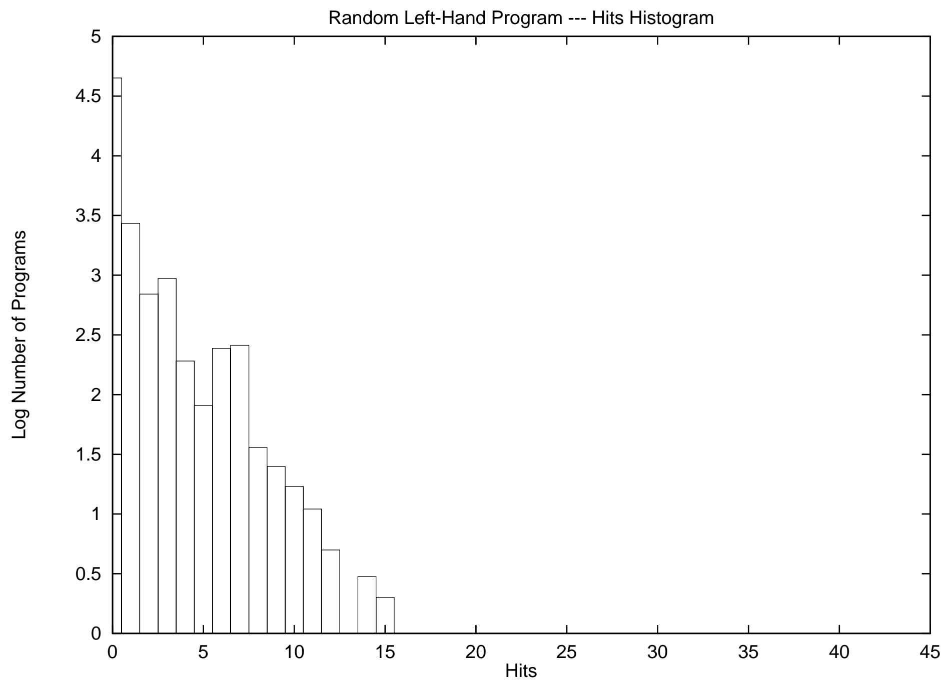

Figure 4: Hit histogram for 50,000 left-hand programs generated randomly and not evolved. Note: y-axis is log number of programs.

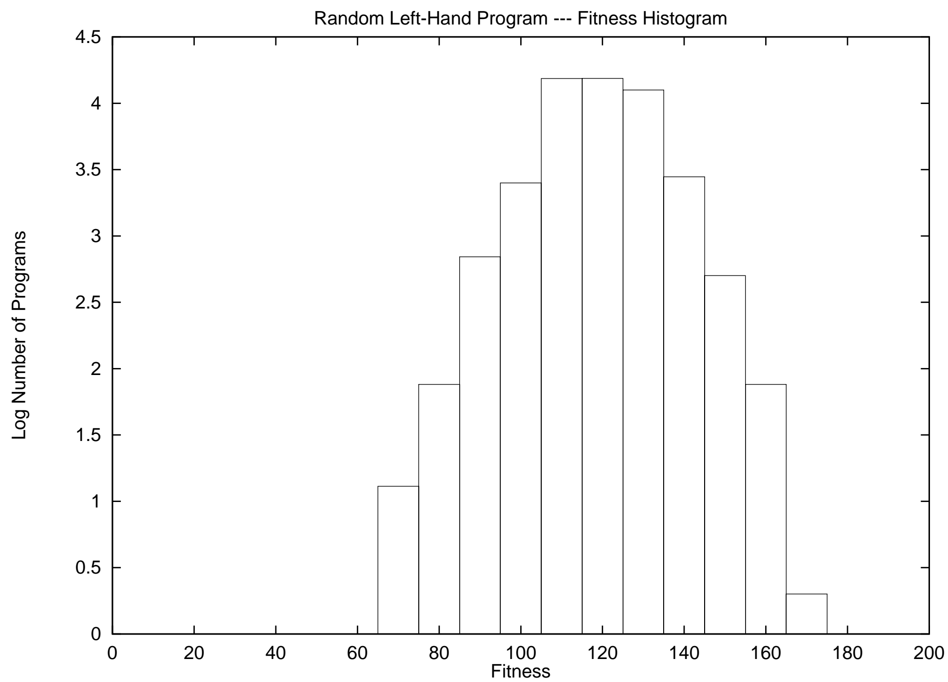

Figure 5: Binned fitness histogram for 50,000 left-hand programs generated randomly and not evolved. Note: y-axis is log number of programs.

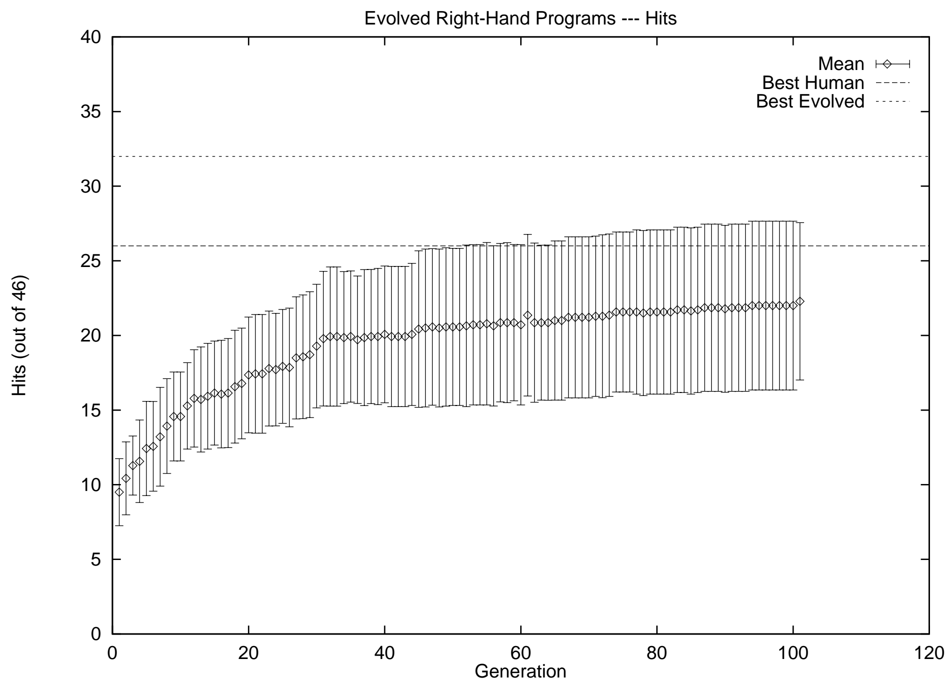

Figure 6: Population: 500. Copy: 0%. Crossover: 85% Mutate: 15%. 46 fitness cases. Mean and Std. Dev. over 15 trials.

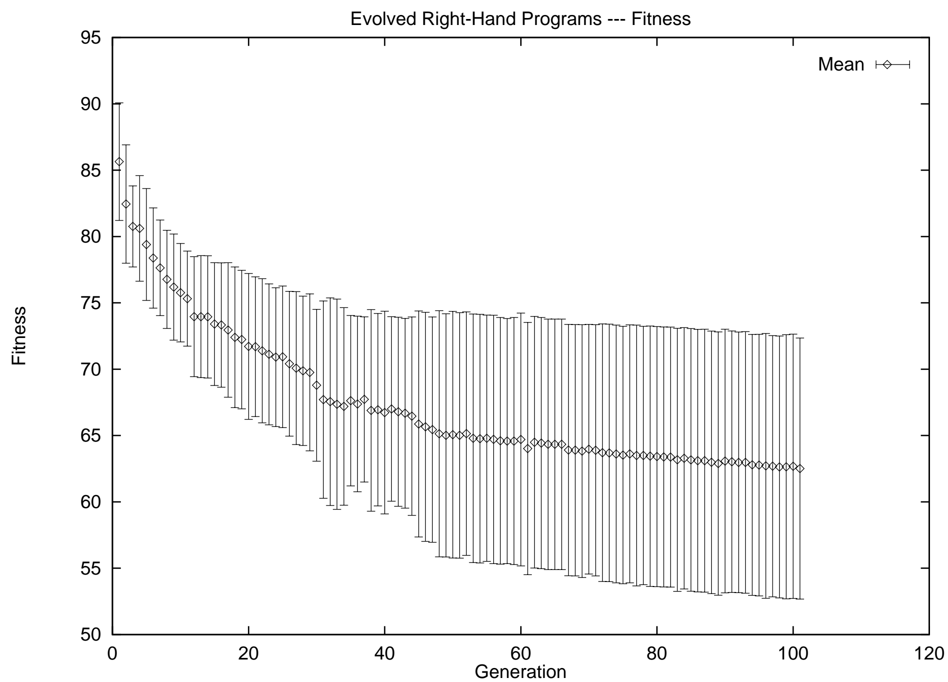

Figure 7: Population: 500. Copy: 0%. Crossover: 85% Mutate: 15%. 46 fitness cases. Mean and Std. Dev. over 15 trials.

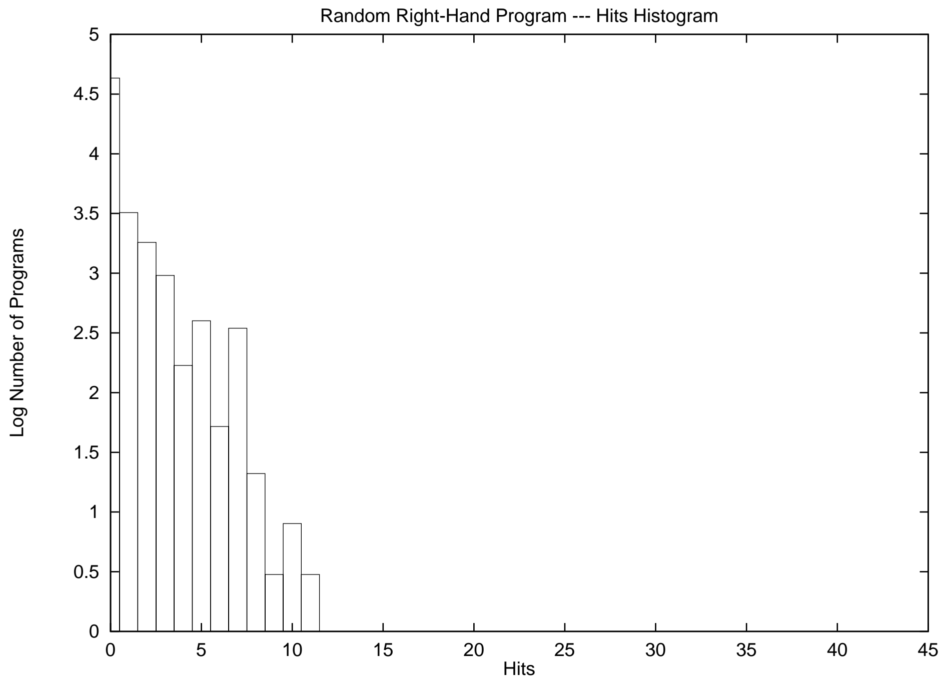

Figure 8: Hit histogram for 50,000 random right-hand programs. Note: y-axis is log number of programs.

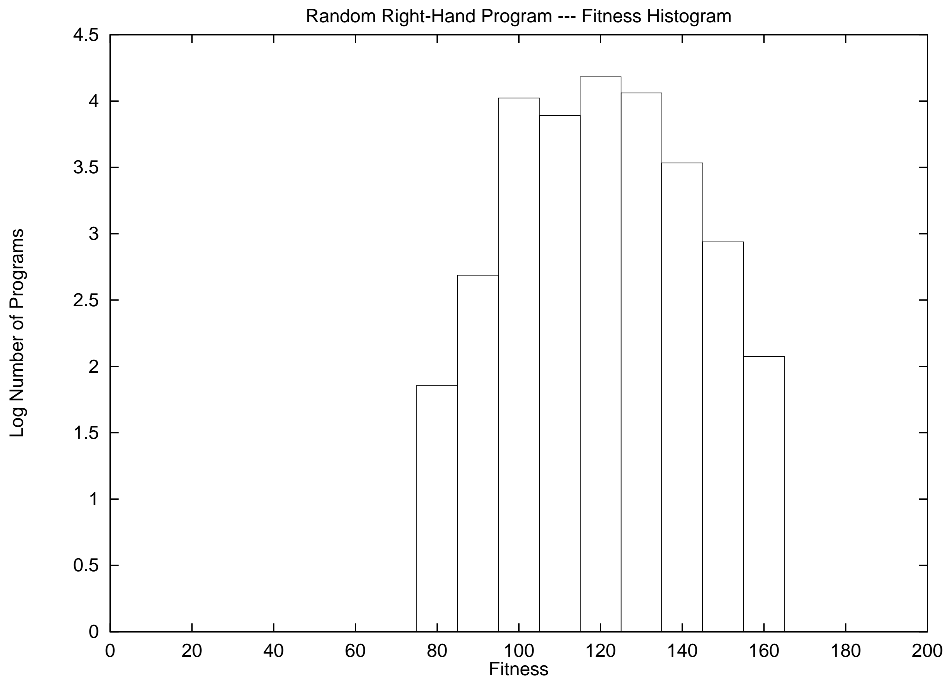

Figure 9: Binned fitness histogram for 50,000 random right-hand programs. Note: y-axis is log number of programs.

We also tested the best right-hand program on the flipped left-hand cases. It gets 27 hits (59%) and a fitness of 54.5. This is not nearly as good as the evolved programs for the left side, but it is still much better than random. Clearly some generalization is occurring. A probable explanation for the poor performance is that there were less "good" fitness cases in the right hand set from which to learn, hobbling the evolution. It also had to trade-off for solutions it could not solve.

Finally, we did 22 cross-validation runs for the left hand GP search. Eighty percent of the 46 fitness cases (30) were chosen randomly to serve as the fitness cases for each separate run. A best solution was evolved for each of these sets of fitness cases and tested against the remaining 10 cases. Results were promising: a mean accuracy of 77% ± 21% was attained.

6 Future Research

We plan to try the Ullman and Chapman style primitives in future experiments. By allowing the primitives to side-effect an incremental representation, we hope to make the search space simpler. For example, being able to store a value that is the result of a difficult computation in a register is useful when you need that result in several places. In a functional approach, you must evolve the hairy subroutine in multiple places.

We also want to use many more fitness cases to be sure the solutions are not overfitting the set. One way to get more fitness cases is to use a computer model of a human body and project it into a silhouette. In this way, we get the silhouette and know where the hand is in the image (from the projection). Thus, we'd have a continuum of fitness cases to sample from each time and could vary the person's height and size. This would get a human supervisor out of the loop, as well. Even more interesting would be to use a different Genetic Algorithm to coevolve configurations of this model with the solutions. Coevolution has been shown to be faster and more robust than a fixed fitness function in some cases [Koza, 1991, Hillis, 1991]. One might also want to add a weighting to the fitness cases based on some notion of their statistics; that is, more common postures should be focused on rather than contorted postures. Finally, giving the system a time series of silhouettes will add a level of power — it can use its previous estimates of hand position to figure out cases where the hand passes in front of the body, for example.

Another way in which we could get the human supervisor out of the loop would be to use a second sensory modality, e.g. sound or touch, to tell the creatures in ALIVE where the hand is at some point in time, so that it can learn about one sensory modality (vision) based on how it correlates to a second sensory modality.

7 Conclusions

We applied Genetic Programming to the problem of creating a simple visual routine, namely locating the left and right hands in a silhouette of a person. Results were promising. We hope to extend the system in terms of speed of computation and use of solutions in a real system. We also hope to try the system with more general primitives on a diverse set of problems, including template matching in spatio-temporal sequences. The current GP algorithm was sensitive to the choice of fitness function. It is hoped that some new methods, such as coevolution, can be used to solve this problem.

Acknowledgements

Dave Glowacki (dglo@cs.berkeley.edu) provided the public-domain C GP implementation used (and modified) in the research described in this paper.

References

[Aloimonos, 1993] Y. Aloimonos. *Active Perception.* Lawrence Erlbaum Associates, Inc., Hillsdale, 1993.

[Ballard and Whitehead, 1990] D. Ballard and S. Whitehead. Active perception and reinforcement learning. *Proceedings of the Seventh International Conference on Machine Learning*, 1990.

[Ballard, 1989] D. Ballard. Reference frames for animate vision. *Proceedings of IJCAI-89 Conference, Detroit*, 1989.

[Chapman, 1993] D. Chapman. *Vision, Instruction and Action.* MIT Press, Cambridge, Massachusetts, 1993.

[Cliff D. T. and I., 1993] Husbands P. Cliff D. T. and Harvey I. Evolving visually guided robots. *Proceedings of the second international conference on Simulation of Adaptive Behavior (SAB 92), MIT Press*, 1993. Ed. by J.-A. Meyer, H. Roitblat and S. Wilson.

[Hillis, 1991] D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Artifical Life II, SFI Studies in the Sciences of Complexity*, vol. X, 1991.

[Koza, 1991] J. R. Koza. Genetic evolution and coevolution of computer programs. *Artificial Life II, SFI Studies in the Sciences of Complexity*, vol. X, 1991.

[Koza, 1992] J. R. Koza. *Genetic Programming.* MIT Press, Cambridge, Massachusetts, 1992.

[Maes *et al.*, 1993] P. Maes, B. Blumberg, T. Darrell, and S. Pentland. Alive: An artificial life interactive video environment. *Visual Proceedings of Siggraph '93, ACM Press*, 1993.

[Maes *et al.*, 1994] P. Maes, T. Darrell, B. Blumberg, and S. Pentland. The alive system: Full-body interaction with animated autonomous agents. *Submitted for publication*, 1994.

[Marr, 1982] D. Marr. *Vision.* W.H. Freeman, San Francisco, CA, 1982.

[Ramachandran, 1985] V. S. Ramachandran. Apparent motion of subjective surfaces. *Perception*, 14:127 – 134, 1985.

[Ullman, 1987] S. Ullman. Visual routines. *Readings in Computer Vision*, pages 298 – 327, 1987. Ed. by Martin A. Fischler and Oscar Firschein.