Evolving Visual Routines

by
Michael Patrick Johnson

B.S., Computer Science
Massachusetts Institute of Technology, Cambridge, MA
June 1993

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE IN MEDIA ARTS AND SCIENCES
at the
Massachusetts Institute of Technology
September 1995

Signature of Author ______________________________
Program in Media Arts and Sciences
11 August 1995

Certified by ______________________________
Pattie Maes
Associate Professor of Media Arts and Sciences
Thesis Supervisor

Accepted by ______________________________
Stephen A. Benton
Chairperson
Departmental Committee on Graduate Students

Evolving Visual Routines

by
Michael Patrick Johnson

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning
on August 11, 1995
in partial fulfillment of the requirements for the degree of

Master of Science in Media Arts and Sciences

Abstract

Traditional machine vision assumes that the vision system recovers a complete, labeled description of the world [Marr, 1982]. Recently, several researchers have criticized this model and proposed an alternative model which considers visual perception as a distributed collection of task-specific, context-driven visual routines [Aloimonos, 1993, Ullman, 1984]. 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, 1992]). In this paper we propose an alternative approach in which visual routines for simple tasks are created 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 (GP) techniques [Koza, 1992]. The results obtained are promising: the evolved routines are able to correctly process up to 98% of the test images, which is better than any algorithm we were able to write by hand. In addition, the solutions generalize to problems outside the training data. The fact that low-level, reuseable primitives can be composed into visual task solutions implies that GP is a good way to automatically create multiple visual routines.

Keywords: Genetic Programming, Visual Routines, Active Vision, Machine Learning

Thesis Supervisor: Pattie Maes
Title: Associate Professor of Media Arts and Sciences, MIT Media Arts and Sciences Program

This work was supported in part by NSF grant number IRI-92056688.

Evolving Visual Routines

by
Michael Patrick Johnson

The following people served as readers for this thesis:

Reader: ______________________________

Alex P. Pentland
Professor of Computers, Communication and Design Technology
Program in Media Arts and Sciences

Reader: ______________________________

Ian Horswill
Postdoctoral Associate
MIT Artificial Intelligence Laboratory

Contents

List of Figures

List of Tables

Acknowledgements

Much thanks go to Pattie Maes, my advisor. Pattie has been supportive and understanding, even as I switched thesis topics faster than a weasel on speed. She is also to thank for the original idea of applying GP to active vision and for helping whip the early resarch project into a journal paper. In addition to being a great advisor, she has also been the model for hipness for our entire group. Case in point: there is a large weekend-long party at one of the MIT dormitories each year called Steer Roast. I knew that Pattie was just too cool when she came into the office and asked why we hadn't stopped by there yet.

Thanks to Justinian Rosca for reading an early (read: painful) draft of this thesis, for many stimulating discussions on GP, and for bringing cheesecake or ice cream late at night.

I would like to thank Ian Horswill and Sandy Pentland for serving as readers for this thesis.

Thanks to Linda Peterson for being there for the rough bits, of which there were many.

Special thanks to Jennifer SoYoung Lee for listening to me babble about virtual dogs, machine learning and Genetic Programming and making me smile even when my code had crashed into flaming bits.

Bruce Blumberg, my officemate, gets a gold star for not dismembering me when I was at my most annoying. He also deserves much thanks for giving me great ideas and being a sounding board for my more inane ones.

Thanks to David Glowacki (dglo@cs.berkeley.edu) whose public-domain Genetic Programming C library I snarfed from the GP WWW site and used for much of this research. It certainly made a difference over the slow LISP version I had implemented and I was thrilled to not have to write it myself in C.

The DOOM deathmatch crew gets thanks for being there to pump 16 ounces of hot lead into when I needed to vent. Frag! Ooh, that *had* to hurt!

I would also like to thank the Guinness brewery for giving me 16.9 ounces of creamy extra stout incentive to look forward to every night.

Finally, and most importantly, I thank my family for being understanding and supportive of me, from diapers to dissertations.

Chapter 1

Introduction

One of the hardest problems when building an intelligent agent situated in the real world is the perception problem. Perception is the process which allows the agent to sense its world. Within perception, computer vision is perhaps the most difficult due to the vast quantity of data. Traditional machine vision as pioneered by Marr assumes that the vision system produces a labeled, perfectly resolved model of the world, distinguishing and representing all objects [Marr, 1982]. This model, known as general vision, has been shown to be intractible in practice.

Active vision is a new paradigm for machine vision that has received significant attention recently [Ballard, 1989, Aloimonos, 1993]. One of the insights that active vision applies to the problem is that most of the time an intelligent agent does not even *need* all the information in a fully descriptive representation, since it is involved in a particular task that only requires specific knowledge of certain objects. Several researchers have proposed ways of allowing the agent's central control system to actively modulate 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 do not need to process and represent all the paper clips, crumbs, every character on every paper, and all the rest of the clutter usually on my 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.

To explain how task-dependent visual processing could be implemented, Shimon Ullman proposed the *visual routines* model [Ullman, 1984]. This model describes a set of primitive operations that can be applied to an input image in order to find spatial relations between objects as well as other useful information. The composition of these operations is called a routine. Visual routines, Ullman argues, are useful for the inherently serial aspects of vision, such as visual search. He suggests that given a specific task by the central control system, the visual routines processor creates or "looks up" an appropriate visual routine and applies it to a base representation of the input image, perhaps changing it in some way or returning a result. It may then be requested to solve another visual task 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. One possible answer is that the central system does some intelligent reasoning in order to determine which routine will solve the task.

On the other hand, V. S. Ramachandran, a human vision researcher, suggests a "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 asserts that this is the manner in which natural selection operates — anything that works will be exploited. Hence, he concludes, perception may be 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 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, 1992], 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 example (and the one on which we will focus) is the simple vision system used by the ALIVE (Artificial Life Interactive Video Environment) virtual environment project [Maes *et al.*, 1995, 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. The system employs a set of hand-coded low-level heuristics for solving specific visual tasks involved in processing live camera input of a person interacting with the system.

These tasks include perceptual problems such as "Find the hands," "Is the person sitting?" and "Is the person pointing?".

The obvious problem with writing routines by hand is that it ties up a lot of programmer-hours fiddling with parameters and conditions, trying to get a program that works well all the time. In addition, if a new visual task needs to be solved, a whole new set of routines has to be created.

An ideal solution would be to find an automatic method for creating appropriate visual routines for a new task. This thesis describes one such method — evolutionary computation — applied to the aforementioned ALIVE visual task. In particular, the Genetic Programming (GP) paradigm described by John Koza [Koza, 1992] was chosen since it meshes extremely well with the visual routines model. GP is a method for automatic induction of computer programs using simulated evolution. Using visual routines as primitives, GP can be used to compose them into programs which will solve a given visual task.

The remainder of this thesis is organized as follows: Chapter 2 describes relevant background in active vision (particularly visual routines) and genetic programming. It also describes related work in visual routines and using genetic programming for vision. Readers unfamiliar with GP should glance minimally at Section 2.2. Readers unfamiliar with the Visual Routines theory should skim Section 2.1.1. Chapter 3 presents the ALIVE problem domain and two implementations of visual primitives — functional and imperative. Chapter 4 describes GP experiments performed using the two sets of visual primitives and presents results. Chapter 5 answers questions raised by the results and evaluates them. Chapter 6 presents conclusions and suggests future directions for the research. The best imperative programs found are included in Appendix A since they were too large to be included in the results chapter.

Chapter 2

Background and Related Work

This chapter describes relevant background for the work described in this thesis. It is divided into three sections: Section 2.1 briefly presents the active vision paradigm and serves mainly to introduce the unfamiliar reader to Ullman's Visual Routines theory on which this research is based. Readers familar with the theory may skip the section. Section 2.2 describes the Genetic Algorithm and basic Genetic Programming (GP). Readers familiar with GP can skip this section. Finally, Section 2.3 surveys related work in automatic creation of vision routines.

2.1 Active Vision

Active vision is a new paradigm for computer vision in embodied (situated) agents, such as mobile robots. It consists of two main ideas, which are closely related:

- An embodied agent can actively control which perceptual inputs it receives over time by performing actions and can use this temporal information in processing.
- The agent is almost always engaged in a particular task which only will require perceptual processing relevant to that task.

These concepts are different from the traditional machine vision school as described by Marr [Marr, 1982] and others. The traditional model, following the Artificial Intelligence research of the time, assumed that the role of the vision system was to produce a labelled, three-dimensional model of the scene. It also specified that this could be done in a bottom-up, passive manner, with no control from the "higher-level" cognitive components, such as the planning system. The vision system was given an image and had to create the scene model from just that image. For example, it had to compute the shape, orientation, color and identity of every object in the scene. In practice, creating this model efficiently has proven intractible. Mobile robots using such vision systems were slow, unreactive and often failed to solve their tasks.

The two active vision concepts can simplify the general computer vision problem in many ways since the agent is embodied. For example, an agent which can move its head side-to-side can much more easily extract depth information than an agent which is fixed in place and has to use other cues such as shading. The second concept, also known as *task-dependent* vision, allows the vision system to bypass the general problem of 3D scene reconstruction and only extract the information which is required to solve the current task. For example, a robot which is trying to navigate down a corridor does not need to process the text of every sign posted in the corridor — it only needs to locate and avoid obstacles and to keep moving forward. Such a robot can use both active vision concepts. First, it can use optical flow to locate potential obstacles and stay centered in the hallway. This is an example of using the fact that the camera is on a moving agent to extract useful information. Second, it will only do this processing if the control system is currently trying to navigate, as opposed to identifying an object. This is an example of using task-dependent routines only when they are needed. Indeed, such active vision methods have been shown to be successful in mobile robots. For example, Horswill's *Polly* [Horswill, 1993] robot was able to navigate through the corridors of a building, avoid obstacles and give people tours, all in real-time.

Active Vision is a young field which is evolving in many ways. Rather than go into a detailed discussion of the various areas of research, the reader is referred to collections such as [Aloimonos, 1993] and [Blake and Yuille, 1992]. In the next section we describe Ullman's visual routines model. The visual routines model was not originally designed as an active vision theory, but many ideas from it have been incorporated into active vision systems by researchers such as Chapman [Chapman, 1992] and Whitehead [Whitehead, 1992].

2.1.1 Visual Routines Theory

One example of a task-dependent theory of vision is Shimon Ullman's Visual Routines theory of intermediate vision [Ullman, 1984, Ullman, 1987]. The visual routines model 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 Ullman refers to as 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 also makes a 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 the following set of specific routines which he admits is not comprehensive, but seems 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, 1984]). 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.

- *Curvature Segmentation* segments an object based on areas of high curvature on the boundary. Ullman suggests that this could be useful for recognition tasks.

Other routines can be applied to the results of these computations. For example, a point thought to be inside a closed curve could be marked and then 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.

Following Ullman, David Chapman applied similar operations to the domain of visual attention in his Sonja video game playing system [Chapman, 1992]. For example, 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. 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 using other routines. 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 correctly. An automated technique for finding visual routines is thus quite desirable.

A major criticism of Chapman's research is that it 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.

1. Let $\mathcal{P}_i$ be the population set of all individuals (genotypes) in generation i.
2. Initialize $\mathcal{P}_0$ to n random individuals.
3. Loop until some stopping condition:
   (a) For each $p \in \mathcal{P}_i$, express the phenotype of p and use it to calculate **fitness**(p)
   (b) $\mathcal{P}_{i+1} \leftarrow \emptyset$
   (c) Until $\|\mathcal{P}_{i+1}\| = n$:
      i. Select p_1 and p_2 from $\mathcal{P}_i$ roughly proportionate to fitness value.
      ii. Create q_1 and q_2 from p_1 and p_2 using a random genetic operator from the available set.
      iii. Add q_1 and q_2 to $\mathcal{P}_{i+1}$.
   (d) Increment i.

Figure 2-1: The Abstract Genetic Algorithm Structure

2.2 Genetic Programming

Genetic Programming (GP) is a powerful new method for automatically inducing computer programs to solve tasks [Koza, 1992]. GP evolved from the original Genetic Algorithm (GA) invented by Holland [Holland, 1975]. Before discussing GP in particular, we start with a brief discussion of the history of genetic algorithms and define terminology that will be used throughout this thesis.

2.2.1 The Abstract Genetic Algorithm

The motivation behind GAs is the obvious power of evolution as seen in nature. GAs take the Darwinian idea of natural selection, or "survival of the fittest," and apply it to digital computation. The essence of the algorithm is that a population of individuals is evolved over discrete generations by allowing the more "fit" individuals a higher chance of reproduction, or passing on a piece of their genotype to the next generation. Starting from a random population of individuals which are in general very unfit, simulated evolution gradually culls unfit individuals and ideally creates progressively more and more fit individuals.

Terminology

An *individual* is a discrete entity which can somehow be evaluated for fitness. Following genetics scientists, GA researchers distinguish between the *genotype* and *phenotype* of an individual. The genotype is a *representation* or *encoding* of the individual which can be operated on by genetic operators. In biology, a creature's genotype is its particular DNA encoding. The phenotype is the *expression* of the genotype in an environment. Looking to biology again, the phenotype is the actual creature, which grows according to its genotype and environment. In a GA system, the term "individual" can actually refer to either the genotype or phenotype, depending on context. When speaking of genetic operators, it refers to genotype. When referring to an individual's fitness it refers to the phenotype. In simulated evolution, a phenotype can be an artificial creature, a graphic, or even a number — it is purely up to the programmer to create a genotype encoding that describes all the possible phenotypes desired. In Holland's original GA, a genotype was represented as a fixed length string of characters or bits, much like DNA. For example, the genotype 01101 can be expressed as the phenotype 13, if the encoding were the binary representation of integers. GA researchers have since realized that the genotype can actually be any structure which can be expressed as a phenotype through some mapping and can be operated on genetically.

A *population* is a collection of some number of individuals who are all said to be in the same *generation*. Members of the population "compete" with each other to reproduce into the next generation, based on fitness.

Fitness, which we have used loosely until now, is a real scalar value which defines how well the individual deals with its environment or solves a problem. For example, if the individual is an animal, a stronger or smarter animal will deal better with the environment and thus presumably live longer and have a higher chance to breed. It is said to be more *fit*. To simulate fitness calculation in a GA, the programmer usually specifies a *fitness function* to calculate this value. An individual's phenotype is expressed and the resulting individual is then tested according to that fitness function.

Finally, to create the next generation of individuals, members of the current population are selected proportionate to their fitness. These members may be cloned directly into the next generation or *genetic operators* may be applied to the genotypes to create new individuals. Many genetic operators have been proposed, but there is a core set of two which are often powerful enough: crossover and mutation. Crossover is a sexual reproduction which creates a new individual by taking part of each parent's genotype and combining them. In biology this is accomplished by combining one set of chromosomes from each parent. This produces an individual with some of the traits of each parent, ideally producing a more fit individual if both parents were fit. Finally, mutation is simply a small random change in some part of the genotype.

This process of creating new generations is iterated until some stopping criterion is reached. Some problems have obvious stopping points, such as when the program gets the right answer, if it is known. In other cases, such as optimization, the evolution is usually stopped when the best fitness stops changing, or reaches steady state. Some evolutions will continue changing forever; this is particularly true if the fitness function changes over time, as is the case in nature.

Algorithm description

Figure 2-1 shows pseudocode for the abstract GA. It is abstract in the sense that there are many unspecified parameters, including stopping condition, population size, fitness function, genetic operator set, how to choose a genetic operator (probability distribution), and which fitness-proportionate selection algorithm to use. Although this may seem overwhelming and more difficult than just solving the problem by hand, it has been shown that often any choice of parameters will produce good results.

Although the algorithm as shown is serial for simplicity, an important point to notice is the massive parallelism that the algorithm allows. Each individual can be expressed and have its fitness evaluated *in parallel*. This allows for large populations of individuals to be calculated quickly.

String-based GAs have been used to tackle many problems. In particular, they seem well suited to machine learning and difficult function optimization problems. A full discussion of string-based GAs is beyond the scope of this thesis. For a good discussion of the use of string-based GAs for these types of problems, see [Goldberg, 1989].

2.2.2 Basic Genetic Programming

A tree genotype

Genetic Programming in its current form is attributed to John Koza. Koza decided that using fixed strings of characters as a genotype was too limiting [Koza, 1992]. Instead, he chose LISP S-expressions to create genotype encodings. S-expressions, also known as forms, describe a function that can be evaluated in the LISP environment. Forms are essentially *trees* in the mathematical sense. They contain a single root node, a number of internal nodes, and a number of leaf nodes. Using a tree structure is more powerful than a linear fixed-length string for three reasons:

- Trees can have arbitrary size
- Trees can "change shape" significantly in crossover
- Trees are much more natural for expressing computer programs

The first point is clearly an advantage since it reduces the amount of thought a programmer must put into designing a genotype representation since he need not worry that he restrict the solutions too much be making the strings too short. The second point is similar to the first, but focuses more on morphology — whereas a string can only crossover in a small range of fixed space, a tree can add an arbitrary amount of new nodes between any other two nodes due to its hierarchical shape. Intuitively, this makes crossover more powerful. The final point is why GP is so useful — it can be used to automatically induce computer programs which compute a desired function.

Phenotype expression

The phenotype of a LISP form genotype can be expressed in essentially two ways:

- As the genotype itself (the function it computes)
- By evaluating the genotype

The difference is subtle, so more explanation is required. In the first case, the genotype does not encode some description for how to create a phenotype using some mapping function — the form itself *is* the phenotype. The function it computes, in some sense, is its phenotype. Another way to look at it is that the solution being sought by the evolution is a function, not some other kind of entity. In contrast, the other way a LISP genotype can be expressed is by considering it as the phenotype evaluation function itself. For example, the LISP form could describe the shape of an artificial creature. The creature is created from the genotype by evaluating it, and then the creature is tested for fitness. Here the solution being sought is a creature that does something, not the function that describes how to make the creature.

GP experiments often tend toward the former representation, since researchers are often looking for functions that solve problems or optimize parameters. A noteworthy example of the latter representation is Karl Sims's work on genetically created artwork [Sims, 1991], where the genotype is a function that computes the pixel values on a computer image (the phenotype) and where the fitness function is a person's aesthetic taste.

GP as search for computer programs

The goal of GP is to automatically find a computer program which solves a given problem or task. Essentially it is a search of the entire space of computer programs (expressed as LISP forms) that could be created using a set of allowable symbols. These symbols can be bound to functions, variables and constants in the LISP environment. In general, this search space is huge — it is hyperexponential in the maximum allowable depth of a program. That is, if the deepest program allowed has a depth of d, there are $O(\alpha^{\beta^d})$ allowable programs, where α is the number of symbols available and β is average branching factor of the program trees. The search starts with a population of programs which mostly compute garbage, but some of which might be useable as parts of a solution. Eventually, one hopes, a good solution will be evolved. GP performs much better than random chance in searching this space. This "nonrandomness" of GP is discussed in Chapter 9 of [Koza, 1992].

The GP algorithm

GP is an instance of the abstract GA algorithm described above. Pseudocode for the GP algorithm is given in Figure 2-2. It is almost identical to the abstract GA algorithm with one key difference — the genotype is a program and can be *evaluated*. In other words, its function is computed by interpreting it in the LISP environment, assuming the symbols appearing in the program reference functions, constants and variables.

1. Let $\mathcal{P}_i$ be the population set of all programs (genotypes) in generation i.
2. Initialize $\mathcal{P}_0$ to n random programs.
3. Loop until some stopping condition:
   (a) For each $p \in \mathcal{P}_i$, evaluate p and use its result to calculate **fitness**(p)
   (b) $\mathcal{P}_{i+1} \leftarrow \emptyset$
   (c) Until $\|\mathcal{P}_{i+1}\| = n$:
      i. Select p_1 and p_2 from $\mathcal{P}_i$ roughly proportionate to fitness value.
      ii. Create q_1 and q_2 from p_1 and p_2 using a random genetic operator from the available set.
      iii. Add q_1 and q_2 to $\mathcal{P}_{i+1}$.
   (d) Increment i.

Figure 2-2: The Genetic Programming Algorithm

For example, consider the form:

```
(not (and a b)
     (or a c))
```

When evaluated, this form will compute a boolean function of the symbols a, b and c, assuming they can be interpreted as boolean values in the environment. Its answers, or return value for various inputs (bindings of the symbols a, b and c), can be compared to desired answers in order to calculate a fitness.

An important point is that these forms are often interpreted during a GP run. This is not a limitation. For use in a real system, the programs can be compiled. They are often not compiled during a GP run since compilation overhead is often higher than interpretation, and most programs are simply thrown away after being evaluated.

Defining a GP experiment

In order to define a GP system to solve a particular problem, a programmer needs to select the following elements:

- A function set and terminal set.
- A fitness function.

- A fitness-proportionate selection function.
- A set of genetic operators.
- A stopping condition.
- Some numerical parameters.

Function and terminal set

The function and terminal sets specify which primitives the GP system can use to create a solution. *Terminals* are symbols which evaluate to a single value without having to evaluate arguments. They are often constants or variables, but can be zero-argument functions as well. *Functions* are elements which operate on a fixed number of arguments. These arguments can be terminals or functions themselves. The number of arguments a function takes is fixed and must be specified. Thus, by composing these terminals and functions, one creates trees, with functions being internal nodes and terminals being leaves.

According to Koza, these sets should be selected to have two properties — *sufficiency* and *closure* under type [Koza, 1992]. Sufficiency means that there exists some composition of functions and terminals which are able to solve the problem. Sufficiency often cannot be decided *a priori* and involves an iterative process of primitive selection. Type closure is a rigid constraint that forces every function to be able to handle any type returned by any other function or terminal. So if the system is of boolean type, all functions must accept booleans and return booleans. We describe a method to relax the closure constraint using syntactic struture in Section 3.2.1.

Fitness function

The most common method of defining a fitness function (and that which is used in this research) is to create a set of *fitness cases* which are essentially examples for the programs to be tested on. To compute a program's fitness, it is evaluated on each of the fitness cases. It is assumed that the correct answer is known. If the program gets it right, this is called a *hit*. A simple fitness function could be the number of hits the program gets, for example. For real-valued problems, Euclidean and other error metrics can be used.

A GP task can be specified as either a supervised or unsupervised machine learning problem. In the supervised case, given a set of inputs and desired outputs (training data), generate a program which produces the desired outputs given the inputs. Thus, an obvious choice of fitness function is the number of examples the program gets right. For continuous data, an error function can be used. The unsupervised case usually interprets the program as a controller of a robot or agent which acts in an environment which gives it rewards and punishments in response to its actions. The program is used to control the agent for some amount of time and the cumulative reward received is used as the fitness.

Another common addition to the fitness function is a *parsimony constraint*. This is often an additive term which lowers a program's fitness proportional to its size, or complexity. The motivation is Occam's Razor — shorter programs are usually better. There is often a trade-off between how well a program performs and how complex it is. By using too strict of a parsimony constraint it is possible to miss longer, better solutions.

Selection function

The selection function specifies how parents are to be selected, given each program has a fitness value. All the methods are fitness-proportionate, meaning higher fitness individuals have a higher chance of being selected for reproduction. Two types of selection functions are in common use. The first is a simple partitioning of probability space based on the relative fitness of an individual compared to the rest of the population. Given fitness function **fitness** and population $\mathcal{P}$, the probability of a program q being selected for reproduction is:

$$\frac{\mathbf{fitness}(q)}{\sum_{p \in \mathcal{P}} \mathbf{fitness}(p)} \tag{2.1}$$

Another common selection function is called *tournament* selection. A "tournament" is held among k individuals by picking them uniformly at random, then selecting the individual with the highest fitness. Tournament selection has the nice property that if one program has an abnormally high fitness compared to the rest of the population (even though it may not be that good in the grand scheme of things) , it does not utterly swamp the next generation as might happen when using equation 2.1. Tournament selection promotes more diversity, hopefully guarding against premature convergence to a local maximum. k is often 3 or 5.

Genetic operators

Koza describes many operators that can be applied to LISP forms [Koza, 1992]. Only the ones used in this research are described here. These are:

- Creation
- Copying (or cloning)
- Crossover
- Mutation

creation In order to create a random program tree, a random element is picked from the union of the function and terminal set. If it is a terminal, that is the final program. If it is a function, the arguments are chosen in like manner, recursively. The recursion bottoms out at the maximum depth where only terminals are allowed to be picked. Koza describes several methods for how to create an initial random population that is diverse. These involve heuristics like picking random depths for each program, choosing only functions until the maximum depth in order to produce full trees, etc. We again refer the reader to his book for more details [Koza, 1992].

copying Copying is a straightforward operation. The program is copied as is directly into the next generation.

crossover Crossover is a sexual operator — it takes two parents and produces two offspring. First, select two parents from the population using the selection criterion. Pick a random subnode, or subtree, in each one and swap them to create two new offspring.

Often GP systems have a maximum depth constraint on programs to keep computation time lower and avoid unbounded program sizes. If the newly created children violate this constraint, the operation is cancelled and the parents copied directly. It is also possible to choose the crossover nodes such that crossover only creates "legal" sized trees.

Koza and others also divide the crossover operator into two variants — internal and entire. The *internal* crossover operator only can choose among non-leaf (hence internal) nodes for crossover. The *entire* operator can choose any. The division is made since a program's leaves make of a large percentage of the program's nodes (half, in a full binary tree, for example). In order to ensure a more efficient search of the program space, it is useful to encourage crossover on internal nodes, especially early in a run. Creating an internal node crossover operator attempts to encourage this. Rosca argues that this may not as useful later in a run, however, where smaller changes may be more beneficial [Rosca and Ballard, 1995].

mutation First, select a subtree (which again could be a leaf). Replace it with a small, newly created subtree made with the creation operator. Again, mutation is liable to depth constraints and similar solutions as in crossover can be used.

Stopping condition

Usually some kind of stopping criterion is needed. Often, a maximum generation is defined, after which the best program found during the run is returned. It is also possible to stop the run when a program gets hits on every fitness case.

Parameters

Most GP systems have many parameters. Some of these include creation method, selection method, probability distribution for each genetic operator, maximum depth constraints, maximum generations, population size, parsimony constraint, and coefficients of the fitness function. Although this may appear to be a disadvantage, it is often the case that the algorithm is not very sensitive to their values. Many GP researchers use the values used by Koza [Koza, 1992] for most of his experiments.

2.3 Related Work

Recently, several other researchers have been investigating automatic ways of creating task-specific visual routines. *Jeeves* by Horswill [Horswill, 1995] and the *Meliora* system by Whitehead [Whitehead, 1992] actually use the visual routines model, but neither uses GP techniques. Other GP researchers have applied GP to visual recognition tasks, such as Koza [Koza, 1994] and David Andre [Andre, 1994], who both evolved programs for Optical Character Recognition, and Astro Teller [Teller and Veloso, 1995], who evolved programs for classifying single objects from greyscale images. Finally, Harvery, Cliff and Husbands used GP to perform visual tasks on a real robot. Each will be described in turn.

Jeeves, created by Ian Horswill, is the first demonstration of a real-time visual routines processor which operates on real camera images rather than on hand-drawn bitmaps or internal simulation structures [Horswill, 1995]. Jeeves processes the camera image into a set of low-level maps and performs a pre-attentive segmentation step to divide the image into distinct regions. Horswill presents an automata-theoretic analysis of the computational power of the visual routines theory. His analysis assumes that visual routines are used to solve Horn clauses, or simple logic statements, which refer to properties and relations in the image (for an introduction to Horn clauses, see for example [Hogger and Kowalski, 1992]). He introduces *enumeration oracles* and explains how they can be used to solve visual queries expressed as Horn clauses. He gives the following example query, which is interpreted as a conjunction of Horn clauses:

$$(blue\ x)(above\ x\ y)(above\ y\ z)$$

This query asks the visual system if there is a blue block stacked on top of at least two other blocks. The system performs a visual search to ascertain the truth value of the statement. Horswill's work is interesting for two main reasons — it is one of the first real-time, low-level implementations of the visual routines theory and it addresses the issue of how queries are actually compiled into a search automatically rather than having the programs being written by hand. Although his system is currently designed mainly for visual search, he describes how it was integrated into a mobile robot platform which utilizes the visual routines in its motor tasks, such as navigating down a corridor.

Steven D. Whitehead's dissertation research also tries to find a method for composing visual routine primitives automatically to solve a task [Whitehead, 1992]. Whitehead uses a machine learning algorithm called Q-learning [Watkins, 1989]. He applies this system to a simple block-stacking problem in which a robot arm must stack blocks in order to make a duplicate of an example stack. An *attention frame*, similar to the attention marker described in Section 3.4.3, can be moved to index blocks and query their properties. Q-learning is used to find a *policy*, or mapping from world state to next action, which solves the stacking problem. This policy can perform *overt* actions, in which the robot arm is moved and external world state changed, or *covert* attention actions, in which the attention frame and internal representation is changed. He discusses the significant problem of *perceptual aliasing* in applying current machine learning algorithms to active vision and presents a few ways to attempt to circumvent it. Many machine learning techniques (including Q-learning) assume that an agent is able to extract the full state of the world on every timestep. When these algorithms are applied to active perception systems, where the agent only has access to small amount of local state every timestep, perceptual aliasing can occur. Perceptual aliasing occurs when two actual, different states of the world are perceived as the same state to the agent. For example, an agent in a maze sees an exit north and east. Let's say this perceptual state can occur in two actual positions in the maze. In one of these, there is a monster to the north and the east is clear. In the other the reverse is true. The agent should perform a different action in each state, but has no way of telling which real state he is in through just the current perceived state.

Several researchers have applied GP to computer vision, but mainly for object recognition tasks. For example, Koza describes an experiment in character recognition [Koza, 1994]. The goal is to classify four by six bitmaps as the letter "I," the letter "L," or "NIL," meaning neither. The system pretends there is a *turtle* which can crawl around the bitmap and see the value of the nine bits in its neighborhood. Again this is similar to the attention marker used in this research (see Section 3.4.3). Koza admits this it is an expensive computation and as usual only lets the evolution proceed for 51 generations. This did not produce a solution. He also implemented a version of the problem which used Automatically Defined Functions (ADFs) which managed to find a good solution (for a description of ADFs, see [Koza, 1992]). Unfortunately, the perfect solutions were rare in the sense that many runs did not find one and the population size was large, containing 8000 individuals.

David Andre performed similar but more in-depth research on the problem of character recognition [Andre, 1994]. The most interesting facet of his work is that he performed evolution of feature detectors and decision trees that utilized them by using a string-based GA and GP simultaneously. The GA was used to evolve three by three feature detecting templates. Simultaneously, these templates were used by the GP evolution in order to classify a given five by five bitmap as an instance of a certain digit or not. Programs had to be evolved for each digit to be classified.

Harvey, Husbands and Cliff used a genetic algorithm to evolve the morphology and control of a vision system for a real robot [Harvey *et al.*, 1994]. This is an extension of research in [Cliff *et al.*, 1993b] and [Cliff *et al.*, 1993a]. The robot has a CCD camera which digitizes the world in front of it. Control structures in the form of networks are evolved to solve the task of moving the robot towards a set of targets of various sizes using the vision system. The morphology of the visual receptors (their placement on the robot and response rates) are evolved in parallel with the control networks which use the output of the vision receptors. This research is ambitious since it used not only a real camera but a real robot with the genetic algorithm. Since there was only one robot, the fitness function for each individual in the GP population must be computed serially, a time-intensive undertaking.

Finally, Teller and Veloso used Genetic Programming for the PADO (Parallel Algorithm Discovery and Orchestration) system [Teller and Veloso, 1995]. PADO performs object recognition on real grayscale camera images. Genetic programming is used to induce programs which operate on pixel values in the image and return a confidence value that the image belongs to the class the program is evolved to recognize. The system was about 50% accurate at distinguishing between seven classes of objects. Due to the complexity of the system, a detailed explananation is not possible here. The interested reader is referred to [Teller and Veloso, 1995].

Chapter 3

Evolving Visual Routines

This chapter describes the ALIVE hand-finding problem and the implementation of the GP system and primitives used for this research. Two alternatives for visual primitives were available for the hand-finding task:

Functional Compute the hand position purely functionally and return the answer as the final value of the program.

Imperative [1] Perform a sequence of actions which side-effect a global representation and leave a marker on the final answer.

Although the imperative approach is more in the spirit of Ullman's visual routines theory, the functional approach was implemented first. The functional approach was closer to how the ALIVE solution worked and it would allow us to sanity check the GP system with a known hand-coded solution. It was also much simpler to implement. We describe the set of primitives for each approach separately.

Section 3.1 gives a brief description of the ALIVE domain and the vision problem — namely, finding features in the bitmap silhouette of a person. Section 3.2 describes the GP system used for this research, which adds syntactic constraints to the basic GP paradigm. Section 3.3 describes a purely *functional* set of GP primitives which were used in the first experiments. Section 3.4 describes a side-effecting, or *imperative*, set of primitives which are more in the spirit of Ullman's visual routines model.

[1] When I asked my friend, a SCHEME lover, what the term was for a non-functional, side-effecting style of programming, he replied, "Wrong." Although I tend to agree, I decided that using *imperative* might be better for this document.

Figure 3-1: The composite image seen by an ALIVE user. Here the dog shakes hands because the user is sitting and holding out his hand.

3.1 Problem Description (ALIVE)

The Artificial Life Interactive Video Environment (ALIVE) system allows a user to interact with a virtual agent using a set of natural hand gestures. A video camera captures an image of the user, which is both used for hand gesture recognition and is directly composited in real-time with the graphical world. The composite image is displayed on a life-size projection screen in front of the user. The effect is as if the user were looking into a "magic mirror" — he sees a mirror image of himself composited with several graphical objects. He can interact with the creatures in the virtual world by gestures that are appropriate for the domain. For example, in Figure 3-1, the user is shaking hands with Silas T. Dog, the star virtual character.

The gesture recognition system relies on knowing the position of the person's hands, head and potentially feet every frame. To find these, the system first performs figure-ground segmentation in order to separate the person interacting with the system from the background. Segmentation produces a binary silhouette of the person. Specialized visual routines then process this bitmap in order to locate the desired features. These routines were written by hand and took many hours to debug and optimize; therefore, an automatic method for generating them is desirable.

The original ALIVE system used single color background subtraction ("Chroma-keying"), a method used commonly for video special effects. The user stands in front of a wall of known color and is observed with a video camera. The algorithm then replaces pixels matching the background color with computer graphics output and leaves the foreground as live video, with the effect that the user is superimposed on the graphics world. This is known as the "weatherman effect" since meteorologists use these systems for giving weather reports. In addition, the algorithm returns a binary image mask which provides a silhoutte of the foreground region.

The current ALIVE vision system actually uses a more sophisticated segmentation algorithm utilizing color class statistics. This allows the background to be fairly abitrary, rather than a single color. The only requirement is that the background is static. This new system also returns a bitmap with a single connected blob which represents the user's silhouette. The compositing in the new system is also more sophisticated, as is seen in Figure 3-1. In this system, the creatures are superimposed onto the video image, producing the visual effect that they are actually in the room with the user.

The fitness cases used in this research were silhouettes produced from the full-frame images returned by the segmentation algorithm by finding the largest connected figure component and then cropping the image to the bounding box of that component. The figure-ground segmentation and cropping would be considered universal routines in the visual routines theory. Figure 3-2 shows the full set of 46 fitness cases which were used. Most were chosen randomly from several users interacting with the system in order to get a range of cases. The first sixteen cases are a sequence of one user interacting with the system for a few seconds. Only 46 cases were used to keep the Genetic Programming runs from taking too long to compute since computational resources were limited. As will be described in Chaper 4, most of the runs take about two or three hours. Ideally, the system should be run on about 500 cases.

The "correct" hand locations, or answers to the fitness cases, were determined by manual inspection. Since the images are silhouettes, several of the fitness cases are extremely difficult to solve since part of the hand is in front of the body or is not easily locatable. For these difficult cases, a best guess was given as a location near the waist. The hand on the

Figure 3-2: The set of 46 fitness cases used.

left side of the image is called the *left* hand and the one on the right the *right* hand.[2]

3.2 GP Implementation Specifics

This section describes the specifics of the GP system used for this research. Section 3.2.1 discusses adding syntactic constraints to the GP system. Section 3.2.2 describes the evolution of the fitness function which was used.

3.2.1 Typed GP

This research uses an implementation of the GP algorithm which is similar to Koza's original system, but with the addition of a type system. This extension was made since trying to make the primitive set closed under type proved impossible. For example, if we want the program to return the point where the hand is, the program should return a **point** object which consists of two coordinates (x, y). This led to problems, however, since single valued numbers were also required as terminals. Although coercing a **point** to a scalar could be done simply by taking the x-coordinate, say, this seemed inelegant. It also made understanding the evolved programs much more difficult and hand-coded solutions difficult to write.

In order to solve this, syntactic constraints were added to the system. 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. The programmer must also specify the top level return type, eliminating the problems of programs returning incorrect data types. Only programs consistent with this type specification are allowed in the evolution. Koza discusses syntactic constraints on certain nodes [Koza, 1992], but the method described here is more general.

Typed creation

In order to create random trees consistent with the type constraints, the system starts by picking a random node with the proper top level return type. If it is function, it recursively fills in each argument slot by randomly choosing nodes with the type of that slot. This

[2]Note that this is the reverse of the user's perspective.

assumes that there always exists such a node. It would be invalid to have a signature specifying an argument of type **foo** and then having no nodes with type **foo** to select.

Typed crossover

For typed crossover, the GP system uses the following approach: one of the two mating programs is picked at random. Next it picks a random subtree within it and finds the return type, **t**. It then chooses randomly among the subtrees with root type **t** in the other mate. If there are none, no crossover occurs and the individuals are copied. The programs are also copied if crossover will produce a program deeper than the maximum allowable depth of a program.

Typed mutation

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.

Advantages

Constraining the syntax 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 and performing extensive run-time type-checking. This in turn should make the GP computation more efficient.

A significant advantage of the typed GP is that we can constrain the system in many useful ways. By creating a function that returns a special type that exists nowhere else in the terminal and function set and specifying that type as the root type, the programmer can force the system to choose that function at the root. This trick can be used to optimize the arguments of a particular top-level function. Another important advantage is that the programmer can create special terminal types that only certain functions use. If the programmer wants a function to take integers in a certain range, he makes a set of terminal variables of a special type, say **smallint**, and restricts the values to lie within the range. Thus, if larger integers are used elsewhere, the protected function won't get them.

3.2.2 Fitness Function

Koza describes the concept of a "hit." When a program returns a result close enough to the answer on a fitness case, 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. For our research we define a hit to be returning an answer within three pixels of the correct hand position. The hands are about five pixels wide, so this error margin is still usually within the area of the hand.

3.2.3 Sum of absolute errors

The first fitness function we tried was the most obvious — use the error between the desired answer and the program's result. The simplest error function is just the Euclidean distance between the two points:

$$\|\mathbf{P} - \mathbf{D}\| = \sqrt{(P_x - D_x)^2 + (P_y - D_y)^2} \tag{3.1}$$

where $\mathbf{P}$ is the program result and $\mathbf{D}$ is the desired result. This error is found for each fitness case and summed. More formally, if ρ is the routine, $\rho(i)$ is the point returned by ρ on fitness case i, $\mathbf{answer}(i)$ is the desired point for case i and $\mathcal{C}$ is the set of fitness cases, we define our first fitness function as:

$$Fitness(\rho) = \sum_{i \in \mathcal{C}} \|\rho(i) - \mathbf{answer}(i)\| \tag{3.2}$$

Notice that this is an error function, so that lower values are actually considered more fit, with zero being optimal. Although "fitness" may seem like a misnomer since "high" fitness implies low values, we will still call this the fitness function following others in the GP field.

Pilot experiments using equation 3.2 for the fitness function did not perform well. The system 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 programs because they happened to be *way* off on just a few hard-to-solve or impossible cases.

3.2.4 Misses plus absolute errors

In order to try and maximize hits, the number of misses was added onto a weighted sum of absolute errors. Formally, the fitness function was:

$$
\begin{aligned}
&Fitness(\rho) = \\
&\qquad (N - Hits) \ + \gamma \sum_{i \in \mathcal{C}} \|\rho(i) - \mathbf{answer}(i)\|
\end{aligned}
$$

where

- N is the number of fitness cases.
- $Hits$ is the number of hits that ρ gets.

This function was slightly better but still fared poorly. It still seemed that the evolution was spending too much effort in trying to solve very hard cases and missing easy ones.

3.2.5 Logarithmic errors

In order to discourage the system from overfitting to outliers, a logarithm was applied to the errors. Such a fitness function allows a program to be way off on one case and still be considered fit. The final fitness equation became:

$$
\begin{aligned}
&Fitness(\rho) = \\
&\qquad (N - Hits) \ + \gamma \sum_{i \in \mathcal{C}} \log(\|\rho(i) - \mathbf{answer}(i)\| \ + \ 1.0) \ + \alpha \, \mathbf{Size}(\rho)
\end{aligned}
$$

where

- N is the number of fitness cases.
- $Hits$ is the number of hits that ρ gets.
- **Size**(ρ) describes the complexity of ρ.

The **Size** term is called a *parsimony constraint* added to satisfy the Occam's Razor impulse for simple-looking solutions — they are often more general and efficient, which we desire since they are to be used in a real-time system. This term can be defined in many ways based on complexity theory. The simple size function used for this research is just the total number of nodes (internal and leaves) in the program.

3.3 Functional Approach

This section describes the functional system. The functional system description also appears in the *Artificial Life* journal [Johnson *et al.*, 1995].

3.3.1 Types

There are only three types for the functional system:

point An ordered pair of integers, (x, y).

point-list An arbitrary length list of points, $(P_1,\ P_2, \ldots,\ P_n)$.

percent A floating point value in the interval $[0,\ 1]$.

3.3.2 Terminals

The terminals hide a significant amount of processing. We assume that the set of universal routines as proposed by Ullman has already operated and that we have done figure-ground segmentation and found the bounding box and centroid of the most salient component (presumably the person). The terminal set is summarized in Table 3.1. The only terminals which need explaining are the percentages, which are used in the *point-between* function described below. They simply have the value of their floating point equivalents — 0.0, 0.1, . . . , 1.0.

3.3.3 Functions

The function set is summarized in Table 3.2. The functions chosen were similar to the operations used in the hand-coded ALIVE solutions.

The *pt+* and *pt-* functions simply add their two arguments (points) together and return the resulting point. A simple vector sum is used.

Name	*Type*	*Description*
centroid	**point**	Centroid of silhouette
tl	**point**	Top left point of silhouette bounding box
br	**point**	Bottom right point of bounding box
0%, 10%, . . ., 100%	**percent**	Discrete floating point values

Table 3.1: Terminals Used in Functional Approach

Name	*Return type*	*Arguments*
pt+	**point**	(**point** *p1*, **point** *p2*)
pt-	**point**	(**point** *p1*, **point** *p2*)
point-between	**point**	(**point** *p1*, **point** *p2*, **percentage** *c1*, **percentage** *c2*)
find-bottom-edge	**point-list**	(**point** *p1*, **point** *p2*)
find-top-edge	**point-list**	(**point** *p1*, **point** *p2*)
leftmost-point	**point**	(**point-list** *pl*)
rightmost-point	**point**	(**point-list** *pl*)
average-point	**point**	(**point-list** *pl*)

Table 3.2: Functions Used in Functional Approach

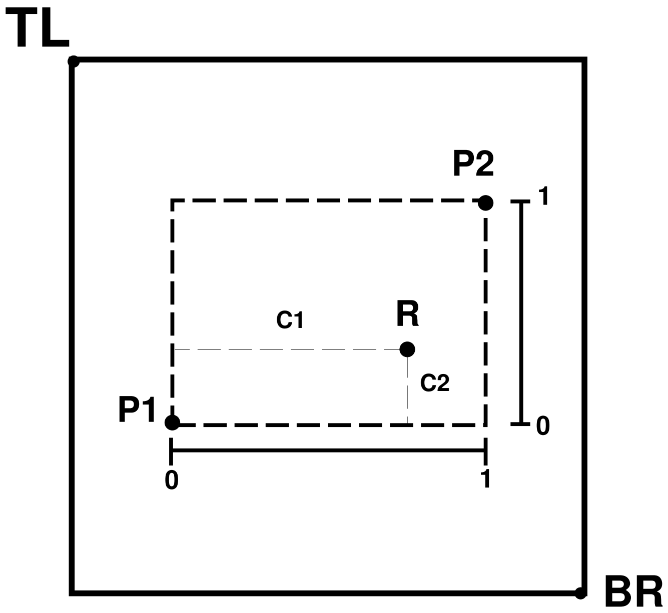

Figure 3-3: The *point-between function*. Given points P1 and p2 and percentages c1 and c2, the function returns R, the point described by c1 and c2 in the normalized coordinate system defined by P1 and P2.

The *point-between* function allows the system to select points within the area of the image more easily than using the addition and subtraction operators, which often create points outside the image in random programs. *point-between* has four arguments — two points (p_1 and p_2) which define a rectangle and two percents (c_1 and c_2) which describe a point in the normalized coordinate system defined by that rectangle (see Figure 3-3). c_1 defines the x-coordinate and c_2 defines the y-coordinate. Explicitly, if we consider the points p_1 and p_2 as the column vectors $\mathbf{P}_1$ and $\mathbf{P}_2$, the return value equals

$$\mathbf{P}_1 + \begin{bmatrix} c_1(P_{2,x} - P_{1,x}) \\ c_2(P_{2,y} - P_{1,y}) \end{bmatrix} \tag{3.3}$$

The *find-bottom-edge* and *find-top-edge* functions are similar to each other. They each take two points, which define a rectangle in the image, and each return a list of points. Specifically, *find-top-edge* returns all points p_i in the specified rectangle which have the *best* (not necessarily maximal) correlation with a top-edge template. A template is a small bitmap which defines a pattern of bits. This template is conceptually layed over the image bitmap, centered around a particular point p. The correlation at point p in the image is defined as the number of pixels where the template and image match (see Figure 3-4). The correlation is calculated at every point in the image by centering the template on each point respectively. This operation of sliding a template around an image is also known as a *convolution*.

Formally, the correlation at point $(x,\ y)$ in the image given a template of width m and height n indexed in the range $[1,\ m]$ horizontally and $[1,\ n]$ vertically is:

$$\mathbf{corr}(x, y) = \sum_{i=-\lfloor\frac{m}{2}\rfloor}^{\lceil\frac{m}{2}\rceil} \sum_{j=-\lfloor\frac{n}{2}\rfloor}^{\lceil\frac{n}{2}\rceil} \mathbf{match}(x + i,\ y + i,\ i + 1 + \lfloor\frac{m}{2}\rfloor,\ j + 1 + \lfloor\frac{n}{2}\rfloor) \tag{3.4}$$

where $\mathbf{match}(x, y, i, j)$ is defined as

$$\mathbf{match}(x, y, i, j) = \begin{cases} 1 & \text{if } image(x, y) = template(i, j) \\ 0 & \text{otherwise} \end{cases}$$

and the $image$ and $template$ functions return the value of the bit at that location in the image or template, respectively. If a point outside the image area is indexed, that point is

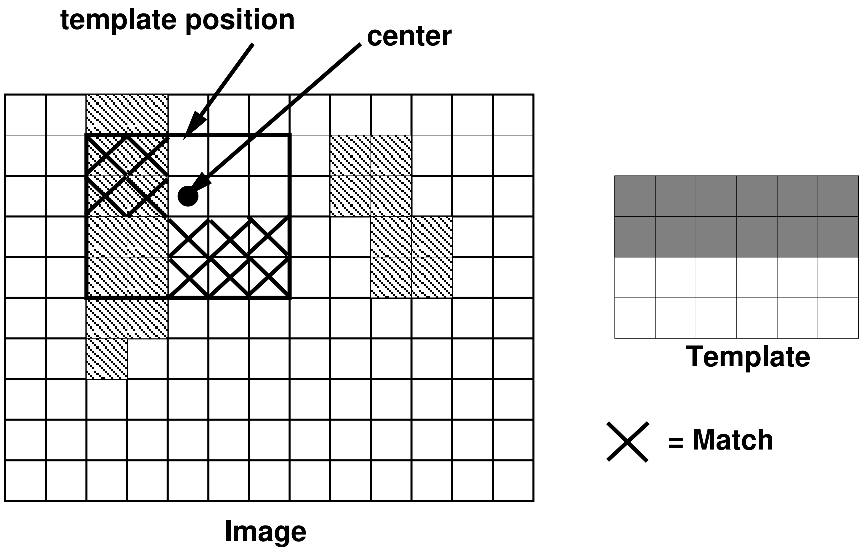

Figure 3-4: The template is layed over the image, centered on a point, and matching bits counted. The number of matches is the correlation at that center point. Here the correlation is 8.

considered off, or to have a value of zero. Figure 3-5 shows the correlation of one of the fitness cases with the top-edge template. Lighter areas are a better match for the template, and dark areas are a negative match.

Once the correlation has been calculated at each point in the region, the maximum correlation value is found over all the points. Finally, all points having that maximal value are returned in a **point-list**. Note that this may not mean that the points are a perfect match to the template, just the best match in the window of the image. For example, if the maximum correlation was eight, and there were thirteen points having this value, all thirteen points would be returned in the list. As a boundary condition, if the rectangle is degenerate and contains no points, the top-left point of the rectangle is returned arbitrarily since the function is required to return a point. The *find-bottom-edge* function is identical except that it uses a bottom-edge template.

The actual templates used were four pixels by five pixels and defined a pattern which corresponded to top and bottom edges (see Figure 3-6). The grey pixels match figure in the silhouette and the blank pixels match background. The size of the edge templates was chosen in the original ALIVE system because hands were about five pixels wide. Thus, the maximal correlation in a subwindow that contained only an arm and a hand would often

Figure 3-5: Output of the correlation function with a 4x5 top edge template.

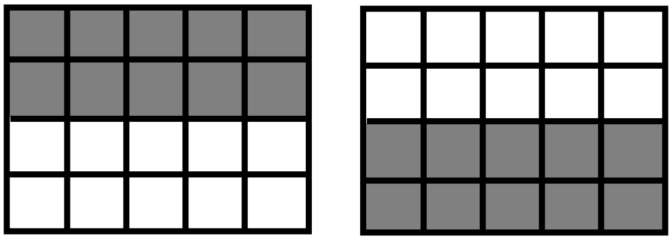

Figure 3-6: The top and bottom edge templates.

appear at only one point — the bottom of the hand. The hand-coded solutions relied on this fact.

The *leftmost-point*, *rightmost-point* and *average-point* functions operate on lists of points. They perform the obvious calculations — calculate the leftmost point in the set, the rightmost point or the average of all points in the set respectively.

3.3.4 Selection of Primitives

The primitives described in this section were chosen because they were similar to the primitives used in the ALIVE hand-finding programs. They are a bit higher in level, however. For example, the ALIVE hand-coded solutions had to implement the point filter by looping over a list of points themselves rather than having a point filter primitive. Thus, the primitives were chosen *with a sketch of a solution using them in mind*. This is a common criticism of GP — that the selection of the primitive set essentially has already solved the problem. We discuss this in greater depth in Section 5.10. However, even having a sketch of a solution in mind with these primitives, it still was the case that it took a human a while to find a good program. Additionally, GP found a better one. These results appear in 4.2.

3.3.5 Implementation Details

The original ALIVE implementation ran on a parallel vision system which made the correlation function extremely fast to compute. Unfortunately, the parallel system was unavailable for the GP runs. Instead, the correlation function was implemented serially, which is computationally expensive. It must perform O($n \cdot m$) operations, where n is the number of pixels in the image and m is the number of pixels in the template. If the actual correlation function were computed each time an evolved individual was evaluated, the GP would take days to run on even a blazing processor. Noticing that the template size is fixed, however, the correlation map for each fitness case will be constant. The system implementation takes advantage of this fact by caching the correlation map for each fitness case before the GP run starts. During the GP run, the system only has to perform O(n) calculations to calculate the maximum and return the points. This is a large savings (20 times since $m = 20$).

It is important to mention that using a serial implementation for the GP runs does not reduce the real-time, on-line ability of the programs evolved. The programs can be compiled onto a parallel machine for actual use in a real-time system.

These primitives, without conditionals, state or flow control, are sufficient to find hands. Results of experiments using these primitives with GP are presented in Section 4.2. The curious reader may skip ahead to that section, then return to this point to explore the second approach to the problem.

3.4 Imperative Approach

A second approach, more in the spirit of Ullman's visual routines theory, is to have a base representation and a set of intermediate objects which can be side-effected by the primitive set. This section describes one such set of primitives which we will call *imperative* primitives. The term *imperative* applies to programming languages and styles which involve assignment to variables and side-effects to a global environment. This is very different from a purely functional set of primitives, as described in Section 3.3. Motivation for this particular choice of primitives will be presented in Section 5.10.

type	*description*
boolean	true or false
direction	vector to describe a ray
property	image property in the base representation
percent	floating point value in interval $[0,\ 1]$
marker	register that references an image location.

Table 3.3: Types used in the imperative approach.

3.4.1 Types

Table 3.3 summarizes the types created for use in the imperative system. Many of them are labels or parameters to functions which affect the function's operation. For instance, the **property** type describes base representation properties, such as top edges, bottom edges and figure. It is used by functions which reference image properties. A property essentially describes a boolean predicate which can be applied to an image location. The location either "has" or "does not have" that property. These properties are described in more detail in Sections 3.4.2 and 3.4.3. The **boolean** type is obvious. The **direction** type is used to implement operations similar to Ullman's ray-scanning routines (see Section 2.1.1). The **percent** type is like that described in the functional section — it describes real values in the interval $[0,\ 1]$. The **marker** type is a marker as described by Ullman — a way to remember or specify a location.

3.4.2 Terminals

Table 3.4 summarizes the set of terminals used in the imperative system. The only terminal which needs some explanation is the *null* property, the last one. This was actually named retroactively since it turned out that there was a bug in the first implementation of another property finding function, which caused it to match no points. Rather than redo the experiments, it was left in for the rest. Reasons why it may be useful are described in Chapter 5.

name	*type*	*description*
true	**boolean**	boolean true for conditionals
false	**boolean**	boolean false for conditionals
M1, M2	**marker**	marker (point) register with read/write ability.
up	**direction**	direction pointing up in imagespace.
down	**direction**	direction pointing down in imagespace.
left	**direction**	direction pointing left in imagespace.
right	**direction**	direction pointing right in imagespace.
top-edge	**property**	point has maximal match with top-edge template
bottom-edge	**property**	maximal match with bottom-edge template
left-edge	**property**	maximal match with left-edge template
right-edge	**property**	maximal match with right-edge template
figure	**property**	point is in figure of silhouette
null	**property**	no points match this property

Table 3.4: Imperative approach terminals

.

3.4.3 Visual Routine Environment: Base Representation and Intermediate Representations

Following Ullman, the imperative system is divided into two different conceptual pieces — the *base representation* and the *incremental representations*, or *intermediate objects*. The base representation is created (conceptually at least) bottom-up in parallel and before the specific serial visual routines are applied. These representations form an *environment* which the visual operators will effect and refer to. The environment is essentially a set of images (base representation) and annotations (intermediate objects). When we say an object is "in the environment," we mean that is is accessible to visual operators.

Base Representation

In our implementation, the base representation consists of a set of bitmaps which contain the output of the following property (or feature) detectors:

- Figure
- Top edges
- Bottom edges
- Right edges

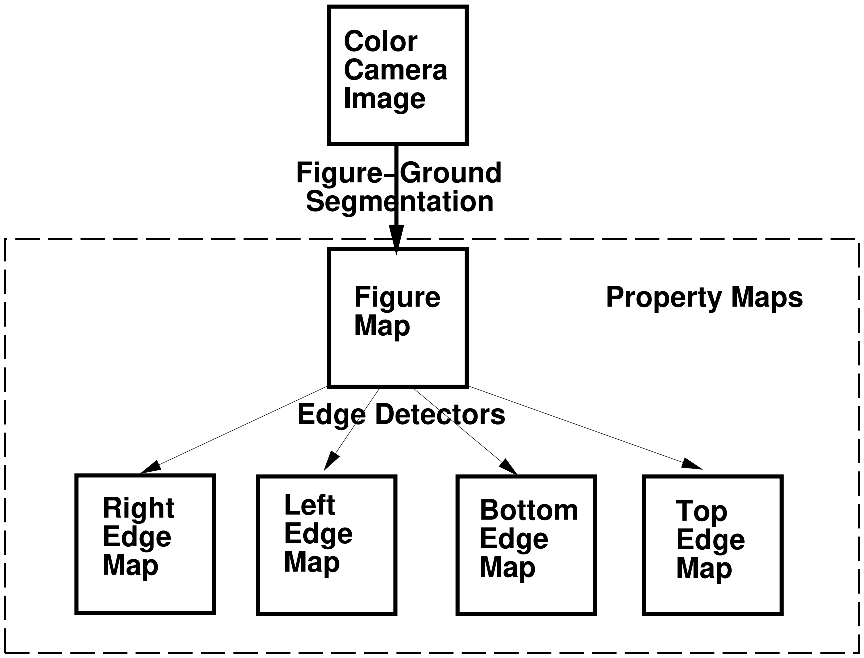

Figure 3-7: Base representation calculation.

- Left edges
- Null

To compute these maps, the figure-ground segmentation is performed first as described in Section 3.1. Then images are cropped to the bounding box of the silhouette area. Finally each of the edge detectors is applied to the segmented image (silhouette) (see Figure 3-7) and all points with maximal correlation are turned on in the property map. The edge detectors are similar to those discussed in Section 3.3.3, except that they are only two pixels by two pixels (see Figure 3-8). The other difference is that the only points which are considered as having the particular edge property are those which match the template exactly at all four pixels. All of these property maps reside in the environment and can be utilized by the visual routines.

Intermediate Objects

The intermediate representation consists of objects which can be created and effected in the visual environment. Ullman mentions markers, rays, lines and activation regions (created by the coloring operation). For our system, a slightly different set was chosen:

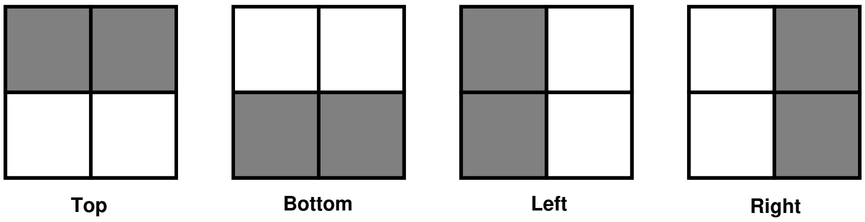

Figure 3-8: Edge templates for the imperative approach.

- *3 Markers*
- *1 Line*
- *1 Ray*
- *1 Iterator*

The activation maps did not seem useful for this problem and were expensive to compute, so they were left out.

Markers *Markers* are simply registers which contain the coordinates of a an image location. Markers are read/write — they can be moved to new locations, or just referenced. As shown above, we chose to have two markers available as terminals, or arguments to functions. In addition, a special marker is used — the *attention marker*, **A**. The attention marker is the most important intermediate object — it is implicitly effected by most of the functions and is also considered the program's answer when it is finished evaluating. Ullman does not mention an attention marker — it was added to this system following Whitehead [Whitehead, 1992].

Lines *Lines*, like markers, closely follow Ullman's proposal. A line is defined by two points and specifies the segment of locations between the two points. Since only one line was allowed in the system at a time, it was not made explicit as a terminal. Rather, functions which use a line object implicitly use the one line in the environment. If another line were allowed, a terminal would have to be created for each one so that the program could specify which one to use. Alternatively, a separate function could be created for each one, say *draw-line-1* and *draw-line-2*. Each function would then implicitly reference only its respective line.

Rays *Rays* are like lines, except that are defined by a point and a direction. A ray contains all points along the direction vector from the endpoint. Conceptually, a ray is a line with its second endpoint on the edge of the image. Also like lines, only one ray is allowed. This ray object is implicitly effected by various functions.

Iterators Finally, an *iterator* is an object used in a visual search to keep track of previously visited regions and points and to move the attention marker to the next point to be processed. Iterators were created specifically for this research and therefore require detailed explanation.

Iterators

An iterator is a special functional object that moves the attention point to successive points having a certain property. It visits each such point only once, hence the name *iterator*. Thus, iterators are guaranteed to terminate; that is, run out of points to visit. Iterators were invented to circumvent early implementation problems with infinite and effectively-infinite [3] loops in evolved programs using *return-inhibition maps* (see, for example, [Chapman, 1992]). Return-inhibition maps keep track of which pixels have been visited so that no pixel is visited twice. Chapman, for example, uses them to implement visual search operators [Chapman, 1992]. The problem is that evolved programs tend to not use them correctly, leading to infinite loops. Iterators were created to abstract the return-inhibition maps so that they were useful with GP.

A short example will help prepare the reader for the discussion of iterators. Iterators must be created and then invoked by a looping function. The imperative system has several different functions which create iterators — we focus on one for this example:

```
(connected-regions-with-property <property>)
```

There is only one looping operator:

```
(while-element <body> <stopping-condition>)
```

The following piece of code:

[3] too darn long to compute

```
(progn
  (connected-regions-with-property top-edge)
  (while-element
false
(above-marker M1)))
```

looks for the first area of top-edges that is above marker M1 or returns **false** to indicate that there are none. Figure 3-9 shows pseudocode for the iterator's operation. First, the iterator is created by *connected-regions-with-property* and placed in the environment. Next, the loop is begun. The loop binds to the only iterator in the environment and invokes it, which moves the attention marker to the first top-edge region, if there is one. If not, the loop returns **false** and exits. If so, the body of the loop — *false* — is evaluated. This is a no-operation and has no effect. Next the stopping condition is evaluated — *(above-marker M1)*. If this is true, the loop terminates with value **true**. If false, it then invokes the iterator again. This process is repeated until the iterator visits all valid points or the stopping condition is met.

Iterators are used to implement the line-scanning, ray-scanning and visual search indexing routines discussed by Ullman. Several of the function primitives described below create and add iterators to the environment, and another one allows looping over an iterator. Loop operators are in general dangerous to add to GP primitive sets, since infinite loops are common in randomly created programs. These can be patched by placing a maximum iteration count on each loop, but this is often inefficient, since most programs will then use the maximum. Iterators were specifically designed to force the loops in our system to be not just bounded, but *efficiently* bounded. In other words, it is still undesirable if loops are bounded but that bound is four hundred quadrillion iterations. By *efficient bounding*, we mean that any loop must be bounded by the number of pixels in the image. This also implies that we cannot allow nested looping. Therefore, for this research, only one iterator was allowed in the environment at a time. This is reasonable since it seems unlikely that a system should need to perform two searches simultaneously.

iterator creation and state An iterator contains state which allows it to remember which points it has visited and compute the next point it must visit. The iterator must remember what line it is searching along and what property it is searching for on that line. It must

1. Iterator $\mathcal{I}$ is created and placed in environment.
2. A looping function (*while-element*) is called.
3. While still looping:
    (a) Invoke iterator to move attention marker to next location.
    (b) If no next location, exit loop and return **false**.
    (c) Evaluate body of the loop, which is not allowed to call another loop function, since the iterator is in use.
    (d) Evaluate stopping condition.
    (e) If stopping condition is **true** then destroy iterator and return **true**. Else, continue loop.

Figure 3-9: Use of an iterator object by a loop.

also remember what points it has already visited on that line so that it does not return to them. It also has a *search state* (pristine in this example) associated with it. A visual routine can create an iterator with certain initial state, such as which line and which property, and place it in the environment to be used by a loop. A newly created iterator is said to be *pristine*.

invocation After an iterator is created, it can be *invoked* by a looping operator. A looping operator invokes the iterator it is bound to once per loop iteration. Each time the loop invokes the iterator, the iterator moves the attention marker **A** to the next valid location as specified by its state, if any. It follows that iterators can be in three search states — *pristine*, *active*, and *exhausted*. Pristine iterators have been created but not invoked. Active iterators have been invoked at least once by a loop, implying that a search is in progress. Exhausted iterators have visited all their valid locations and can index no other valid points. Pseudocode for the use of an iterator is shown in Figure 3-9.

sneaky infinite loops One problem discovered with having only one iterator is that infinite loops can still occur if a function creates a new iterator while another active iterator exists. Since there can be only one iterator, one must be removed from the environment. If the active iterator is removed, the search loop will then invoke the new iterator on the next iteration. If a new iterator is created like this each iteration, the loop will be infinite — each time through the loop the iterator state is reset by making a new one. The correct answer is that the new iterator is not placed in the environment and the function trying to create it fails.

preemptive looping Notice that although only one iterator can exist in the environment, a program can have several calls to loop functions. Sometimes they bind to different iterators (i.e. one loop call is not in the body of another) and sometimes they bind to the same one (for example, when the body of one loop contains another loop). The final problem with iterators was that a new loop function could be evaluated while another was in effect — that is, in the body of the outermost loop. While this does not produce an infinite loop, it would ruin the idea that the upper search loop was performing a sequential search, since the lower search would preempt its authority and exhaust the iterator itself. Although this is probably not catastrophic, it would make programs harder to analyze. One solution is to bind an iterator to the instance of the looping operator that first invokes it and then just ignore any other function's request to invoke the iterator, similarly to above. For no particular reason, a different approach was taken: if an instance of any other function besides the one that first activated the iterator attempts to use it, a signal is raised which stops the computation at that point and causes its fitness to be calculated immediately.

exception exploitation Exceptions can be exploited by the evolution in the following manner: let us say that the current point (location of the attention marker) is the point being looked for by the search — the answer. Normally, loops have stopping conditions associated with them. Thus, the program at that point should stop the loop. Unfortunately, it is very likely that code evaluated after the loop will then move the marker away from the answer. If, however, the program *intentionally* calls a second search loop and causes an exception, the program will be evaluated for fitness immediately, with no chance for later code to break things. This is a surprising side-effect of the implementation which was noticed after the fact.

3.4.4 Functions

Name	*Return type*	*Arguments*
	predicates and booleans	
not	**boolean**	(**boolean**)
has-property	**boolean**	(**property**)
is-on	**boolean**	()
right-of-marker	**boolean**	(**marker**)
left-of-marker	**boolean**	(**marker**)
above-marker	**boolean**	(**marker**)
below-marker	**boolean**	(**marker**)
	non-iterator environment effectors	
move-attention	**boolean**	(**direction**)
move-attention-percent	**boolean**	(**direction**, **percent**)
attend-to	**boolean**	(**marker**)
place-marker	**boolean**	(**marker**)
draw-line	**boolean**	(**marker**)
	iterator creators	
scan-in-direction	**boolean**	(**direction**)
scan-along-line	**boolean**	()
property-scan-in-direction	**boolean**	(**property**, **direction**)
property-scan-along-line	**boolean**	(**property**)
connected-regions-with-property	**boolean**	(**property**)
	control flow	
if-then-else	**boolean**	(**boolean** cond, **boolean** if, **boolean** then)
progn	**boolean**	(**boolean** a1, **boolean** a2, . . . , **boolean** a10)
while-element	**boolean**	(**boolean** body, **boolean** stopping)

Table 3.5: Imperative Functions

The imperative function set is summarized in Table 3.5 [4]. The table is divided into sections which divide the functions conceptually. Notice that all functions return the same type — **boolean**. This is possible since the "answer" of the program is considered to be the value of the attention marker **A** when the program exits. The actual root return value is ignored for fitness calculations. This does not mean the the system is closed under type, since the arguments are not all boolean.

Predicates and booleans

First in the list are the predicates and boolean functions. Currently the only boolean function is *not*. The *and* and *or* functions were left out since it is possible to use nested *if-then-else* functions to emulate their behavior.

The *has-property* function returns whether or not the attention point has the property

[4] Notice that some "functions" have no arguments. Strictly, these should be called "terminals," since they terminate the tree at that point. Since they perform computation, however, I choose to put them in the function set. This makes it easier conceptually to describe the system. I use "terminals" to decribe variables and other objects which perform no computation.

specified by the argument. The function queries the appropriate property map in the base representation.

The *is-on* function is a redundant function which tells whether the attention point is inside the figure (silhouette) or not and is identical in function to *(has-property figure)*.

The four relational predicates *left-of-marker*, *right-of-marker*, *above-marker* and *below-marker* return whether or not the attention marker is left of, right of, above or below the marker given in the argument.

Environment effectors

Next is the set of functions that effect the environment, but do not create iterators. All of these functions return true arbitrarily. They all also implicitly reference the attention marker.

The *move-attention* function moves the attention marker one pixel location in the direction specified by the argument. This is clamped to remain inside the image region — the marker cannot move outside of it.

The *move-attention-percent* functioned was added in order to allow larger jumps of the attention marker than one pixel. It moves the marker in the direction specified by the direction argument by a percentage given by the percent argument. This percentage specifies how far towards the edge of the image to move. That is, how far along the remaining distance to the edge to move. For example, moving 100% to the right would put the marker on the right edge of the image and moving 0% in any direction will not move it at all.

The *place-marker* function sets the position of the marker argument to the current attention location in order to specify that point.

The *attend-to* function is the inverse of *place-marker*. It moves the attention marker to the point indexed by the marker argument.

The *draw-line* function replaces the current line object in the environment with a new line. The new line is specified as going from the attention marker to the marker given in the argument.

There is no function to explicitly create a ray. Rays are created implicitly by the *scan-in-direction* functions defined below.

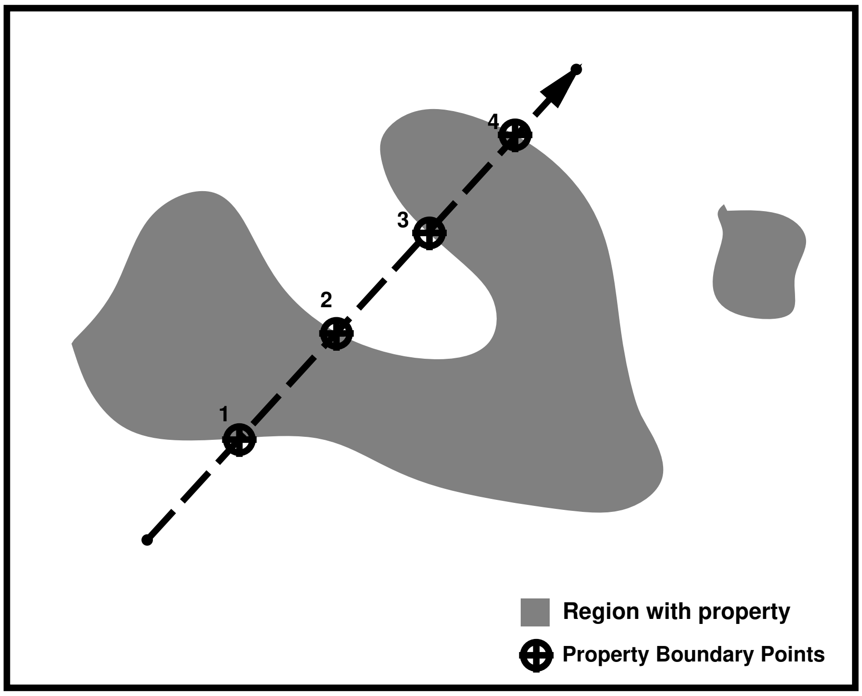

Figure 3-10: The *property-scan-along-line* function

Iterator creators

The iterator creating functions do just that — create iterators. Much like *draw-line*, they do not really perform any computation. To simplify the discussion, the operation of each function's iterator will be described along with the function, since there is a one-to-one correspondence between them — each function creates a special type of iterator.

The *property-scan-along-line* function creates an iterator that scans along a line for boundaries of regions composed of the given property. The function uses the *line* intermediate object from the environment as the line to scan along, if one has been created already. If no line exists, the function returns false and has no effect. If another iterator exists in the environment already, the function also has no effect and returns false. If there are no iterators in the environment, the function creates a pristine iterator with information about the line and property contained within it and returns true to signify it succeeded. An important point to mention here is that the line is stored by value in the iterator, not by reference. In other words, if the line object in the environment is changed while the iterator exists, the iterator's state will not change.

Figure 3-10 illustrates the *property-scan-along-line* iterator's operation. When first in-

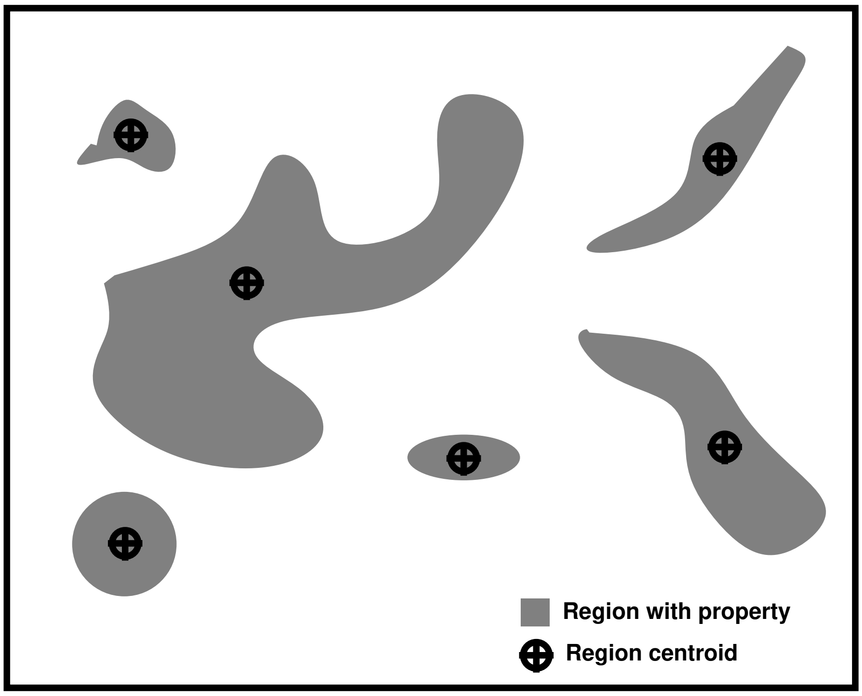

Figure 3-11: The *connected-regions-with-property* function

voked by a looping function (*while-element*, described below), the iterator scans along the line from the first point toward the second point. If it finds any point along the way with the property, it indexes that location, i.e. sets the attention marker to that location. It also remembers which point it last indexed. Once it has already found a point with the property, the next invocation will look for the next point along the line *without* the property, in order to produce the boundary finding behavior desired. This behavior continues for each call, alternating between looking for the next point with or without the property. When it reaches the other end of the segment it informs the loop that it failed. It is then marked exhausted and is destroyed. In Figure 3-10, the four points marked by crosshairs will be returned in the the order shown, one for each subsequent invocation of the iterator. On the fifth invocation, the iterator will reach the final point, inform the loop that it is exhausted and will destroy itself.

The *property-scan-in-direction* function is very similar to *property-scan-along-line*. Instead of creating an iterator than scans from one point to another, it creates an iterator that scans from a point along a cardinal *direction*. An iterator it creates also indexes boundary points in the order it encounters them, but does not exhaust itself until it hits the edge of the image. When this occurs it informs failure and destroys itself.

The *scan-in-direction* and *scan-along-line* functions are both redundant convenience functions which are equivalent to their *property-* counterparts with *figure* as the property argument.

Finally, *connected-regions-with-property* creates an iterator that indexes successive points which correspond to the centroids of all the connected components of a given property. For example, consider Figure 3-11. The iterator will index each of the centroids, signified by a crosshair, of the property components (blobs). Each blob will only be indexed once. Also, points are indexed in an arbitrary order. These iterators contain an inhibition map in order to only return each region once. Inhibition maps are described in [Ullman, 1984] and [Chapman, 1992]. They are basically bitmaps which store whether a point is inhibited or not. Points are inhibited when they are visited. Thus, an uninhibited point has not been visited.

When invoked, the iterator looks for an uninhibited point in the property map specified. It then grows a connected component by adding all uninhibited neighboring points with the property to the component, recursively. Once the blob is grown, the entire blob is inhibited, and the centroid of it is indexed. This procedure is repeated each invocation until there are no uninhibited points with the property, at which time the iterator is exhausted. An iterative connected components algorithm is described more formally in Figure 3-12. The $Neighbors$ function returns the set of points adjacent to the argument point, regardless of property or inhibition. The neighbors of a point are defined as the eight points surrounding the point, one for each of the principal eight compass directions.

Control flow functions

The last set of functions are used for program control flow. The *progn* operator is allows a block of functions to be called where normally only a single function can be called. The *progn* function evaluates each of its arguments in turn and returns as its value the value of the last argument's evaluation. The GP system used for this research allowed functions to have variable numbers of arguments, so *progn* can have from two to ten arguments. This number is chosen when an instance of a *progn* primitive is created and remains fixed in the future.

The *if-then-else* operator performs conditional evaluation. First it evaluates its first

1. Let $\mathcal{C}$ be the component and $\mathcal{S}$ a set which contains points being processed and p the point to grow the component around.

2. Initialize $\mathcal{C} \leftarrow \emptyset$ and $\mathcal{S} \leftarrow \emptyset$.

3. Add p to $\mathcal{S}$.

4. While $\exists s \in \mathcal{S}$:

   (a) Remove s from S.

   (b) If s has the property, is uninhibited, and is not already in $\mathcal{C}$ then:

      i. Add s to $\mathcal{C}$.

      ii. Inhibit s.

      iii. Add $Neighbors(s)$ to $\mathcal{S}$.

5. Return $\mathcal{C}$.

Figure 3-12: A connected-component algorithm.

argument, a condition. If the condition is true, it evaluates only its second argument, the then-clause. If it is false, it only evaluates its third argument, the else-clause. It is important in an imperative system that the evaluation is conditional. In a normal LISP evaluator, the arguments are *all* evaluated first, then the function is called on the arguments. This behavior would be devastating in a side-effecting system, where only one argument should be evaluated.

The last function is *while-element*:

(while-element body stopping-condition)

This is the loop function which was referred to in the discussion of iterators (see Section 3.4.3). When evaluated, the *while-element* operator checks to see if there is an iterator in the environment. If not, it returns false. If there is already an *active* iterator, a program-terminating exception is raised as described earlier. If there is a pristine iterator, it begins looping. A loop progresses as follows:

1. If the iterator is exhausted, destroy it and return false.

2. Otherwise, invoke the iterator to move the attention point to the next location.

3. Evaluate the body.

4. Evaluate the stopping condition.

5. If the stopping condition is false, go to 1. and continue.

6. If the stopping condition is true, destroy the iterator and return true.

The *while-element* function is designed to perform a visual search which evaluates code at each indexed point and decides whether it has succeeded (via the stopping condition) or not. Since iterators are guaranteed to be bounded and since an iterator cannot be replaced or removed during a loop except when the loop exits, looping is guaranteed to be bounded.

3.4.5 Rationale and Motivation

Contrary to the method in which primitives were chosen in the functional case, the imperative primitives that were chosen had two main properties:

- They were in the spirit of the visual routines theory, or taken directly from Ullman's suggestions

- They seemed general and reuseable for other tasks

These properties are important if GP is to be useful for solving active vision problems in general, since it is undesirable to have to write a new set of primitives for every new visual task. We would prefer to throw a huge set of common primitives at each new problem and have evolution figure out which ones to use. Ullman argued that visual routines theory addresses this problem, which is the main reason it was chosen as a basis for this research. We discuss this more in Section 5.10.

The original primitive set was created by considering the set of primitives proposed by Ullman and Chapman. The original set also had a more general looping construct which was later changed into an iterator after pilot tests failed due to infinite loops.

A hand-coded solution using the imperative primitives was actually not attempted until *after* the GP runs were performed, so it was not clear whether the set was sufficient or not at the time of its creation. In fact, it was difficult at first to find a hand-coded algorithm using them that worked at all.

3.4.6 Implementation Details

Again, the implementation had be done using a serial machine. Caching of property maps as discussed in Section 3.3.5 was also performed for the imperative system. In addition, since the property maps did not change, the connected components could be cached.

Results for experiments with this set of primitives are discussed in Section 4.3.

Chapter 4

Experiments and Results

The genetic programming system described in Section 3.2.1 was used to evolve visual routines with the functional primitives and with the imperative primitives. The main objective of these experiments was to show that GP could automatically evolve programs that were:

- Accurate
- Not catastrophic when inaccurate
- Able to generalize to cases outside the training set

Accuracy was defined as being at least as accurate as the best program a human could create in about an hour. Catastrophic failure was defined as placing the answer someplace not even close to the right answer, like on the head or the other hand. Inaccuracy can be tolerated occasionally if the misses are near misses. The ability to generalize was defined as also being able to solve cases not in the training set of fitness cases for the GP. This is an important criterion to check in all machine learning algorithms to make sure the system is not *overfitting* the samples. If these objectives could be satisfied by GP, the solutions could be used in the real ALIVE system.

An important secondary objective was to show that fairly general, low-level visual routines which seemed *reusable* could solve this task efficiently. This is important if the claim that GP is a good way of solving multiple active vision tasks more efficiently than by programming them by hand is to be demonstrated. We believe the operators are reusable since Chapman successfully used them for many different visual tasks [Chapman, 1992].

This was the driving force behind performing the imperative tests even after the functional tests were successful. This will be discussed in depth in Section 5.10.

This chapter is divided into three main sections — experimental methodology, functional results and imperative results. Results will be presented — in-depth discussion and interpretation of the results will be reserved for Chapter 5.

4.1 Methodology

The results for each series of runs are divided into seven main sections:

- Parameter values and primitives
- Hand-coded solution
- GP run results
- Best program statistics
- Non-triviality tests
- Mirror case generalization tests
- Random subsampling generalization tests

4.1.1 Parameter setting

Except where noted otherwise, parameters were chosen intuitively and not tweaked. Population size and maximum generation were chosen to make a run complete within a few hours and were chosen after one pilot run.

4.1.2 Hand-coded solution

A hand-coded program was created for each hand and for each set of primitives in order to compare with the evolved programs. These were tweaked until the author's attention span waned, usually within a half hour to forty-five minutes.

4.1.3 GP run results

Multiple GP runs were performed for each hand and for each approach in order to collect statistics on convergence rate and performance (best program found). These results are shown mainly by three types of graphs. The first type shows the sample mean and standard deviation over all the runs of the best of generation program's hits. This serves to show on average how quickly the runs converge, to what average value they tend to converge (asymptote of the mean curve) and how widely distributed the best program found is (standard deviation from the mean curve). The second graph is similar to this, but plots best of generation fitness rather than hits. Finally, some of the experiments have a histogram showing the distribution of hits for the best of run program over all the runs. This gives an idea of how likely it is that a run will produce a good program.

4.1.4 Best program statistics

The best program over all the runs is tested on the fitness cases and an accuracy assigned to it based on the percentage of fitness cases it gets hits on. Its solutions are compared graphically with the desired solutions using a picture showing each fitness case, the desired location and the program's result. The program itself is either included in the text or is referred to in the appendix if it was too long. If possible, the program is analyzed.

4.1.5 Non-triviality test

In order to show that the solution to the problem is not trivial given the primitives, a solution was searched for using pure random search rather than GP. The distribution of fitness for the random programs characterizes the *hardness* of the problem. The best results for the random search and GP can then be compared in order to show that the GP is more efficient than random search. In order to compare results, a number of random programs equivalent to the number of programs evaluated in a GP run were created and evaluated for fitness. In general, if a population of size p is run for n generations, $n \cdot p$ individuals are evaluated.

4.1.6 Generalization tests

Since it is unfair in a sense to test the evolved solutions on the training set, generalization tests were done to test whether the solutions were general or only worked on the fitness cases (this is called *overfitting* the training data). This is done by evaluating the best solutions on a *test set* which contains examples not in the training set. Two types of generalization tests were performed — *mirror case testing* and *random subsampling testing*. For some of the generalization tests, the hit constraint was relaxed from 3 pixels to 5 pixels since many of generalization misses seemed to be really close visually.

Mirror cases

Mirror cases are created by reflecting the training cases through the vertical axis. These are essentially "new" cases to the programs. This method of generalization testing has the added advantage that it allows us to compare the generalization test results with the programs specifically evolved to find the hands on the other side. In other words, testing an evolved left hand program on mirror cases effectively is asking it to find right hands in the image. Its generalization results can then be compared to the programs evolved to find right hands.

Random Subsampling

Another common method for testing generalization of solutions in machine learning is called *random-subsampling* [Weiss and Kulikowski, 1991]. Random subsampling involves dividing the full set of training samples into two sets at random — the *training set* and the *test set*. Only the training set is used during evolution. Then, the best solution is tested on the test set in order to see if it can generalize. This is done multiple times to get a distribution of generalization ability. Random subsampling is a useful method since it gives an idea of how much the training set matters in the learning — a high variance implies the algorithm is sensitive to the training set. Random subsampling is often specified by the percentage of programs used in the training set and the test set. In these experiments, 80% of the original 46 cases were randomly chosen as training examples and the remaining 20% were used as the test examples.

Experiment	*Hit Accuracy*
Left Hand Runs	
Best hand coded, fitness cases	.87
Best evolved, fitness cases	.93
Best evolved, mirrored generalization	.57
Random subsampling, .8/.2 random dataset split	0.77 $\pm$ 0.21, N=35
Right Hand Runs	
Best hand coded, fitness cases	.57
Best evolved, fitness cases	.70
Best evolved, mirrored generalization	.59
Random subsampling, .8/.2 random dataset split	.267 $\pm$ .163, N=17

Table 4.1: Summary of Results Over All Functional Runs

4.2 Functional Approach

A summary of the best results from all the runs using functional primitives appears in Table 4.1. Hit accuracy is defined as the number of hits divided by the number of fitness cases. For the random subsampling tests, the number of samples to calculate the mean and standard deviation (N) is given. Runs took on average an hour each on a Silicon Graphics $Indigo^2$ R4400 machine.

4.2.1 Run parameters

The functional system was run on the fitness cases described above, using a population size of 500. Copy percent was zero, internal crossover was 70%, global crossover was 15% and mutation was 15%. The effective copy rate was still about 10% due to failed crossovers[1], 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. The run parameters are summarized in Table 4.2. A parsimony constraint was added for these runs as well — the value of α in the fitness function (Equation 3.3) was 0.1 and the error coefficient γ was 1.0. All of these parameters were chosen intuitively and did not need to be tweaked in order to get good results, which is important since the goal is to avoid parameter tweaking. The entire set of primitives in Table 3.1 and Table 3.2 was used.

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

Run Parameters	
Population Size	500
Max generations	200
Parsimony constraint	0.1 * Size (added to fitness)
Probability of Genetic Operators	
Copying	0%
Internal Node Crossover	70%
Any Node Crossover	15%
Mutation	15%
Selection Method	
Tournament	3 programs
Creation Method	
Ramped Half-and-half	
Depth Constraints	
Maximum depth of initial tree	5
Maximum depth of mutant subtree	4
Maximum depth of program after operator	11

Table 4.2: Functional Run Parameters

4.2.2 Left Hand Runs

Hand-coded solutions

We describe the progression of programs written for this task rather than given the final program. The first hand-coded solution 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. A better solution could not be found after a half hour of fiddling.

GP run convergence

Results for the left hand runs are summarized in Figures 4-1 and 4-2. These figures represent the best of generation individual's hits and fitness, averaged over twelve separate runs. The error bars show a sample standard deviation from the mean. [2]

Best program statistics

The best evolved 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
```

[2]Notice that the best evolved program's hits actually fall within the bars, suggesting that the error bars do not give an perfectly accurate description of the distribution. This is probably due to the fact that only twelve points are used to calculate the sample standard deviation, which is not enough to produce a normal distribution.

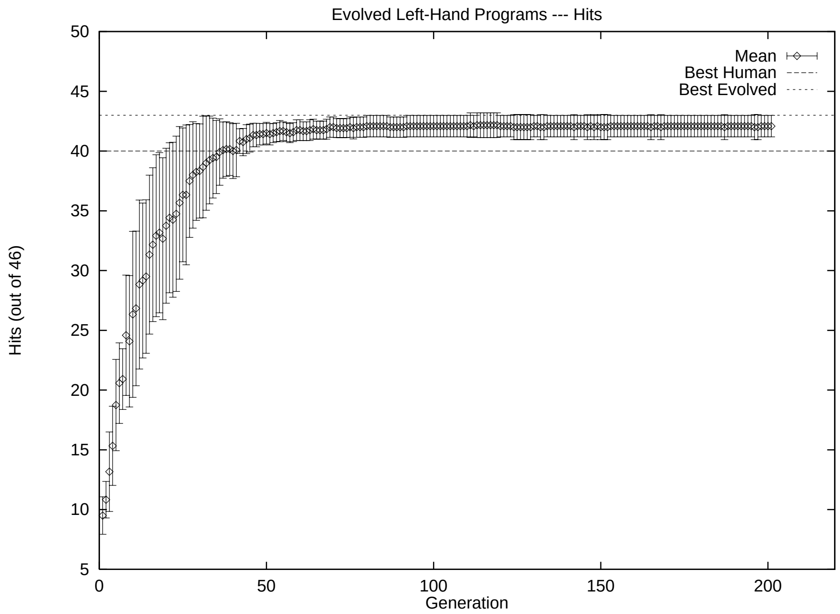

Figure 4-1: Best Functional Left-Hand Program in Each Generation — Hits: Population: 500. Copy: 0%. Crossover: 85%. Mutate: 15%. 46 fitness cases. Mean and Std. Dev. over 12 trials.

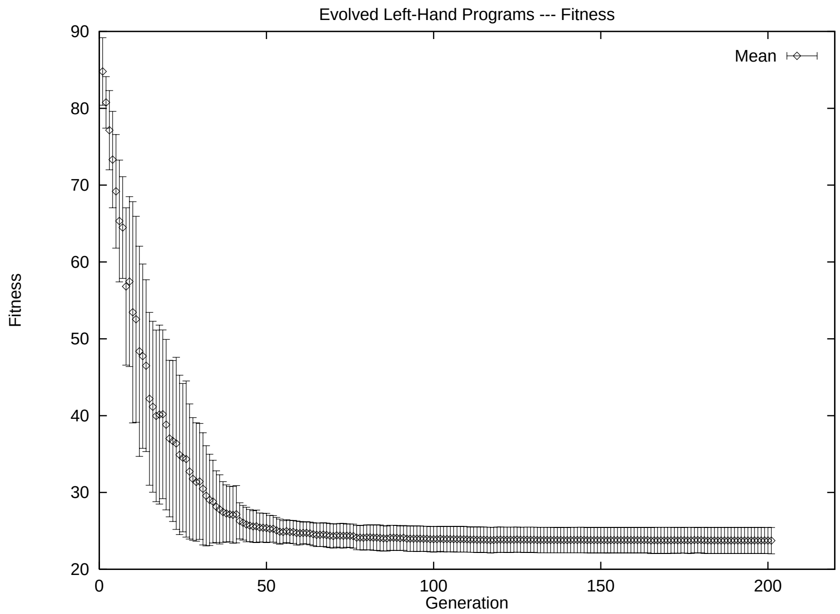

Figure 4-2: Best Functional Left-Hand Program in Each Generation — Fitness: Population: 500. Copy: 0%. Crossover: 85%. Mutate: 15%. 46 fitness cases. Mean and Std. Dev. over 12 trials.

```
                         (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 as the left-hand solutions in Figure 4-3. The program can be simplified by hand to:

```
(point-between
   (leftmost-point
      (find-bottom-edge
          tl
          (leftmost-point
             (find-bottom-edge
                 (point-between tl br 20% 70%)
 (point-between tl centroid 0% 90%)))))
    tl 30% 10%)
```

This finds the leftmost bottom edge in a small window on the central left edge of the image. It then finds the leftmost bottom edge between that point and the top left of the image. Finally it moves that point slightly left and slightly up and returns it. Essentially, this is performing a similar computation to the hand-coded solution, but with finer windows and parameters. It is also unclear how the two-layer leftmost edge-finding helps.

Proof of nontriviality

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). These results are presented in Figures 4-4 and 4-5. Note that the y-axis is logarithmic scale. Most random programs get zero hits. The best received 15 hits.

Mirror case generalization

Testing the best left-hand program on the 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 hand which receives 32 hits (70%), but is still not bad. The program is performing some generalization. Also, the right-hand fitness cases (which essentially are what the mirrored cases are) are harder than the left-hand cases. This is discussed more in Chapter 5.

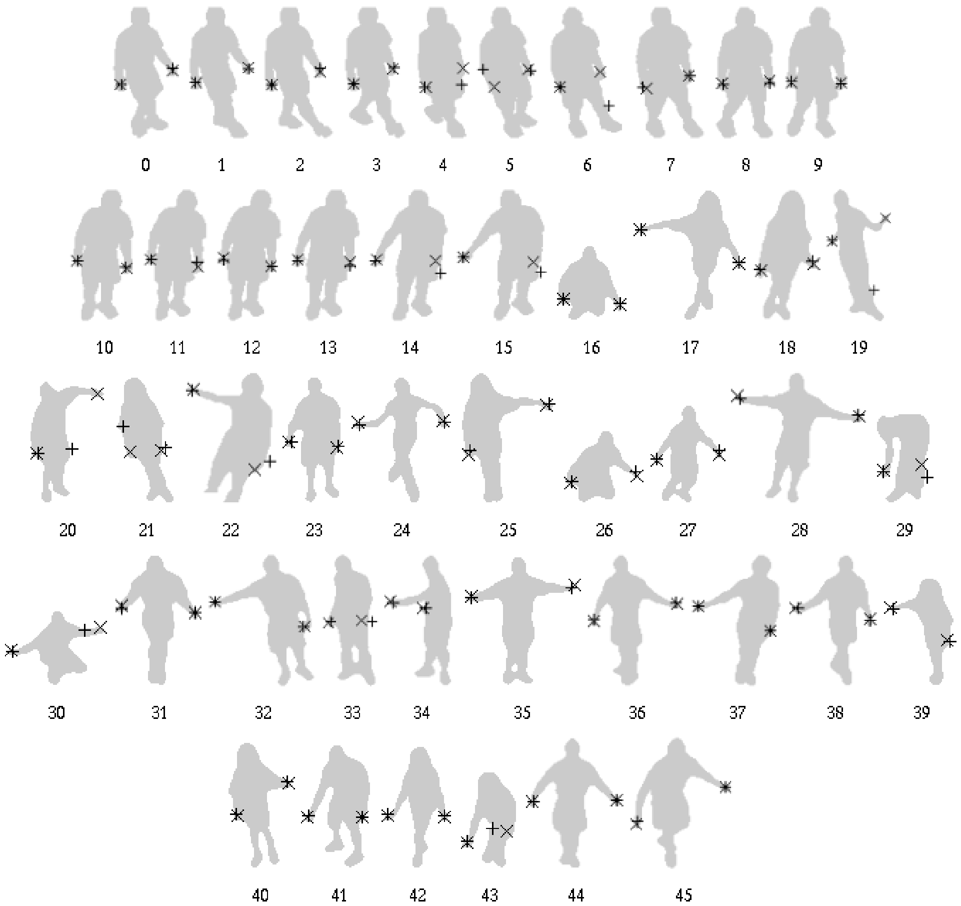

Figure 4-3: Functional Best Evolved Program Answers, Left and Right X shows the desired location. + shows the best program's result.

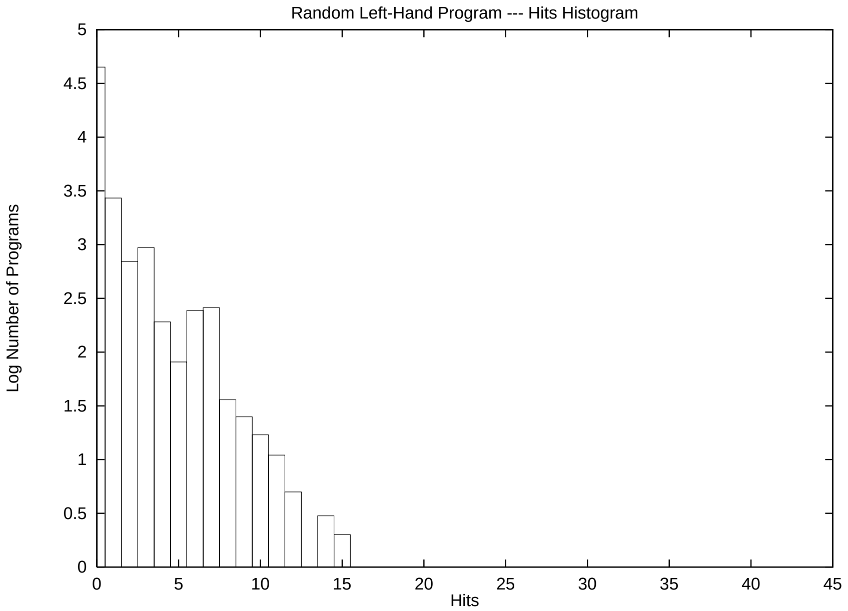

Figure 4-4: Hit histogram for 50,000 functional left-hand programs generated randomly and not evolved. Note: y-axis is log number of programs.

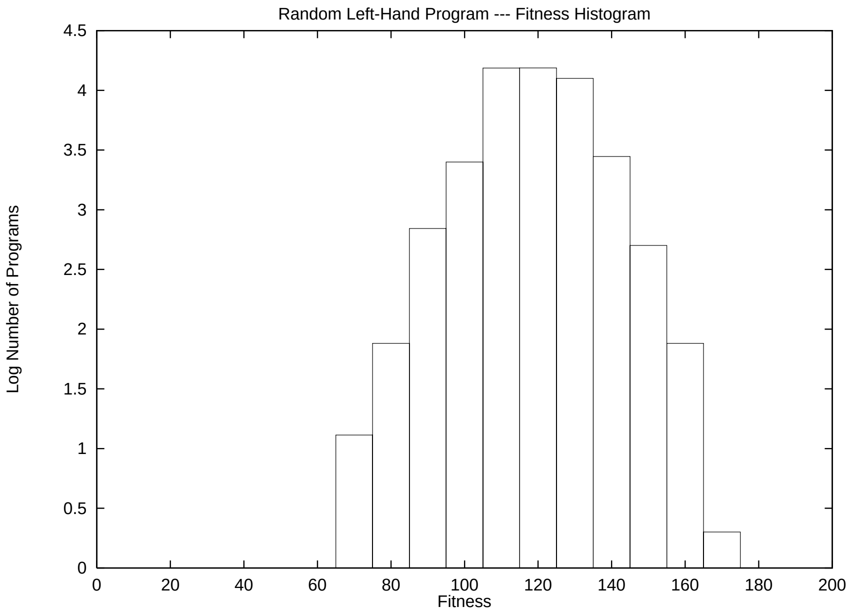

Figure 4-5: Binned fitness histogram for 50,000 functional left-hand programs generated randomly and not evolved. Note: y-axis is log number of programs.

Random subsampling

Twenty-two random subsampling runs were done 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.

4.2.3 Right Hand Runs

Hand-coded solution

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 mirror image 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.

GP run convergence

Results for the right-hand runs are summarized in Figures 4-6 and 4-7, similarly to the left hand runs.

Best program statistics

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
```

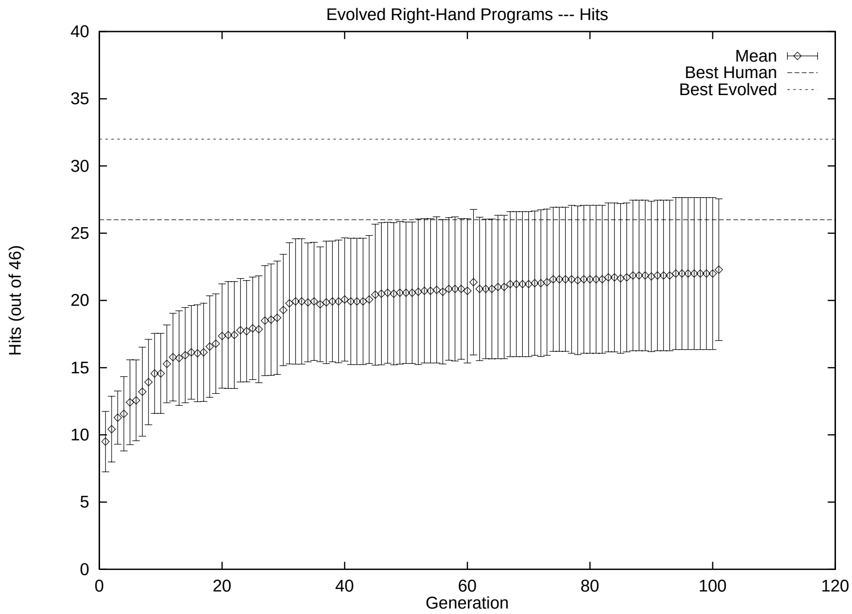

Figure 4-6: Best Functional Right-Hand Program in Each Generation — Hits: Population: 500. Copy: 0%. Crossover: 85%. Mutate: 15%. 46 fitness cases. Mean and Std. Dev. over 15 trials.

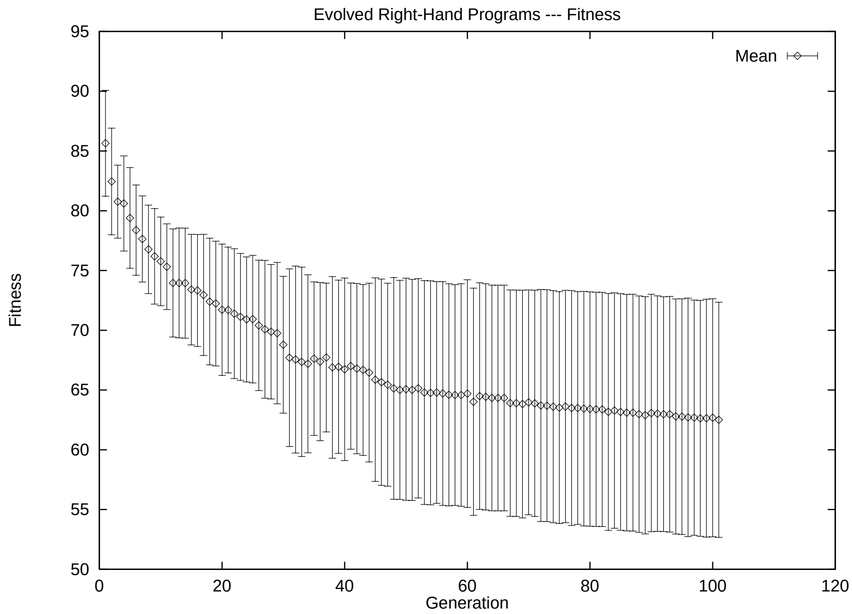

Figure 4-7: Best Right-Hand Program in Each Generation — Fitness: Population: 500. Copy: 0%. Crossover: 85%. Mutate: 15%. 46 fitness cases. Mean and Std. Dev. over 15 trials.

```
      (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 as the right hand answers in Figure 4-3. 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.

This program is large and thus harder to analyze. The fact that the topmost two functions are *rightmost-point* and *find-top-edge* implies that is it finding the rightmost top edge in a window that it defines in a complex manner (i.e. the rest of the program).

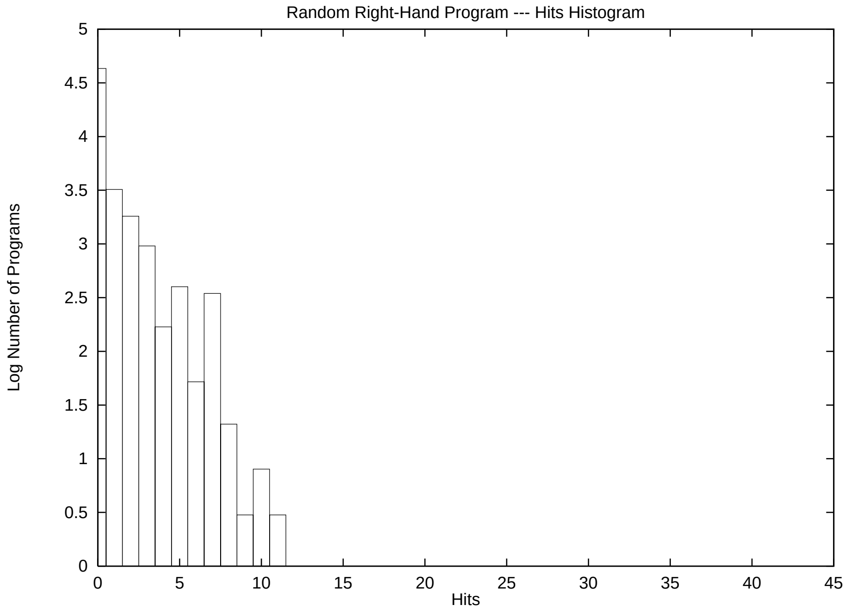

Figure 4-8: Hit histogram for 50,000 random functional right-hand programs. Note: y-axis is log number of programs.

Proof of nontriviality

50,000 random programs were also created for the right-hand programs. The histograms for these experiments are shown in Figure 4-8 and Figure 4-9. Once again, the random solutions are much worse than the evolved solutions.

Mirror case generalization

As in the left hand program, the best right-hand program was tested on the mirrored fitness cases as well. 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 is still much better than random. Clearly some generalization is occurring. A reasonable 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.

Random subsampling

Seventeen random subsampling runs were performed. These resulted in a mean accuracy of .266 (12 hits) and a standard deviation of .163 (6 hits) (.266 ± .163, N = 17).

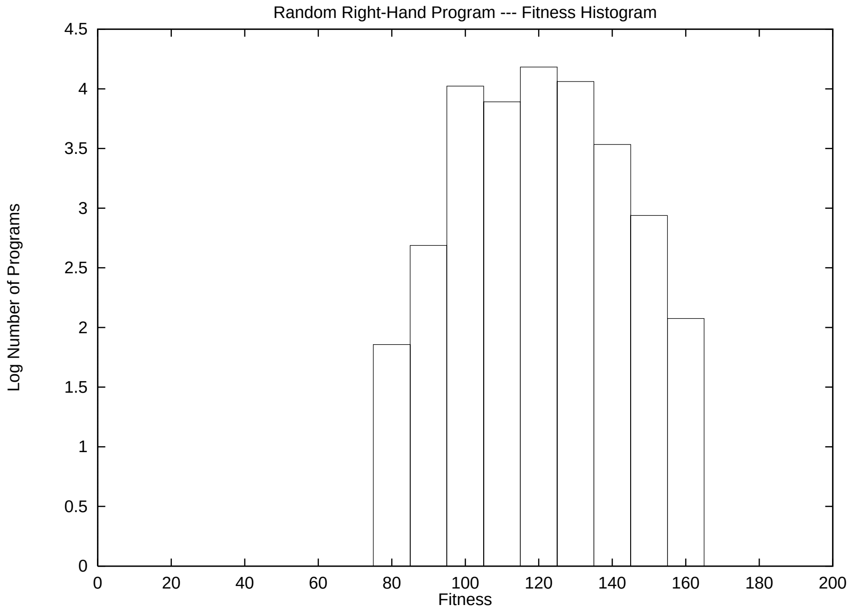

Figure 4-9: Binned fitness histogram for 50,000 random functional right-hand programs. Note: y-axis is log number of programs.

4.3 Imperative Approach

Experiments similar to those described above were done using the imperative primitives as well. Programs were evolved separately for finding the left hand, right hand and additionally for this section, the head. The results over all the runs are summarized in Table 4.3. Runs took on average several hours each on a Silicon Graphics $Indigo^2$ R4400 machine. In general, they took longer than the functional runs. This is almost definitely due to the lack of parsimony constraint for these experiments — the imperative programs tended to get very large towards the middle of the run.

4.3.1 Run parameters

The markers — including the attention point — were all initialized to $(0, 0)$, the top left corner of the image. Unless specifically mentioned, the parameters in the fitness function (Equation 3.3) were fixed over all runs described in this section. The coefficient of the error term, γ, was 1.0 and the parsimony constraint coefficient, α, was set to 0.0. The lack of parsimony constraint meant programs were allowed to grow as large as the depth constraint allowed.

Experiment	*Hit Accuracy*
Left Hand Runs	
Best hand coded, training set (hit radius 3 pixels)	.67
Best hand coded, training set (hit radius 5 pixels)	.87
Best evolved, training set	.98
Best evolved, mirrored generalization (hit radius 3 pixels)	.48
Best evolved, mirrored generalization (hit radius 5 pixels)	.63
Random subsampling, .8/.2 random dataset split	0.711 $\pm$ 0.227, N=35
Right Hand Runs	
Best hand coded, training set (hit radius 3 pixels)	.39
Best hand coded, training set (hit radius 5 pixels)	.52
Best evolved, training set	.76
Best evolved, mirrored generalization (hit radius 3 pixels)	.48
Best evolved, mirrored generalization (hit radius 5 pixels)	.85
Random subsampling, .8/.2 random dataset split	.314 $\pm$ .185, N=20
Head Runs	
Best hand coded	N/A
Best evolved, training set	.98
Random subsampling, .8/.2 random dataset split	.740 $\pm$ .111, N=10

Table 4.3: Summary of Results Over All Imperative Runs

4.3.2 Left Hand Runs

Programs were evolved for finding the left hand using the set of primitives in Table 4.4. Notably, the *move-attention-percent* was not used for these runs. It was created for the right runs since it seemed that initializing the markers to the top left made it hard for the right evolution to move the marker far enough to the right to solve the problem. Parameters for the left hand runs are summarized in Table 4.5. No mutation was allowed mainly to see if only crossover was powerful enough to solve the problem. Initial results were quite good, so mutation was left at zero.

Hand-coded solution

Programming a solution using these primitives was not obvious. The primitives were not selected with hand-finding in mind but were mainly taken from Ullman and Chapman, so the author did not have a notion of how to use them. After about fifteen minutes, the following program was written:

```
(progn
```

Functions	*Terminals*
not	true
has-property	false
is-on	M1, M2
right-of-marker	up
left-of-marker	down
above-marker	left
below-marker	right
move-attention	top-edge
while-element	bottom-edge
attend-to	left-edge
place-marker	right-edge
draw-line	figure
scan-in-direction	null
scan-along-line	
property-scan-in-direction	
property-scan-along-line	
connected-regions-with-property	
if-then-else	
progn	

Table 4.4: Imperative Primitives For Left Hand

.

Run Parameters	
Population Size	500
Max generations	200
Parsimony constraint	None
Probability of Genetic Operators	
Copying	0%
Internal Node Crossover	50%
Any Node Crossover	50%
Mutation	0%
Selection Method	
Tournament	3 programs
Creation Method	
Ramped Half-and-half	
Depth Constraints	
Maximum depth of initial tree	5
Maximum depth of mutant subtree	4
Maximum depth of program after operator	10

Table 4.5: Imperative Left Hand Run Parameters

```
(move-attention-percent right 100%)
(place-marker M1)
(connected-regions-with-property bottom-edge)
(while-element
 (progn
   (if-then-else (left-of-marker M1)
 (place-marker M1)
 true))
 false)
(attend-to M1)
(move-attention up)
(move-attention up)
(move-attention up))
```

Essentially, the function this program computes is "find the leftmost bottom-edge region, then move up a little." It initializes M1 to the right edge and then loops on all bottom-edge regions, updating M1 to always contain the leftmost of these regions. When the loop finishes, it attends to M1 and moves up a little to center on the hand. This program got 31 out of 46 hits (67% accuracy) in a hit error radius of 3 pixels, and 40 hits for a hit radius of 5 pixels (87%). Unfortunately, the program exhibits catastrophic failure on several hard cases, where it places the answer on the left foot.

GP run convergence

Results for the left hand runs are shown in Figures 4-10 and 4-11. The main curve in each graph shows the mean best of generation values — the best of generation program's hits or fitness, averaged over all runs. The error bars show a sample standard deviation from the mean. Notice that this is not necessarily an accurate measure since the maximum possible number of hits falls within a standard deviation of the mean, but it gives an idea of variation over runs. A better graph for showing performance over multiple runs is Figure 4-12, which shows a histogram of the maximum number of hits for each run.

Best program statistics

The best program found over all runs received 45 hits. This is better than the best in the functional runs, which received only 43. (A full comparison between the two styles can be found in Chapter 5). It misses only one case, where the hand is completely invisible.

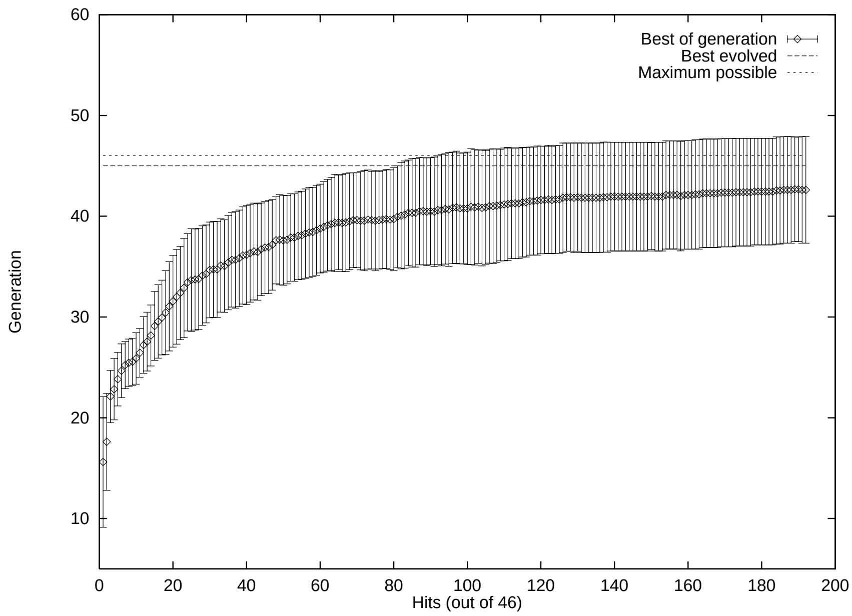

Figure 4-10: Best Imperative Left-Hand Program in Each Generation — Hits: Population: 500. Copy: 0%. Crossover: 100%. Mutate: 0%. 46 fitness cases. Mean and Std. Dev. over 19 trials.

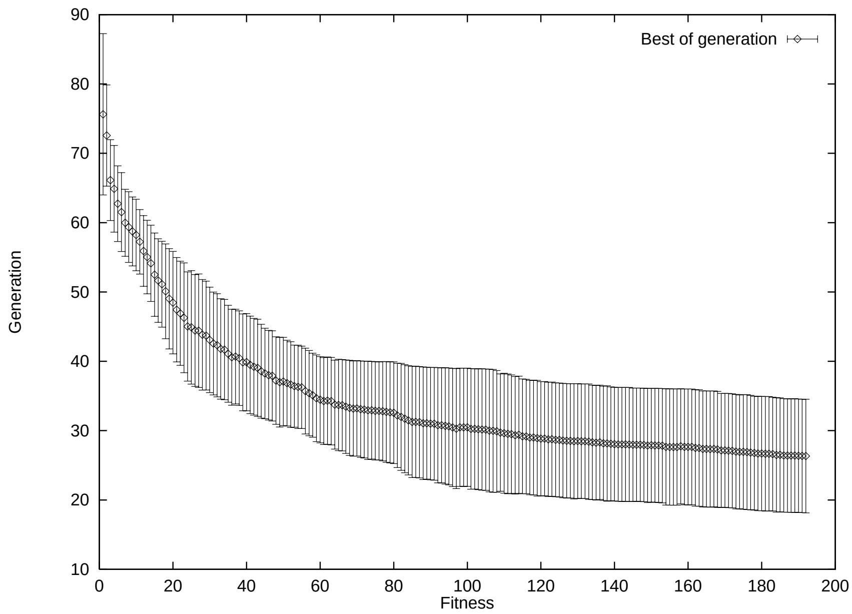

Figure 4-11: Best Imperative Left-Hand Program in Each Generation — Fitness: Population: 500. Copy: 0%. Crossover: 100%. Mutate: 0%. 46 fitness cases. Mean and Std. Dev. over 19 trials.

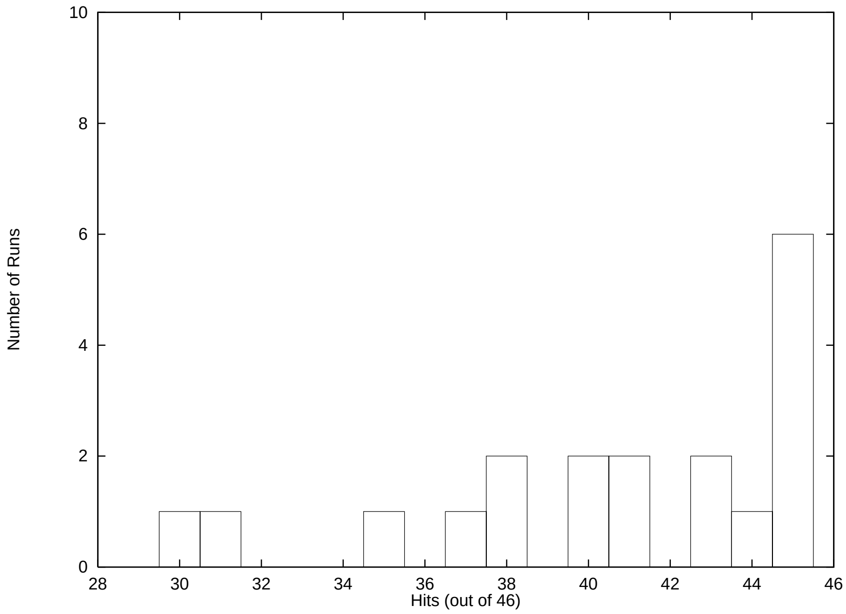

Figure 4-12: Histogram of Maximum Hits Received for Each Imperative Left-Hand Run Over 19 Runs

Figure 4-13 shows the best program's solutions. The + is the program's answer and the X shows the desired solution. The program is quite large and difficult to analyze; it is included in Section A.1.

Proof of nontriviality

As in the functional experiments, random programs were also generated in order to ensure that the problem was non-trivial and that GP was advantageous. Since the imperative runs converged more slowly, 100,000 random programs were created in order to parallel the computational expenditure of the GP — a population size of 500 times 200 generations equals 100,000 evaluations. Results are shown in the logarithmic histogram in Figure 4-14. The best found got only 16 hits. For these experiments in particular since only crossover was allowed, GP is performing quite well.

Mirror case generalization

Mirror generalization tests were also done. The best program evolved receives 22 hits, or an accuracy of 48%. Qualitatively, results are actually better than this, since the requirement

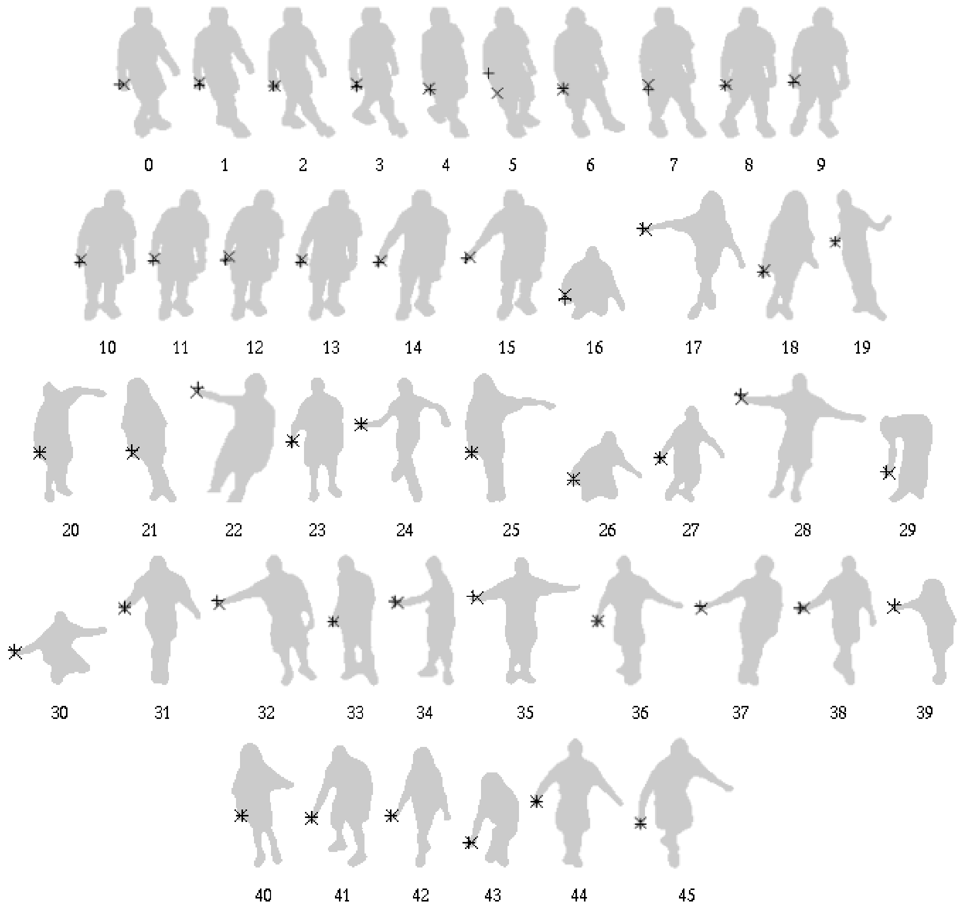

Figure 4-13: Best Evolved Imperative Left Program's Results on Training Set. X is desired, + is program result.

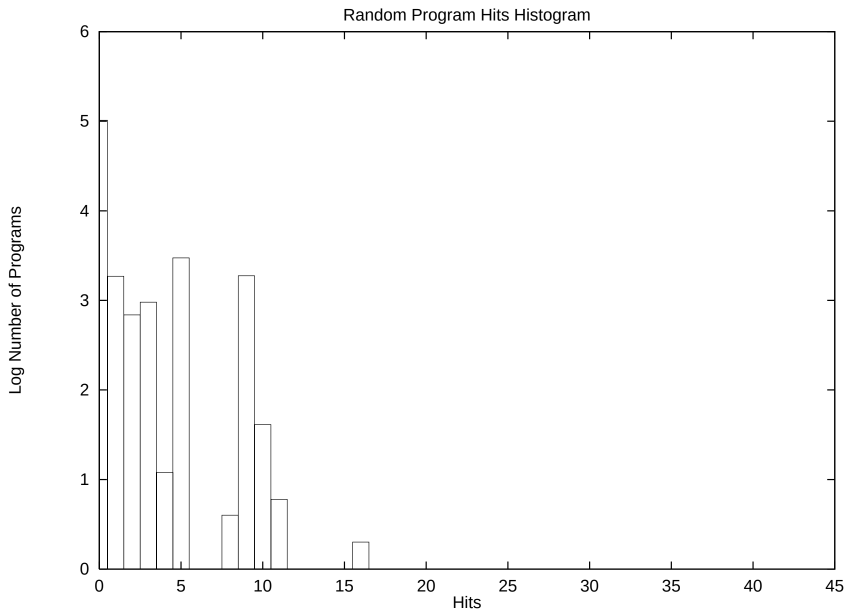

Figure 4-14: Hit histogram for 100,000 imperative left-hand programs generated randomly and not evolved. Note: y-axis is log number of programs. Maximum was 16 hits.

that a hit be within three pixels is very stringent. Testing was redone using 5 pixels as the radius for a hit — the program receieved 29 hits, for an accuracy of 63%. Unfortunately, this program misses many of the first cases, placing the marker on the foot.

Random subsampling

Over 35 random subsampling runs using 80% of the 46 fitness cases as training data and the remaining 20% as test data, a mean accuracy of 0.711 with a sample standard deviation of .227 was achieved $(0.711 \pm .277,\ N = 35)$. This corresponds to 32.7 ± 12.7 hits.

4.3.3 Right Hand Runs

For the right hand runs, the same primitives as in Table 4.4 were used, except *move-attention-percent* was added since early runs could not easily move the attention marker to the left side of the image, leading to poor initial results.

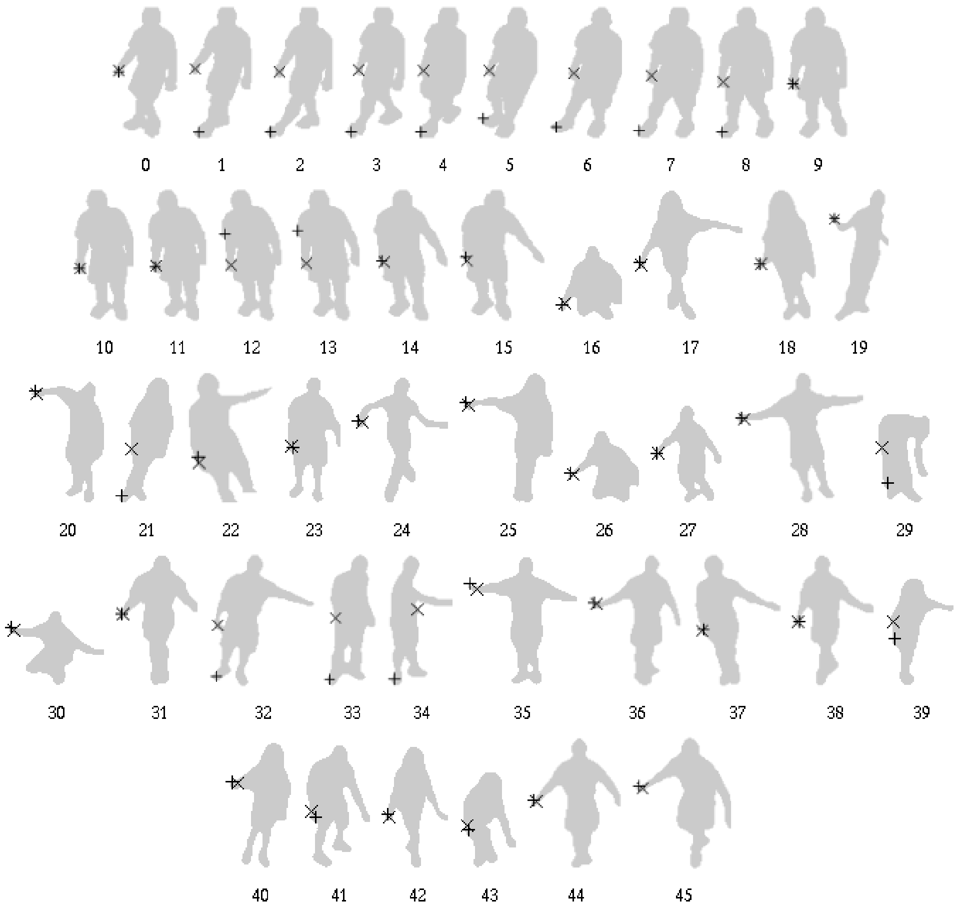

Figure 4-15: Best Evolved Imperative Left Program's Results on Mirrored Fitness Cases to Test Generalization. X is desired, + is program result.

Hand-coded solution

The mirror equivalent of the left-hand program was used for the right. Specifically:

```
(progn
  (connected-regions-with-property bottom-edge)
  (while-element
   (progn
     (if-then-else
      (right-of-marker M1)
      (place-marker M1)
      true))
   false)
  (attend-to M1)
  (move-attention up)
  (move-attention up)
  (move-attention up))
```

This program is not very good. It only gets 18 hits, or 39% accuracy.

GP Convergence

For the first series of runs, the same parameters as in Table 4.5 were used. Figure 4-16 is a histogram of maximum hits over all 20 of those runs. Since the results were not as good as expected, mutation was added hopefully to nudge the evolution out of local minima. It helped only slightly — the histogram of maximum hits for runs with mutation at 10% is shown in Figure 4-17. Plots of the mean best of generation hits and fitness for the second set of runs are presented in Figure 4-18 and 4-19. As in the functional experiments, the convergence is slower and with a wider variance.

Best program statistics

The best individual over all runs received 35 hits (76% accuracy), which is only slightly higher than the functional maximum of 32 hits. This individual's results are shown in Figure 4-20 and the actual program text is included in Section A.2.

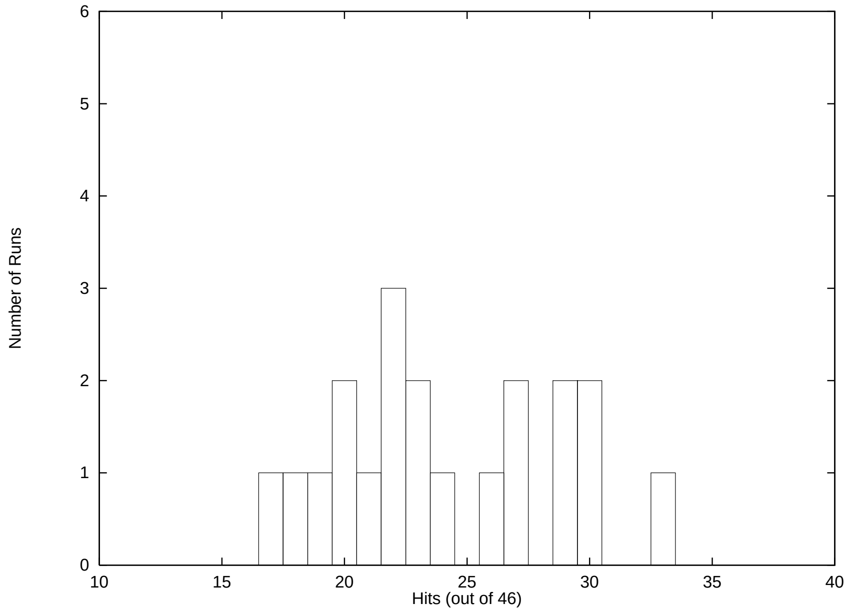

Figure 4-16: Histogram of Maximum Hits Received for Each Imperative Right Run Over 20 Runs (no mutation)

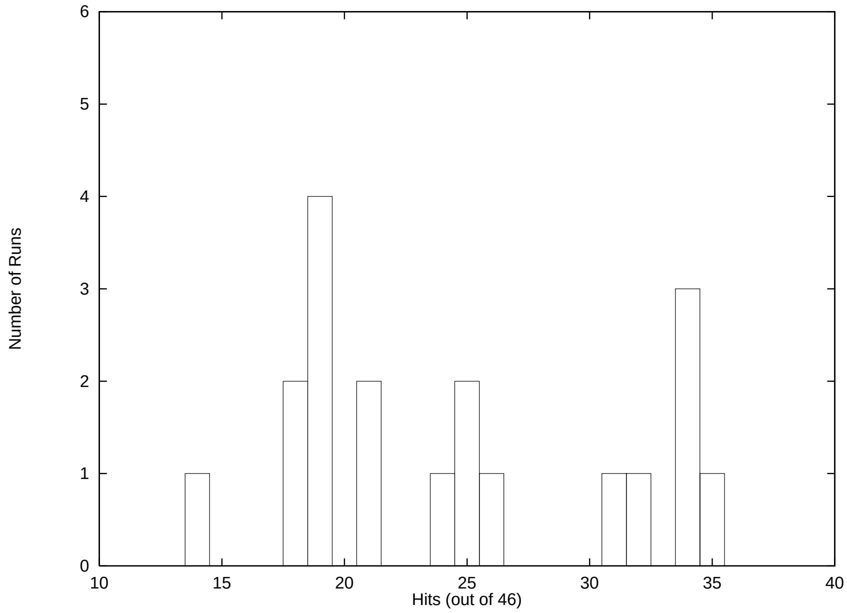

Figure 4-17: Histogram of Maximum Hits Received for Each Imperative Right Run Over 20 Runs (with mutation)

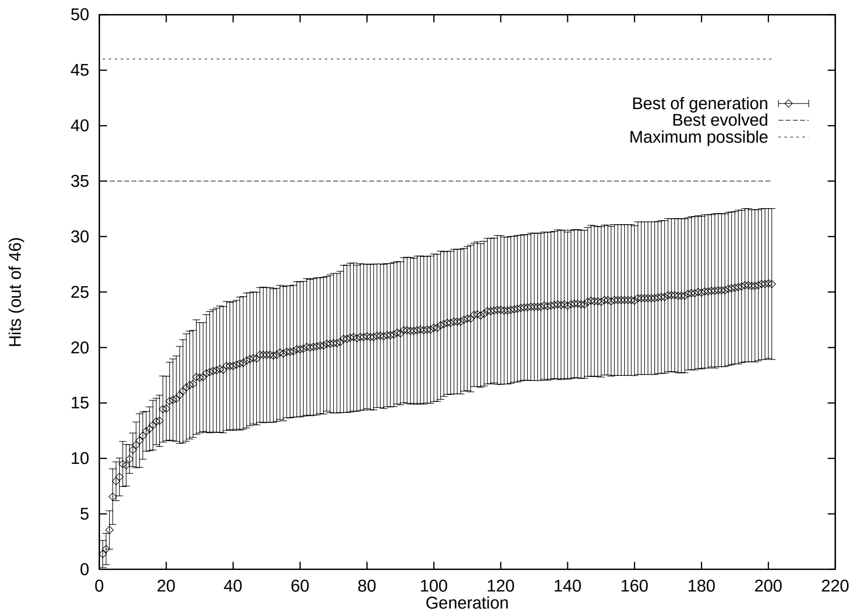

Figure 4-18: Best Imperative Right-Hand Program in Each Generation — Hits: Population: 500. Copy: 0%. Crossover: 90%. Mutate: 10%. 46 fitness cases. Mean and Std. Dev. over 20 trials.

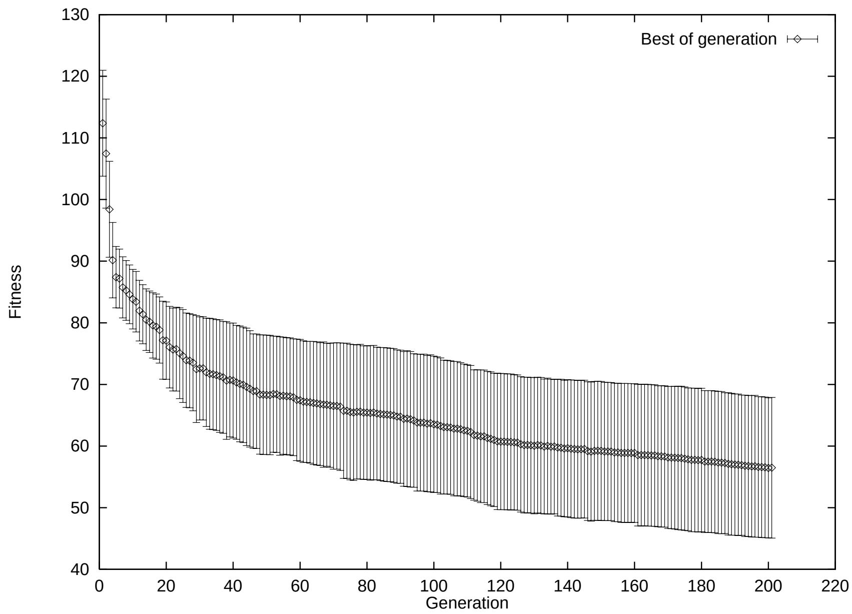

Figure 4-19: Best Imperative Right-Hand Programs — Fitness: Population: 500. Copy: 0%. Crossover: 90%. Mutate: 10%. 46 fitness cases. Mean and Std. Dev. over 20 trials.

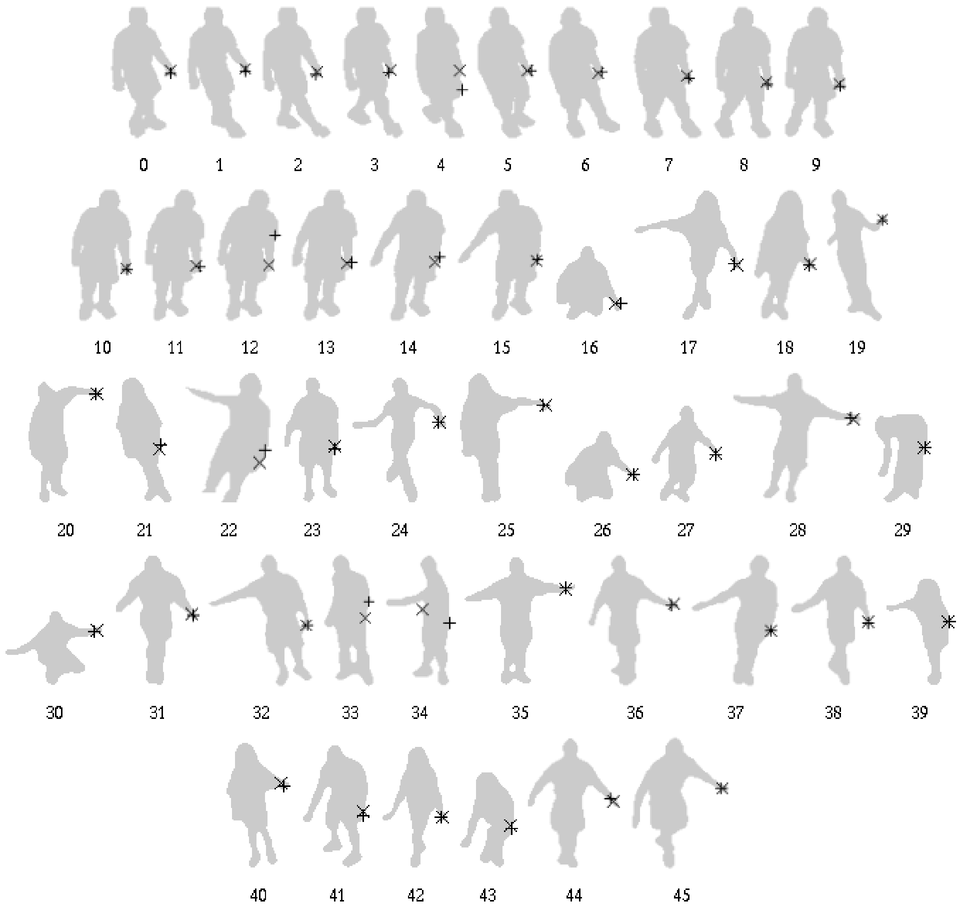

Figure 4-20: Best Evolved Imperative Right Program's Results on Training Set. X is desired, + is program result.

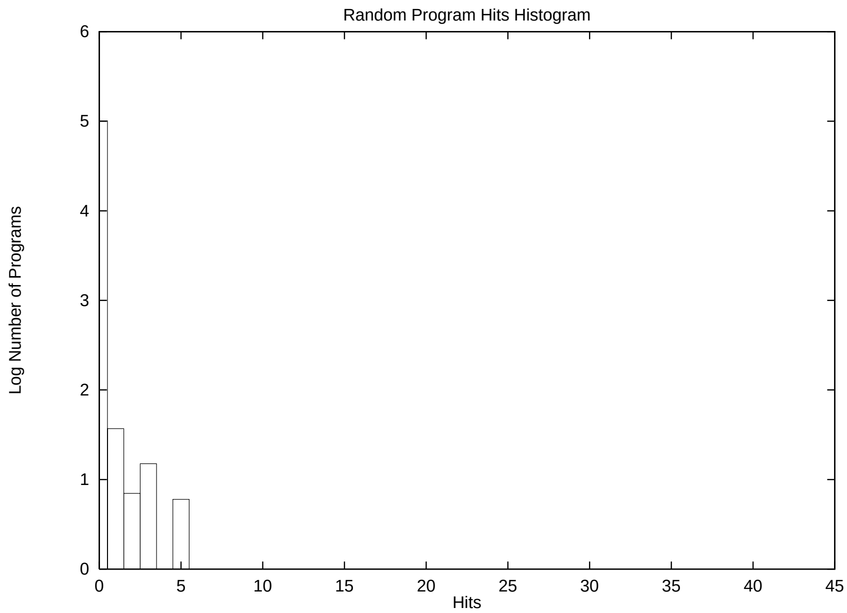

Figure 4-21: Hit histogram for 100,000 imperative right-hand programs generated randomly and not evolved. Note: y-axis is log number of programs. Maximum was 6 hits.

Proof of nontriviality

The results of evaluating 100,000 random programs are illustrated in Figure 4-21. Again, the random programs are poor, getting a maximum of six hits.

Mirror case generalization

This program was also run on the mirrored fitness cases, as above, to test for generalization. The results of this test are shown in Figure 4-22. Although it receives only 22 out of 46 hits (48% accuracy), the figure shows that it only misses the ones it misses by one or two pixels, and never fails catastrophically. Indeed, when the hit constraint is relaxed to within 5 pixels (still often withing the area of the hand), the program gets 39 out of 46 hits, or 85% accuracy.

Random subsampling

Finally, 20 random subsampling runs were done. The mean accuracy was 31% with a sample standard deviation of 19% $(.313 \pm .185,\ N = 20)$. This is equivalent to 14 ± 9,

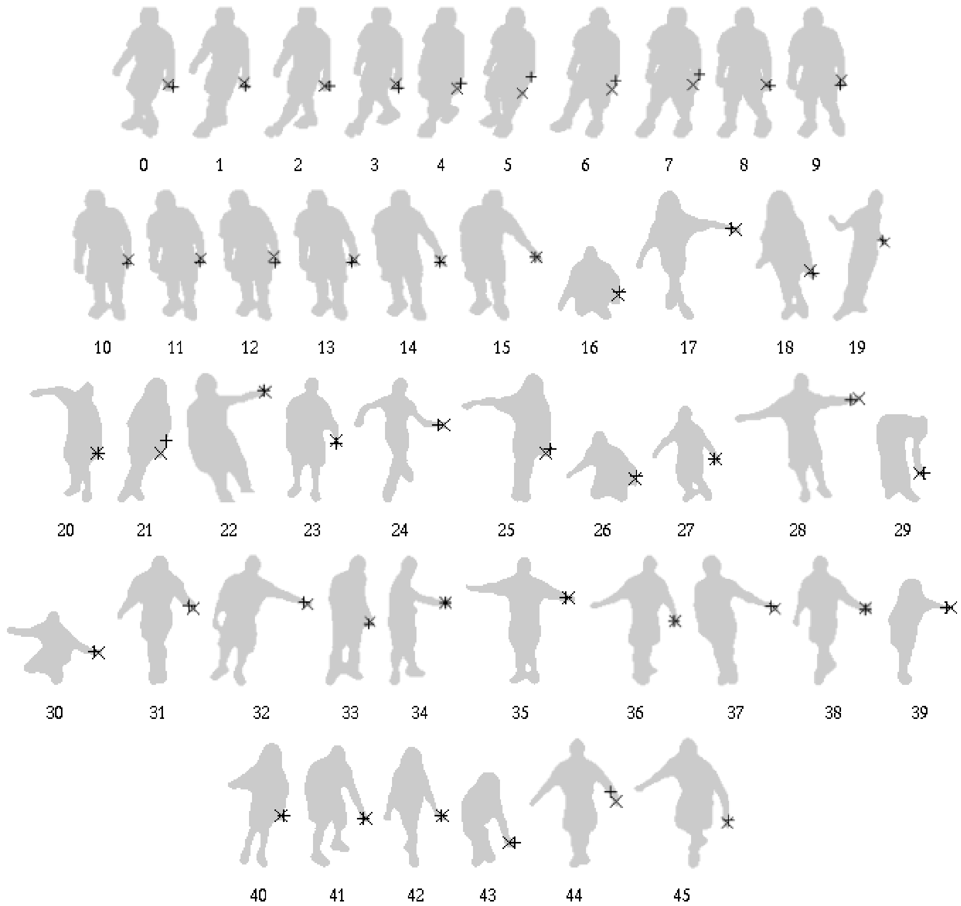

Figure 4-22: Best Evolved Imperative Right Program's Results on Mirrored Fitness Cases to Test Generalization. X is desired, + is program result.

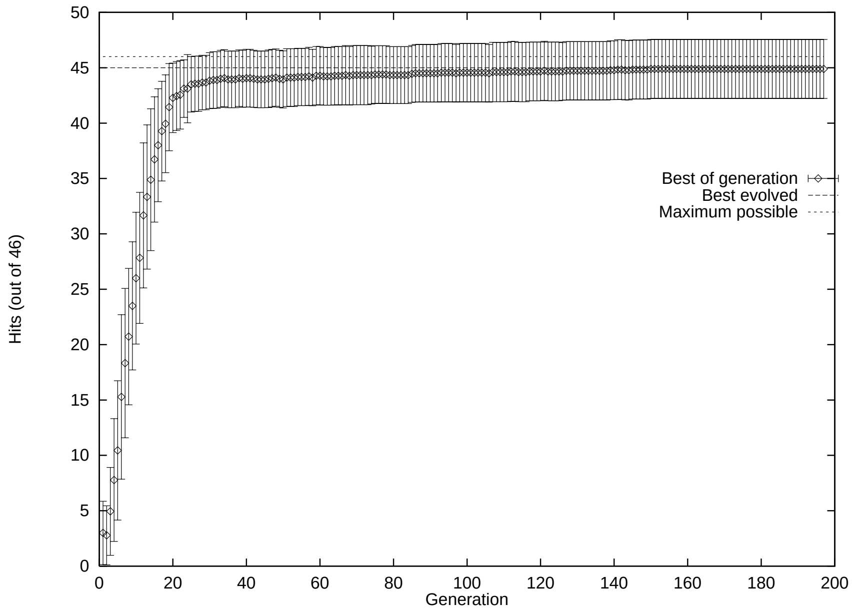

Figure 4-23: Best Imperative Head Program in Each Generation — Hits: Population: 500. Copy: 0%. Crossover: 100%. Mutate: 0%. 46 fitness cases. Mean and Std. Dev. over 20 trials.

which is not exceptional.

4.3.4 Head Runs

As an additional experiment with the imperative primitives, a set of runs was performed to find the head of the person in the silhouette. The set of primitives is the same as in Table 4.4. Parameters are identical to those in Table 4.5 — mutation was again not allowed.

GP run convergence

In the same style as the plots above, results from these runs are presented in Figures 4-23, 4-24 and 4-25.

Best program statistics

The best individual gets 45 out of 46 (98% accuracy) hits and is included in Section A.3. Its solutions are depicted in Figure 4-26. The fact that the runs converge much more quickly and with less variance implies that finding the head is an easier problem. This should

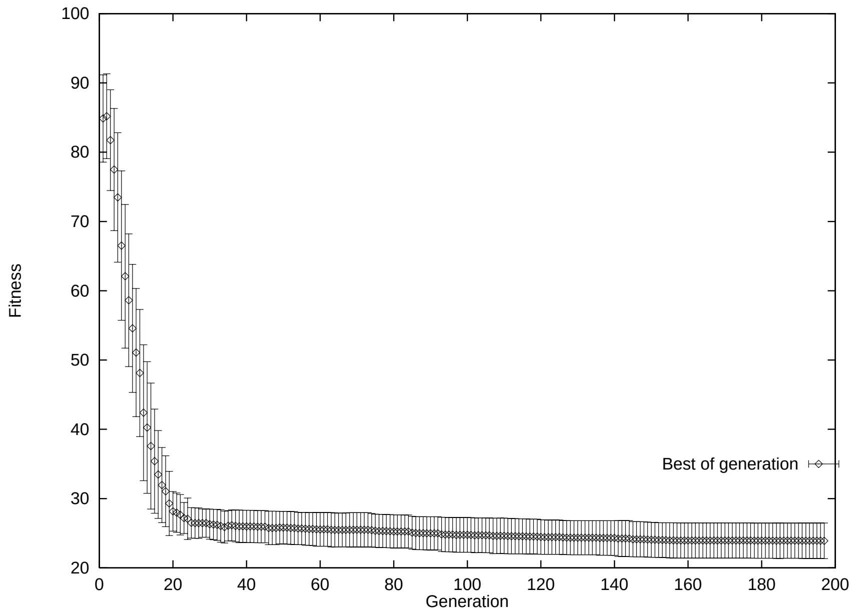

Figure 4-24: Best Imperative Head Program in Each Generation — Fitness: Population: 500. Copy: 0%. Crossover: 100%. Mutate: 0%. 46 fitness cases. Mean and Std. Dev. over 20 trials.

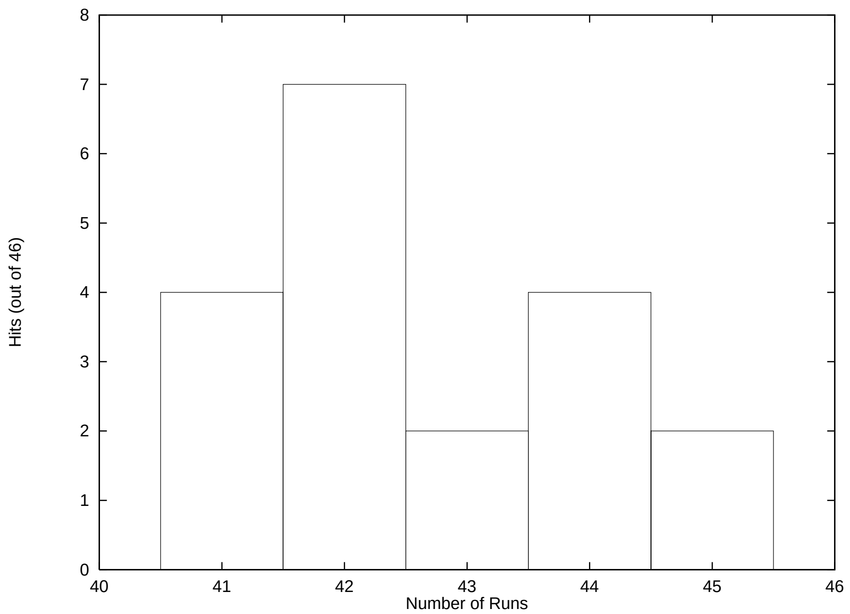

Figure 4-25: Histogram of Maximum Hits Received for Each Imperative Head Run Over 20 Runs

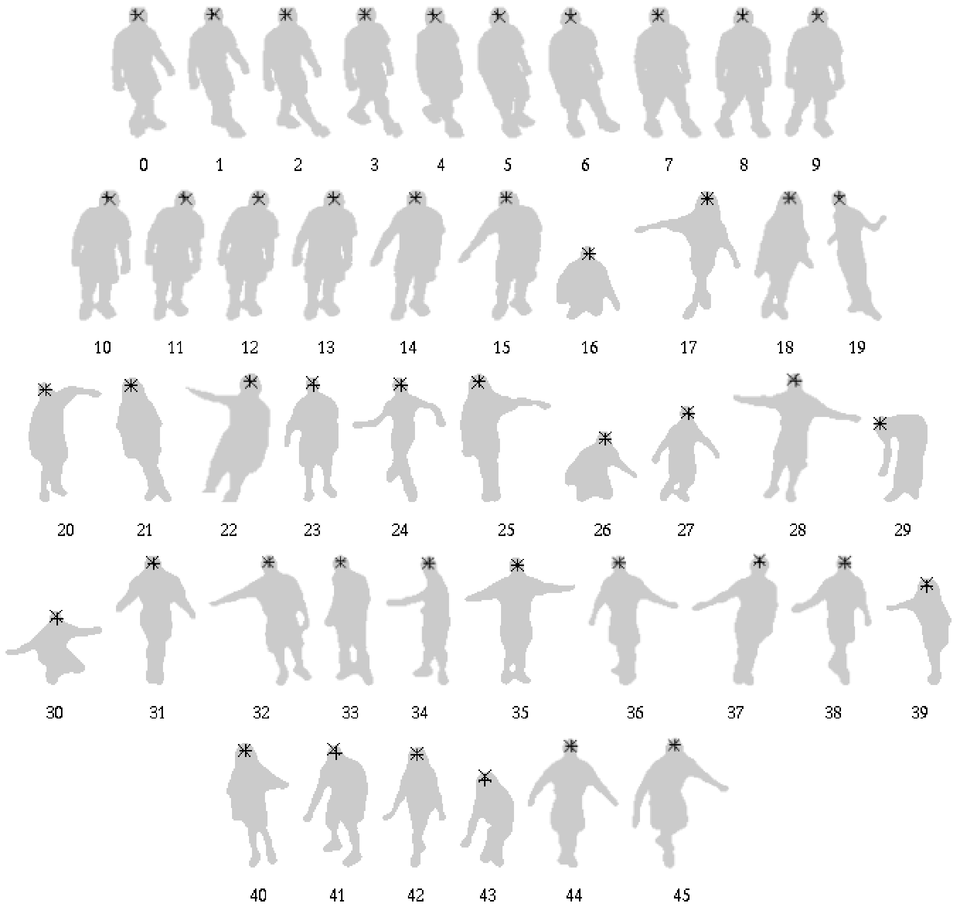

Figure 4-26: Best Evolved Imperative Head Program's Results on Training Set. X is desired, + is program result.

actually be the case, since the head does not move nearly as much as the hands and it cannot be obscured in from of the body.

Random subsampling

Mirrored cases were not possible for the head (at least for this set of people), so only random subsampling generalization was performed. In 10 random subsampling runs, again using 20% as the test set, 74% ± 11% accuracy ($.740 \pm .111,\ N = 10$) was achieved.

Chapter 5

Discussion

This chapter discusses questions raised by the results presented above. It also compares the results of the two approaches. Table 5.1 contains the results from both approaches to simplify comparisons. The chapter is arranged as a series of questions and answers.

5.1 Did it work?

This question really has two facets — the primary objective and the secondary objective. The primary objective, as mentioned above, was whether good solutions could be found at all using GP — a proof of concept. The secondary objective was whether using GP was an efficient, practical way to create active vision routines. We address both issues respectively.

5.1.1 Does GP find *good* solutions?

As mentioned previously, three criteria for good solutions were sought:

- Accuracy
- Non-catastrophic failure
- Generalization

Each is discussed in turn.

Experiment	*Hit Accuracy*
Functional Runs	
Left Hand Runs	
Best hand coded, fitness cases	.87
Best evolved, fitness cases	.93
Best evolved, mirrored generalization	.57
Random subsampling, .8/.2 random dataset split	0.77 ± 0.21, N=35
Right Hand Runs	
Best hand coded	.57
Best evolved, fitness cases	.70
Best evolved, mirrored generalization	.59
Random subsampling, .8/.2 random dataset split	$.267 \pm .163$, N=17
Imperative Runs	
Left Hand Runs	
Best hand coded, fitness cases	.67
Best evolved, fitness cases	.98
Best evolved, mirrored generalization (hit radius 3 pixels)	.48
Best evolved, mirrored generalization (hit radius 5 pixels)	.63
Random subsampling, .8/.2 random dataset split	0.711 ± 0.227, N=35
Right Hand Runs	
Best hand coded, fitness cases	.39
Best evolved, fitness cases	.76
Best evolved, mirrored generalization (hit radius 3 pixels)	.48
Best evolved, mirrored generalization (hit radius 5 pixels)	.85
Random subsampling, .8/.2 random dataset split	$.314 \pm .185$, N=20
Head Runs	
Best hand coded	N/A
Best evolved, fitness cases	.98
Random subsampling, .8/.2 random dataset split	$.740 \pm .111$, N=10

Table 5.1: Summary of Results Over All Runs

Accuracy

Referring to Table 5.1, one can see that the results are good in many cases, especially on the left runs (functional and imperative), which get over 90% accuracy on the fitness cases and around 50% on the mirror cases, and around 70% on the random subsampling runs. Thus, the programs evolved were quite accurate, especially given such rigid constraints on what constituted a hit (3 pixels). The cases on which "misses" were recorded in the numerical data actually turned out to be only near misses upon inspection of the graphical representation. The solutions were also better than human-programmed programs, which is also an important result.

Catastrophe

Judging by the figures showing the output of the program and the desired result (for example, Figure 4-3), in many cases the guesses were reasonable and were not off-the-wall answers. Therefore, the number of catastrophic failures was low for most cases, except in left-hand generalization runs, which are discussed later.

Generalization — Mirror cases

Referring to the table again, one can see that the programs do generalize. For example, the best functional left program gets 57% accuracy on the mirror cases, whereas the evolved program for finding that particular hand (i.e. the right hand program) gets 70%. These are not that different, which implies generalization.

As another example, consider the imperative right hand evolved program, which receives 85% generalization accuracy on the mirror cases when the hit constraint is relaxed to 5 pixels. This is significant generalization. Refer to the graphical output of that program in Figure 4-22 — none of the answers exhibit catastrophic failure. This is remarkable for a generalization test.

Generalization — Random subsampling

The random subsampling runs did not exhibit nearly as good generalization results, as is shown in Table 5.1. The best explanation for this is that the training set was not large enough, having only 36 cases. Since some of the hard cases only have one example similar to them, if this example ends up in the test set, the program will most likely miss it. This is analogous to teaching a student algebra and then slipping a calculus question on the test. The fairly wide variances on the random subsampling accuracies suggest that this kind of problem is occuring. A larger set of fitness cases with many hard cases should be used in future random subsampling runs to avoid this problem.

Generalization conclusions

It is difficult to make a solid conclusion about generalization since the random subsampling runs did not do well and the mirror generalizations sometimes did remarkably well. It is clear that some generalization is occurring or else the right hand programs would not have done nearly as well on the mirrored cases. The failure at random subsampling, however, suggests that the programs are overfitting the training set. It is the author's intuition that a larger number of cases, with many hard ones, will tend to produce better generalization results and overfitting will occur on small data sets. For example, for random subsampling, only 36 cases were used, which is a fairly small number. Whether larger training and test sets will increase generalization will have to be tested empirically.

5.1.2 Is GP a *practical* way to create visual routines?

Parameter tweaking

A main argument in this thesis was that we did not want the programmer to have to tweak parameters. There is no savings if the tweaking is displaced from parameters in the program being sought to the method for finding this program, GP. This is clearly a worry, since GP appears to have many parameters to set. Although some early tweaking of the fitness function was required, numerical parameters were basically chosen intuitively and produced good results with few exceptions. This is important evidence for practicality.

Computation time

Most runs took on average several hours. Good results were usually found within ten runs, which could be performed overnight on a serial $Indigo^2$ R4400 machine. This is a practical amount of computation. The author, for one, prefers to go home and sleep and come back in the morning with some elves having done his work for him. Also, since GP is massively parallel, runs can be spread out across as many processors as are available in order to get results more quickly.

Preparation time

The final major issue, however, is preparation time for using GP. First of all, the answers to the problems need to be found manually. This is actually fairly simple for humans to do. For example, for this research the author wrote a simple program that allowed the answers to be labelled by mouse clicks on a bitmap. Labelling the answers took only several minutes for all 46 cases.

Finally, there is the large issue of primitive selection and programming. As this is actually a very hairy problem in GP in general, we will only assert that it can be solved for this domain and defer the argument to Section 5.10.

5.2 Why are the functional terminals *weak*?

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 to hopefully force generalization. The primitives were carefully chosen to be "weak" in the sense that they had little observational power. For instance, there are no conditionals or predicates which could be used to recognize specific fitness cases. This is very different from the primitives used in the imperative approach, which had predicates and conditionals. These were allowed for those runs specifically to test whether overfitting would occur.

5.3 Why are the right hand solutions less accurate than the left solutions for both approaches, and similarly, why are the convergence rates different between left and right?

The results for the right hand runs were less good than those obtained for the left hand runs in both approaches. In addition, the GP algorithm converges more slowly and with a wider variance in the right runs than in the left-hand cases.

The variation in accuracy is most likely due to the fact that the right hand cases are on average more difficult; in many of these cases the hands are somewhere in front of the body where our vision representation does not have enough information to find them (the human answer oracle used a knowledge of anatomy to guess). This difference in difficulty is probably due to the fact the "left" hand in the image is *really* the user's right hand. Since most people are right-handed, they tend to use that hand more often so it sticks out from their body more often and is easier to find. This accounts for the higher accuracy on the left hand cases.

Even though they are not strictly possible given the primitives, we are implicitly forcing the GP to perform well since we add the total error into the fitness function rather than using just misses. This leads to a tradeoff — the GP has to make good guesses on impossible cases in order to keep fitness low, even with the logarithmic error function. This tradeoff is the best explanation for the slower convergence and wider variation in final solutions. It is harder for the evolution, as it was for the human answer oracle, to solve the right hand cases than the left cases. This is demonstrated by a slower convergence rate and a wider variance.

5.4 Why were the imperative programs more accurate at the training set?

In order to compare the results from both approaches, the result summaries from all the runs is presented in Table 5.1. The functional primitives and imperative primitives performed similarly, although the imperative primitives found a slightly better solution to the training set (imperative 98% compared to functional 93% on the left cases, 76% compared to 70% on the right).

The best explanation for the slightly better performance of the imperative functions in solving the training examples is that the primitives were computationally more powerful. For example, the imperative primitives can store state in markers and have conditionals for performing branching whereas the functional primitives could only perform operations on points. This extra power allows the imperative primitives to solve the hard cases better than the functional primitives. For example, a conditional branch operator lets a program realize it has failed with one attempt and try another method. The functional primitives form a single answer based on functional combinations of the image points and do not allow multiple methods to exist simultaneously in the same program.

5.5 Why did the functional primitives generalize slightly better than the imperative primitives?

The functional primitives were slightly better at mirror generalization (functional 57% compared to imperative 48% on the left, 59% compared to 48% on the right). On random subsampling, however, there was a minor discrepancy on the right cases — the functional primitives got 27% mean accuracy whereas the imperative got 31%. For the left, functional did slightly better with 77% compared to the imperative 71%. As mentioned earlier, the random subsampling runs seemed very sensitive to the fitness case set, so it is hard to compare these results. Also, the means are within a standard deviation of each other, so the difference is not that significant. Therefore, we will focus mainly on the mirror generalization in this section.

The fact that the functional programs were better at generalization was expected, as the functional primitives are weak observationally — there are no conditionals and primitives that can detect particular bits. By making them weak, it is unlikely that the GP will find a program that overfits the training set, which can occur with small numbers of fixed fitness cases. For example, programs with powerful primitives are able to evolve solutions that essentially perform pattern matching on each fitness case using some quirk specific to that image. Once the case is recognized, the fixed answer to that case can be returned, essentially making the program a lookup table. Therefore, it is much more likely that programs with weak primitives will compute general solutions, since they are forced to generalize.

Although the imperative primitives were stronger, the results indicate that programs using them were also able to generalize, though perhaps not as well as the functional programs. Even so, this is an important result, since it was believed that overfitting would be very likely with the powerful primitives. One reason that special-casing did not blatantly appear could be that the space required for a program that could recognize 46 cases and perform the appropriate absolute motions of the attention marker (using primitives that just move it and do not reference image features) was larger than the size of program allowed by the depth constraints. A more likely reason is that a generalizing solution happened to be easier to find in the search space than a lookup-table solution. The fact that the more powerful and (hopefully) more general Ullman-style primitives were able to produce general solutions when discovered with GP is a promising result and strong result. We discuss this more in Section 5.10.

5.6 Why did the right hand evolved imperative program fail catastrophically at mirror cases less often than the left-evolved?

The left-hand evolved program often labelled the foot as the left hand on the mirrored generalization cases (see Figure 4-15). This is a catastrophic failure, since the answer is not even close. The evolved right hand program did not exhibit this behavior on those cases, which it was essentially evolved to find. Also, the evolved left hand programs did fine on the training set (all 46 fitness cases). It has already been suggested that the right hand cases were harder in the sense that the hand was much less visible and harder to classify for a human as well. The better success at generalization of the right hand programs suggests that by being trained on harder cases, better ability to generalize was evolved as well. The fact that many of the left cases are "obvious" probably let the evolution slack off early and not produce as general solutions as it could have.

Simple experiments could be designed to test this hypothesis on other domains using GP in order to see if it is a common phenomenon. An experiment would only require a way to divide a large pool of fitness cases into *easy* and *hard*. Then one series of runs would train on the easy cases, another series on the hard cases. Finally the solutions would be evaluated on the other set in order to see if the programs trained on the harder cases were more general.

5.7 How do the best evolved programs work?

The functional solutions were much easier to analyze for two reasons. First, they were smaller due to the parsimony constraint. Second, they are pure functions. Side-effecting programs are much harder to understand since the environment matters. As mentioned above, the best functional programs basically look for a bottom or top edge in an appropriately sized window of the image and then move the point up or down a little to center on the hand.

After looking through the imperative programs, it is still unclear how they work. Stepping through the evaluation of the programs and displaying a graphical representation of the environment might help to elucidate their behavior, but was not completed by the time of this thesis. The programs certainly perform certain obvious inefficient or useless steps, such as moving the attention left then immediately right or setting a marker right off the bat when the marker is already initialized to the attention point.

5.8 Why are the final solutions in the imperative case so unwieldy and inefficient?

Several GP researchers have noticed the idea of "bloating" [Tackett, 1995]. Bloating is the tendency of useless code to collect toward the end of a GP run. Some research has shown that this useless code (called *introns* after biology, or the unexpressed bits of DNA which comprise some 90% of the genome) serves to protect the useful, important bits of code from being broken apart by unfortunate (destructive) crossover [Nordin *et al.*, 1995]. It is an artifact of the evolution and may serve a useful role. Parsimony constraints often keep this bloating to a minimum, but tend to discourage diversity. Intuitively, high diversity is desirable to search a larger space and not waste computation. The results in [Nordin *et al.*, 1995] are the main reason that the decision not to apply a parsimony constraint to the imperative sections was made. It is also why the *null* property was left in the experimental runs once the bug was discovered — it could potentially be the source of introns for the evolution. Whether this is useful or not will need to be investigated further.

5.9 If the obese programs are to be used in a real system, can they be optimized?

Several methods could be used to prune out the introns after the main evolution is done. One way is to apply dead-code removal algorithms to the programs. Examples of these are common in compiler optimizers (see, for example, [Aho *et al.*, 1986]). Unfortunately, dead code removal can be tricky and inefficient for side-effecting code since much more state must be examined.

Another method, suggested as a way of finding implicit introns in code in [Nordin *et al.*, 1995], is to simply try removing each node one by one and retesting the program on the fitness cases each time. If the resulting answer or state is different, then the code is performing calculation and should be left in. If not, it might be safe to remove it.

Finally, a simple approach is to just turn the parsimony constraint on in the GP system after the run seems to have converged. Hopefully, the GP will start to reduce the size of the programs without reducing the fitness. Intuitively, this should be a reasonable approach since the GP's purpose is to find a subspace where good solutions exist. Once the subspace has been found and the code bloated to imply that no better subspace will be found in the run, the programs can be pruned. The hope is that the GP will remain in that good subspace but find smaller programs in it. If not, the best program will still be available in the GP history, large or not, to be used as is.

5.10 Don't you essentially solve the problem by picking the primitives?

This is an important question. One of the most common and most valid criticisms of GP is that the GP cannot *help* but find the solution since the primitives are chosen with an idea of how they will solve the problem. They are often high-level, abstract and not reusable for other problems. We argue here that at least in the domain of active vision, we can get around this problem.

Claim: GP is practical for composing *reusable*, *low-level*, *general* vision primitives into visual routines to solve various tasks.

The imperative primitives were chosen to meet these criteria. First of all, the edge detectors were made smaller so they were not perfectly suited for finding hands. Second, the primitives were generally taken right from Ullman and Chapman, with only minor changes (such as the creation of iterators to make them GP-friendly) and a few additions such as the *move-attention-percent*. Ullman argues that the visual routine primitives will have these properties. The fact that these primitives, which Chapman used for many different tasks in Sonja [Chapman, 1992], can be composed to solve our task as well is an important result. This implies that these general, low-levels primitives can be reused for many different visual tasks. Although this is not explicitly shown here, the evidence for it is strong. Thus, the overhead for the programmer is a one-time cost — once the primitives are programmed, they can just be reused. It is evolution's job to cull out useless primitives from the set and to find which ones are useful for the given problem. In order to strengthen support for this claim, future work will use these primitives on other qualitatively different tasks.

Another related point is that many industrial vision systems come with a wide set of fixed primitives for doing correlation, pattern-matching, subsampling, etc. These primitives can be called directly during the interpretation of a GP program, removing the entire problem of writing, debugging and choosing the primitives. These primitives are used to write vision programs for industrial vision, so are clearly general. The programmer will still need to do some work in the integration, however. For example, GP does not deal with potentially unbounded loops well.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Computer vision is still a difficult problem, even after years of research. The traditional approach, consisting of creating and maintaining a fully-resolved model of the entire visual field, has proven intractible in practice. Recently an alternative, active vision, has been proposed. Active vision decomposes vision into context-driven, task-specific visual routines. One disadvantage of this approach is that the visual routines are numerous and must be rewritten for each task. They are also time-consuming to program and optimize. To solve this problem, we applied Genetic Programming to the task of developing simple visual routines from common, general primitives. Our system was able to evolve routines which analyzed silhouettes of people extracted from real camera data. The programs recovered salient information from the images, specifically the location of a person's hands. The evolved routines performed better than those created by a human programmer and were able to generalize to images outside of the training data, which is important for a machine learning solution. The solutions were found within a short amount of time, with practical computational resources and with virtually no parameter optimization in the GP algorithm. These routines can been used as part of a larger system for interaction with a virtual environment, which allows control of a virtual world and virtual agents with simple hand gestures.

A common criticism of genetic programming is that most of the work is in selecting and programming the primitives, which are then not general or reuseable for other tasks. In this research, the primitives used for half of the experiments were chosen to be general and reuseable for other vision problems. The fact that GP was able to use them to solve this task is strong evidence that GP is a practical approach to the problem of creating active vision routines automatically without having to create a new set of primitives each time.

Our implementation used a typed paradigm for genetic programming, which restricts the algorithm search space and speeds up fitness evaluation. We found empirically that in our fitness domain, which considered the distance of a predicted hand location with the actual hand location, the use of a "robust" fitness function that weighted errors logarithmically yielded considerably better results than a linear error function.

6.2 Limitations and Future Work

One of the main limitations of the experiments described in this thesis is that the primitives were only used on one problem — evolving hand-finding programs. Ullman suggests that the visual routines primitives should be useful for many various tasks. An obvious extension to this research would be to take the set of more general imperative routines and try to solve a problem qualitatively different with them, such as posture recognition. If the primitives are very specific to one task and are not reuseable, the gain in efficiency from the GP might be lost to finding new primitives to solve the task. At that point, it might be easier to write the programs by hand, although GP could certainly be used to optimize them.

A major criticism is that the number of fitness cases was low, so generalization results were not strong. Future research will try runs using an order of magnitude more fitness cases, especially for the random subsampling trials. The main reason the fitness case set was small was due to computational limitations — it would take almost a month to run twenty random subsampling runs with 500 fitness cases. Using parallel vision hardware could speed up the evaluation. Also, since the algorithm is parallelizable, it is possible to split the runs across multiple machines. Finally, a simple trick for having a large fitness set but reducing computation time is to use a sampled fitness calculation — pick a small fraction of the fitness cases at random each generation in order to estimate fitness.

Another minor limitation is that the algorithm as specified is a *supervised* learning algorithm, which means that it requires an "answer oracle" to tell it the correct answers on its training data in order to learn. This implies that there is a human in the loop specifying answers. Although it is in general much easier for a human to label the answers to cases than to write and debug a program to do it, it would be desirable to get the human out of the system entirely. Nature clearly has no oracle. One way to do this would be to use an unsupervised learning method to decide fitness of a huge population of random programs. For instance, one could use a different modality to produce a reward value for the visual routine. A robot that needs to find an object that it knows the taste of, say, can run around trying routines to find the object and constantly tasting things. When it tastes the right thing, it rewards any visual routines which may have suggested the object or helped in locating it. In this way, the system can bootstrap the fitness function. This is clearly a highly contrived example, but serves to illustrate the idea.

Another idea to get the human out of the loop is to use the slow general vision algorithms to produce answers to the various training tasks. Although the general systems are not useful on-line in a real system, they could be useful for finding fast task-specific routines to be included in the real-time system. For example, one method proposed to find hands in a silhouette using classic techniques was to use a three-dimensional geometric model of a human, complete with realistic joint constraints. An inverse projection could be used to find a configuration of the 3D model which could produce the silhouette. Since the hand locations are known in the model, their projection into the silhouette can be calculated, giving the hand locations in the silhouette. This is a computationally expensive method which is not currently viable in real-time, but it could be used on the fitness cases as the answer oracle for the GP system.

Appendix A

Evolved Programs

This appendix contains evolved programs too long to include in-text, just for readers to have a feel for what the evolution produces.

A.1 Best Evolved Imperative Left Program

```
(progn (place-marker M2) (progn (move-attention right) (progn
(move-attention right) (progn (property-scan-in-direction figure down)
(progn (scan-in-direction down) (move-attention up) (progn
(move-attention up) (scan-along-line)) (is-on)) (attend-to M2)
(scan-in-direction down) (scan-along-line) (not
(property-scan-in-direction figure left)) (progn (progn (while-element
(move-attention right) (below-marker M2)) (move-attention up)
(property-scan-in-direction null left) true) (move-attention
up) (not (property-scan-in-direction figure down)) (attend-to M2)
(is-on) (move-attention down) (not (below-marker M2))) (progn
(while-element (is-on) (scan-along-line)) (not
(property-scan-in-direction figure down)) (place-marker M2) true)
(move-attention up) (scan-along-line)) (not (move-attention left))
(not (while-element (property-scan-in-direction figure down)
(property-scan-in-direction figure left))) (scan-along-line)
(property-scan-in-direction figure down) (progn (below-marker M2)
(property-scan-in-direction figure down) (progn (move-attention right)
(not (move-attention right)) (move-attention down) (scan-in-direction
down) (scan-along-line) (progn (right-of-marker M2) (move-attention
up) (property-scan-in-direction figure up) (while-element
(move-attention up) (property-scan-in-direction top-edge left))
```

```
(scan-along-line) (below-marker M2) (scan-in-direction down)) (progn
(while-element (is-on) (scan-along-line)) (not
(property-scan-in-direction top-edge left)) (place-marker M2) true)
(progn (scan-in-direction down) (move-attention up) (progn
(move-attention up) (scan-along-line)) (is-on)) (scan-in-direction
down) (scan-along-line)) (move-attention right) (scan-along-line)
(below-marker M2) (not (while-element (move-attention up)
(move-attention left)))) (progn (while-element (move-attention up)
(property-scan-in-direction top-edge down)) (move-attention up)
(while-element (move-attention up) (move-attention left))
(while-element (move-attention up) (while-element
(property-scan-in-direction figure down) (not (below-marker M2))))
(is-on) (below-marker M2) (while-element (move-attention up)
(while-element (move-attention right) (move-attention up))))
(move-attention up) (scan-along-line)) (not
(property-scan-in-direction figure down)) (progn (move-attention up)
(attend-to M2) (is-on) (move-attention right) (progn (move-attention
right) (while-element (property-scan-in-direction figure down) (not
(property-scan-in-direction figure down))) (move-attention down)
(property-scan-in-direction figure down) (scan-along-line)
(property-scan-in-direction right-edge up) (progn (move-attention up)
(property-scan-in-direction figure down) (property-scan-in-direction
figure up) (while-element (move-attention up)
(property-scan-in-direction top-edge left)) (is-on) (below-marker M2)
(while-element (not (move-attention right)) (move-attention up)))
(progn (scan-in-direction down) (move-attention up) (progn
(move-attention up) (scan-along-line)) (is-on)) (progn (move-attention
right) (not (move-attention up)) (property-scan-in-direction figure
left) (scan-in-direction down) (scan-along-line) (move-attention left)
(progn (right-of-marker M2) (move-attention up)
(property-scan-in-direction figure up) (while-element (move-attention
up) (property-scan-in-direction top-edge left)) (scan-along-line)
(below-marker M2) (scan-in-direction down)) (progn (scan-in-direction
down) (move-attention up) (progn (move-attention up)
(scan-along-line)) (is-on)) (scan-in-direction down)
(scan-along-line)) (scan-along-line)))) (move-attention up)
(move-attention right))
```

A.2 Best Evolved Imperative Right Program

```
(progn (move-attention-percent down 30%) false
(property-scan-in-direction null right)
(move-attention-percent up 70%) (move-attention-percent right 70%)
(while-element true (progn false (move-attention-percent down 30%)
(above-marker M1) (above-marker M1) (move-attention-percent down 30%)
false false (not (property-scan-in-direction figure right)) (progn
(progn (move-attention up) (move-attention up) (progn false false
false (while-element (move-attention down) (scan-in-direction down))
false false true true) (progn (move-attention right)
(property-scan-in-direction figure right) (property-scan-along-line
bottom-edge) (not (below-marker M1)) (below-marker M1) (progn
(scan-along-line) false false (property-scan-in-direction
null right) false false false (scan-along-line))
(property-scan-in-direction null right)
(property-scan-in-direction null right))
(property-scan-in-direction null right) (left-of-marker M2)
(above-marker M1) (not (progn (scan-along-line) false false
(property-scan-in-direction null up) false false false
(scan-along-line)))) (scan-in-direction up) (move-attention up) false
(while-element (above-marker M1) (property-scan-in-direction
null right)) (progn (move-attention up)
(property-scan-in-direction figure right) (scan-in-direction up)
(progn (move-attention down) (property-scan-in-direction figure right)
(property-scan-in-direction null right) (not
(property-scan-in-direction figure right)) (below-marker M1)
(above-marker M1) (property-scan-in-direction figure right)
(while-element (below-marker M1) (move-attention up))) (progn
(move-attention down) (property-scan-in-direction figure right)
(property-scan-in-direction figure right) (property-scan-along-line
null) (below-marker M1) (draw-line M2) (move-attention down)
(while-element (property-scan-along-line null)
(left-of-marker M2))) (move-attention-percent down 30%)
(move-attention up) (not (below-marker M1))) (left-of-marker M1)
(above-marker M1) (above-marker M1) (property-scan-in-direction figure
right)) (scan-along-line))) (if-then-else (progn false (not
(move-attention-percent right 70%)) (property-scan-in-direction
bottom-edge right) (progn (scan-along-line) false false
```

```
(property-scan-in-direction null right) false false false
(scan-along-line)) (move-attention-percent down 30%) false false
(property-scan-in-direction left-edge right) (progn (progn
(move-attention up) (move-attention up) (progn false false false
(while-element (move-attention down) (scan-in-direction down)) false
false true (property-scan-in-direction right-edge up)) (progn
(move-attention down) (property-scan-in-direction figure right)
(property-scan-in-direction figure right) (left-of-marker M2)
(below-marker M1) (draw-line M2) (property-scan-in-direction figure
right) (while-element (property-scan-along-line null)
(move-attention-percent right 30%))) (move-attention down)
(move-attention-percent down 30%) (move-attention down) (not
(property-scan-in-direction figure right))) (scan-in-direction up)
(progn (scan-along-line) false false (property-scan-in-direction
left-edge right) false false false (scan-along-line)) false
(while-element (above-marker M1) (property-scan-in-direction
null down)) (property-scan-in-direction null right)
(left-of-marker M1) (right-of-marker M1) (move-attention up)
(move-attention up)) (scan-along-line)) false (progn (move-attention
down) (property-scan-in-direction figure right)
(property-scan-in-direction null right) (not
(property-scan-in-direction null right)) (below-marker M1)
(draw-line M2) (property-scan-in-direction figure right) (below-marker
M1))))
```

A.3 Best Evolved Imperative Head Program

```
(progn (progn (progn (move-attention down) (while-element (is-on)
(move-attention right)) (if-then-else false true (progn (is-on)
(if-then-else false false (scan-along-line)) (not false) false true
false)) (property-scan-along-line right-edge) (progn false true
(scan-along-line) false true true true true false true)
(property-scan-along-line right-edge) (place-marker M1) (while-element
(is-on) (progn (is-on) (if-then-else false false (scan-along-line))
(not (is-on)) false true false)) (progn (move-attention right) (progn
(is-on) (property-scan-along-line right-edge)
(property-scan-along-line right-edge) (is-on) false true) (not (progn
true (if-then-else true false false) (below-marker M2) true true
```

```
true)) (above-marker M1) (move-attention down)
(property-scan-along-line right-edge) (progn true (progn false
(move-attention down) (above-marker M2) true true true)
(property-scan-along-line right-edge) true true true) (progn true true
(scan-along-line) false true true true true false true) (progn (is-on)
(if-then-else false false (scan-along-line)) (progn (draw-line M2)
(progn true (move-attention right) (if-then-else true false true) true
true true) false (move-attention right) (if-then-else true false
false) (scan-in-direction right) true (above-marker M1)) false true
false))) (move-attention down) (scan-along-line)
(property-scan-along-line right-edge) (scan-in-direction right)
(while-element (is-on) (not (progn (progn (scan-along-line) true
(scan-along-line) false (is-on) true true (is-on) false true) (progn
true (move-attention down) (if-then-else true true false) false true
true) true (move-attention right) (move-attention down)
(property-scan-in-direction right-edge right) true (progn false false
(scan-along-line) false true false true true false true))))
(scan-along-line) (move-attention right)) (if-then-else false true
(progn true true)) (progn false true) (move-attention right)
(while-element (is-on) (not (progn (place-marker M1) (progn true
(progn (is-on) (property-scan-along-line right-edge) (progn (draw-line
M2) (progn true (progn false true) (if-then-else true false true) true
true true) false (progn true (move-attention down)
(property-scan-in-direction right-edge right) false false true) (progn
false true (scan-along-line) (is-on) true false true true false true)
(if-then-else false true (not (scan-along-line))) true (progn true
true)) false true false) (scan-in-direction right) true true true)
false (move-attention down) (move-attention down) (move-attention
down) true (property-scan-in-direction right-edge right))))
(move-attention right) (scan-along-line) (move-attention right))
```

Bibliography

[Aho *et al.*, 1986] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

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

[Andre, 1994] David Andre. Automatically defined feautures — the simultaneous evolution of 2-dimensional feature detectors and an algorithm for using them. In Kim Kinnear, editor, *Advances in Genetic Programming*, chapter 23. MIT Press, Cambridge, Massachusetts, 1994.

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

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

[Blake and Yuille, 1992] Andrew Blake and Alan Yuille, editors. *Active Vision*. Artificial Intelligence. MIT Press, Cambridge, Massachusetts, 1992.

[Chapman, 1992] David Chapman. *Vision, Instruction and Action*. MIT Press, Cambridge, Massachusetts, 1992.

[Cliff *et al.*, 1993a] Dave Cliff, Phil Husbands, and Inman Harvey. Explorations in evolutionary robotics. *Adaptive Behavior*, 2(1):73–110, Summer 1993.

[Cliff *et al.*, 1993b] Dave T. Cliff, Phil Husbands, and Inman Harvey. Evolving visually guided robots. In J.-A. Meyer, H. Roitblat, and S. Wilson, editors, *Proceedings of the* *second international conference on Simulation of Adaptive Behavior (SAB 92)*, pages 374–383, 1993.

[Goldberg, 1989] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[Harvey *et al.*, 1994] Inman Harvey, Phil Husbands, and Dave Cliff. Seeing the light: Artificial evolution, real vision. In Dave Cliff, Phil Husbands, Jean-Arcady Meyer, and Stewart W. Wilson, editors, *From Animals to Animats 3: Proceedings of the Third In ternational Conference on Simulation of Adaptive Behavior*, pages 392–401, Cambridge, Massachusetts, 1994. MIT Press.

[Hogger and Kowalski, 1992] C. J. Hogger and R. A. Kowalski. Logic programming. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 873–891. John Wiley & Sons, Inc., New York, second edition, 1992.

[Holland, 1975] John H. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor, MI, 1975.

[Horswill, 1993] Ian D. Horswill. *Specialization of Perceptual Processes*. PhD dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1993.

[Horswill, 1995] Ian Horswill. Visual routines and visual search: a real-time implementation and an automata-theoretic analysis. *Accepted for International Joint Conference on Artificial Intelligence (IJCAI-95)*, 1995.

[Johnson *et al.*, 1995] Michael Patrick Johnson, Trevor Darrell, and Pattie Maes. Evolving visual routines. *Artificial Life*, 1(4):373–389, 1995.

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

[Koza, 1994] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, Massachusetts, 1994.

[Maes *et al.*, 1993] Pattie Maes, Bruce Blumberg, Trevor Darrell, and Sandy Pentland. Alive: An artificial life interactive video environment. *Visual Proceedings of Siggraph '93, ACM Press*, 1993.

[Maes *et al.*, 1995] Pattie Maes, Trevor Darrell, Bruce Blumberg, and Sandy Pentland. The ALIVE System: Full-body Interaction with Animated Autonomous Agents. In *Proceedings of the Computer Animation Conference, Geneva, Switzerland*. IEEE Press, 1995.

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

[Nordin *et al.*, 1995] Peter Nordin, Frank Francone, and Wolfgang Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications (ML-95)*, pages 6–22. National Resource Laboratory for the study of Brain and Behavior, University of Rochester, June 1995. Technical Report 95.2.

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

[Rosca and Ballard, 1995] Justinian Rosca and Dana Ballard. Causality in genetic programming. In *Proceedings of the Fifth International Conference on Genetic Algorithms (ICGA95)*, 1995.

[Sims, 1991] Karl Sims. Artificial evolution for computer graphics. *Computer Graphics*, 25:319–328, 1991.

[Tackett, 1995] Walter A. Tackett. Greedy recombination and genetic search on the space of computer programs. In D. Whitley and M. Vose, editors, *Foundations of Genetic Algorithms III*. Morgan Kauffman, San Mateo, CA, 1995.

[Teller and Veloso, 1995] Astro Teller and Manuela Veloso. PADO: Learning tree-structured algorithms for orchestration into an object recognition system. Technical Report CMU-CS-95-101, Carnegie-Mellon University, School of Computer Science, CMU, Pittsburgh, PA, 1995.

[Ullman, 1984] Shimon Ullman. Visual routines. *Cognition*, 18:97–159, 1984.

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

[Watkins, 1989] Chris Watkins. *Learning From Delayed Rewards*. PhD thesis, King's College, 1989.

[Weiss and Kulikowski, 1991] Sholom M. Weiss and Casimir A. Kulikowski. *Computer Systems That Learn*. Morgan Kaufmann, 1991.

[Whitehead, 1992] Steven D. Whitehead. *Reinforcement Learning for the Adaptive Control of Perception and Action*. PhD dissertation, University of Rochester, Department of Computer Science, February 1992.