The Simulated Evolution of Robot Perception

Martin C. Martin

A Dissertation Submitted in Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy

in the field of

Robotics

Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:

Hans Moravec, chair
Peter Cariani
Illah Nourbakhsh
Simon Penny

Abstract

This dissertation tackles the problem of using genetic programming to create the vision subsystem of a reactive obstacle avoidance algorithm for a mobile robot. To focus the search on computationally efficient algorithms while dealing images from a non-toy problem, the representation restricts computation to be over a window which moves vertically over the image.

The evolved programs estimated the distance to the nearest object in various directions, given only a camera image as input. Using a typical supervised learning framework, images of the environment were collected from the robot's camera and the correct distance in various directions determined by hand. Evolving programs were evaluated on this fixed training set and compared to the hand determined answers. Once the evolution was complete, obstacle avoidance programs were written to use the best evolved programs, and the combined system used to control a robot.

The approach can be seen as automating the iterative design process. A researcher's main contribution is typically at a high level—techniques and frameworks—yet most time is spent on an example problem, trying different instantiations until one works. When faced with such a problem, one can usually think of a half dozen very different approaches, and even write them out in pseudo code. The technique proposed here can be seen as searching the space spanned by that pseudo code.

In a series of experiments, programs were evolved in three different ways for two different environments to both create working systems and push the limits of the approach. Even in this nascent form, the evolved programs work about as well as existing, hand written systems. They used a number of architectures, including a recurrent mathematical formula and a series of `if` statements similar to a decision tree but with non-linear relations between as many as five image statistics. They successfully coded around peculiarities of the imaging process and exploited regularities of the environment. Finally, when given a representation so general as to cause the genetic algorithm to fail, and hand constructed rough answer was used as a "seed," which the genetic algorithm successively modified to cut its error rate by a factor of 5.8.

This dissertation grew out of my conviction that critiques of Artificial Intelligence can be viewed constructively, as intellectual lighthouses to guide us closer to the fundamental nature of thought, to the real problems at the heart of intelligence. To not address them, to work on techniques with fundamental flaws, would be fooling oneself no matter how impressive the demonstrations. There seems to be something fundamental about AI that we are all missing, and I believe these critiques bring us closer to it.

This dissertation describes the experiments and their results, discusses ways to develop them further, then presents critiques of AI and discusses the potential of this approach to overcome those critiques.

Acknowledgements

First and foremost I would most like to thank Hans Moravec, my advisor and friend, for his support and encouragement. He has been kind enough to give me the one thing I desired most: the freedom to follow my own ideas, although they must have seemed to come from another planet. His wit and humour brightened hours of fascinating discussion about computers, the nature of reality, the culture and events of the 60s and 70s, and occasionally, robotics.

Illah Nourbakhsh was a fabulous resource, always able to provide a deep insight no matter how brief the discussion. This work wouldn't have succeeded as well as it did if it wasn't for his thoughtful guidance. I am also grateful to Peter Cariani for his valuable comments and engaging discussion. His breadth of knowledge—from cybernetics to the mechanics of biological vision to how to write a dissertation—has helped my thinking both here and on larger issues. Simon Penny provided friendship and a sympathetic view, especially for the broad strokes in the last chapter. Whenever I disparaged at what a technical audience would make of that chapter, I needed only think of Simon's belief that technology can and should be aware of the culture surrounding it.

Peeter Piegaze was not only a cohort in my first explorations into genetic programming and computer vision as an undergrad, but a good friend who, like me, enjoyed few things more than an evening discussing philosophy over a few pints. Sven Dickenson and John Tsotsos not only provided me my first research opportunity and coached me through it, but supported me in applying for graduate school and again when searching for a faculty position. Their kindness will never be forgotten.

Jesse Eusades was not only a witty office mate who exposed me to new music and fashion, but a good friend, always ready to help out, whether with a drive home or to give up his Saturday to fix the robot when this dissertation was already late. Dottie Marsh was always ready with a laugh, and could perform administrative miracles getting me reimbursements from whatever scraps of paper I managed to save. I was lucky enough to have Marce Zaragoza and Suzanne Lyons Muth as guardian angels, heading off more administrative problems at the pass than I probably ever want to know.

Several friends shared in the growing up I did over those years, and in my staying young. Garth Zeglin was there from the beginning to the end, and was always ready to grab a pint and talk of computers, politics, and life. John Murphy dragged me out, against my better judgement and to my great benefit, to go dancing and have fun; I dragged him into the lab to play Doom. And while Matt Deans and I only got to know each other for a short time before I left, he was always ready with a laugh and his friendship.

Finally, I would like to thank my family. My brother has always shared my love of computers, and I will always be grateful that he has shown me new ways to enjoy them. And it was my parents' high expectations of me in everything I did that led me to never compromise in my work. This dissertation would not have been possible without them.

Contents

Abstract iii
Acknowledgements v

1 Overview 1

Part I • Motivations

2 Related Work 13
3 Technical Framework 27

Part II • Experiments

4 Data Collection 39
5 Offline Learning: Basics 49
6 Expanded Representation 63
7 Focused Representation 79
8 Online Validation 105

Part III • Reflections

9 Discussion 117
10 Philosophy & Manifesto 133

References 155

1 Overview

Introduction

Artificial Intelligence (AI), the scientific and engineering endeavour of creating a machine that can think, has made great advances over the decades. Computers have beaten the best human chess players, speech recognition systems allow computers to respond to voice commands, and robots give tours of museums, reliably navigating rooms full of people.

Yet the original goal of human-level mental competency has proven elusive. Perception is a key cognitive ability, and human perception has been shown to use very subtle and complex mechanisms, so vision seems as good a place as any to look for intelligence. Various authors have argued that embodiment and perception are key, and this dissertation does just that, creating a visually guided mobile robot.

Creating systems of the subtlety and complexity of human vision has proven difficult to do by hand, so this dissertation uses Evolutionary Computation (EC) to create the vision subsystem, automating the process of testing and revising algorithms so common in robotics. These points are expanded at the end of this chapter and in the "Philosophy & Manifesto" chapter. Non-technical readers, and those primarily interested in the broader implications of this dissertation, would do well to read that chapter first.

Motivations

The use of EC to create control programs for mobile robots is not new. The field, approximately 10 years old and known as Evolutionary Robotics, evaluates potential control programs by setting the robot to a task and seeing how well the program achieves it. This happens either in simulation or on a real robot, although the results are always validated on a real robot.

Previous evolutionary robotics work has incorporated video as a sensor, but to do so has forced the image through a huge bottleneck, to either three or sixteen pixels. With such a low resolution, only problems in carefully constructed worlds were possible.

For the robot to be truly embodied in a real world environment, and to avoid all manner of obstacles using only vision, it requires more complex algorithms that respond to a larger part of the image. Simulating real images to the required fidelity would be a thesis in itself, if even possible at the speeds needed to support simulated evolution. However, evolving on the real robot is also not possible, since this limits the number of evaluations, and hence the complexity of what can be evolved.

In addition, without specifying any *a priori* structure for the evolved programs, the problem becomes many orders of magnitude harder than previous work. The input, essentially the amount of light in various directions, is simply too distantly related to the output, i.e. the direction to travel.

For both these reasons, the control program was divided into two components, the *vision subsystem* and the *navigation subsystem.* The

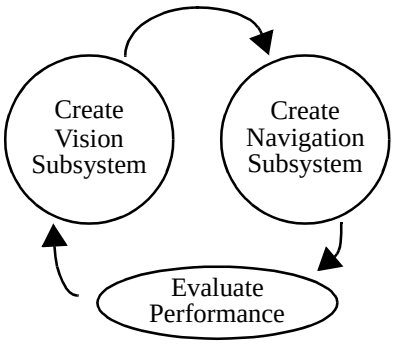


Figure 1: The Outer Loop. First, the vision subsystem is created in isolation. Then the vision system is fixed and used in the construction of the navigation system. The performance of the resulting robot is used to guide the next version of the vision system.

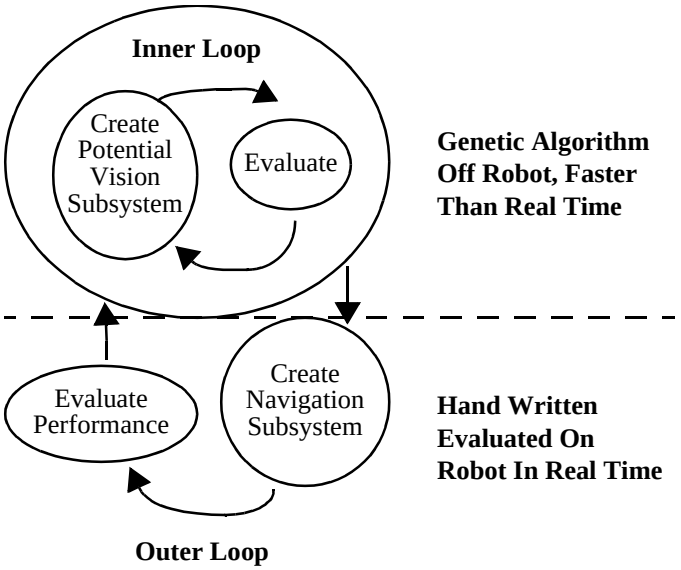


Figure 2: The Outer and Inner Loops. The "Create Vision Subsystem" step has been expanded with its own generate-and-test loop, implemented with a genetic algorithm.

interface between the two was completely specified: the vision algorithm takes in an image and a direction, and returns an estimate to the nearest obstacle in that direction. The navigation algorithm starts with a number of such estimates from the current image, and computes a direction to travel.

While specified, this interface is not arbitrary. Sonar and laser range finders, by far the most popular sensors, return distances, and the number of such estimates per image, six for training and twelve while controlling the robot, are a minimal representation of the environment compared to the dense depth map typically used in stereo based navigation. This minimal representation was inspired by arguments that representation should be used sparingly.

This division into vision and navigation subsystems led to constructing the control program in two steps; see Figure 1. The vision subsystem is constructed first, then the navigation algorithm is constructed to handle the sorts of errors made by the vision algorithm. Lessons learned about which errors are easy to compensate for and which are more difficult can guide the design of the next version of the vision system.

The vision system is constructed through a similar "generate and test" paradigm, see Figure 2. Since this loop happens as one step within the previous loop, it is called the "inner loop," whereas the loop in Figure 1 is referred to as the "outer loop."

For the person constructing the system, there is another asymmetry between the two subsystems: the vision subsystem is much harder to create. The output of the vision subsystem is the estimated distance to the first obstacle in various directions, and in this way similar to sonar. The navigation subsystem can therefore adapt designs created for sonar. When it comes to navigation, sonar is a much more common sensor than vision, so navigation from this type of data is well understood. In contrast, while there has been much work on computer vision, there has been little applying it to obstacle avoidance on a mobile robot.

For this reason, the core of this dissertation is the development of the vision subsystem. Two traversals through the outer loop were made, that is, the vision subsystem was developed by making many passes through the inner loop, then fixed. A navigation subsystem was then created, and finally the entire control program was evaluated by using it to control the robot in a real world environment. Reflecting on the difficulties and successes along the way led to changes in the vision framework, which lead to a new navigation algorithm and a second round of validation on the robot. Suggestions for a third iteration are presented.

In constructing the vision subsystem, the inner loop is automated using evolutionary computation. Potential programs are given an image and column location as input, and produce a single real-valued output, which is interpreted as the pixel location of the lowest non-ground pixel

Figure 3: What is computed. For a given column of the image, the evolved vision subsystems compute the boundary between the ground (lower line) and a non-ground object (upper line). If there is more than one such boundary, the lowest is desired. This is done independently for six columns in the image.

within that column. (See Figure 3.) Assuming objects touch the ground and that the ground is roughly flat, the lowest non-ground pixel can be easily converted to a distance from the robot. No state is maintained from one image to the next or between columns within an image, which means all programs are reactive.

A traditional supervised learning framework was used. To collect a training set, the robot navigates using sonar and passively collects a video stream. In each of six columns of each image, the correct answer, i.e. the location of the lowest non-ground pixel, was determined by a human and recorded. The fitness of a potential algorithm was simply how close it came to the human-provided result. Thus, one pass through the outer loop follows the four phases of Figure 4.

Record Video Robot, Real Time

Learn Offline Genetic Algorithm

Build Navigation By Hand

Validate Online Robot, Real Time

Figure 4: The four phases of one pass through the outer loop. First, a set of representative images are collected. Then the inner loop, an offline genetic algorithm, creates a vision subsystem. Next, a navigation system is written by a human programmer. Finally, both systems are used to control the robot.

As for which EC framework to use, genetic programming (GP) is attractive in this setup because it uses a representation close to traditional programming languages. As programmers, we feel it is a natural representation for algorithms. For example, when sitting down to write a program by hand, most programmers would not simply adjust the weights of a neural network, but instead use variables, control flow, and the other standard features of a programming language. In addition, as programmers we are very familiar with such a representation, which helps us understand the evolved algorithms. Finally, existing computer vision algorithms are expressed in this format, making them easier to incorporate into the framework. In fact, no other machine learning framework uses a programming language as its representation, nor allows such a range of architectures to be searched. For example, neural networks, both feedforward and recurrent, can be represented in GP, along with many other learning frameworks. Once the general representation was decided, evolutionary computing was the only choice.

The most general and straight forward way of incorporated image information into GP is to simply allow it to access individual pixels and give it some looping constructs. However, with the limited computing power of today's computers, it would take a long time to find even a simple algorithm that examines the correct location in the image.

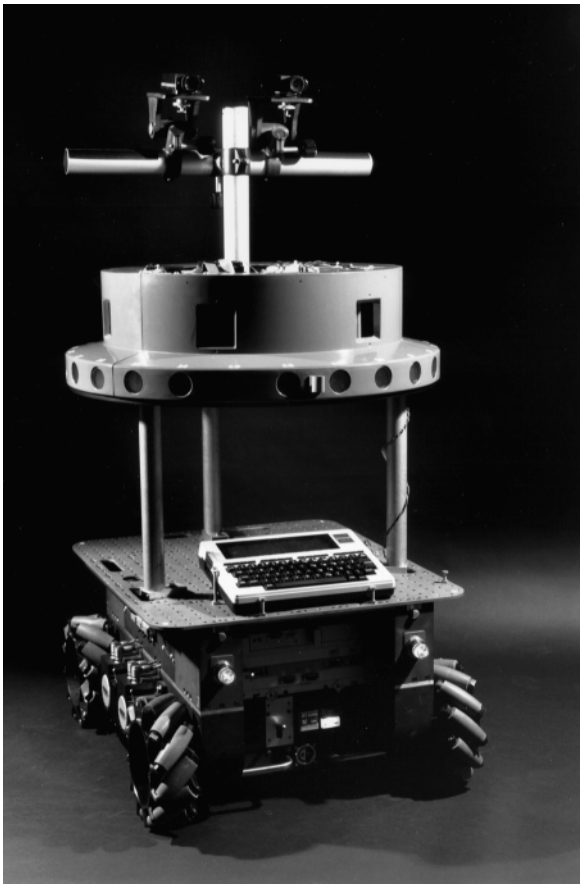
To give the representation a little more structure, successful visual obstacle avoidance algorithms were examined and a common building block was found. In all these existing system the bulk of run time was spent performing a computation over a rectangular window which iterated over a column or row of the image.

Thus, a new type of node was created, an *iterate* node, whose arguments determine the size of the rectangle, its initial location, the direction and distance to iterate, and finally a piece of code to execute at each location. This last piece of code had access to the results of various image operators over the window.

In addition to the *iterate* node and its associated image operator nodes, standard mathematical functions were provided, as well as flow control and five floating point memory registers, with associated read and write nodes.

Experiments

The robot used in this work (next page, at left) is 60 cm by 75 cm (2 feet by 2.5 feet), with a camera mounted at about eye height. For *data collection*, a simple wandering algorithm was developed that used sonar to avoid obstacles. The robot was run in two different buildings and the camera's video stream recorded.



The different environments presented slightly different challenges. While both were composed mainly of a textureless carpet, the FRC hallway contained two turns and burned out fluorescent lights in one part of the hallway. The NSH run included a person visible for several frames and a stripe of red carpet, which showed up as black in the camera.

Expanded Representation

Two sets of experiments were performed, one for each cycle through the outer loop. While there were a host of smaller differences, the main difference was in how much of the loop was under genetic control. In the first set of experiments the iterate nodes could move horizontally or vertically and had five arguments that determined the size of the window, the horizontal and vertical start location, the ending location, and finally the code to execute at each step. Each individual had three iteration branches, and an additional fourth branch that combined the results of the other three.

In this condition the evolutionary computation discovered vision algorithms that did only slightly better than the best constant approximation. That is, the program which ignored the image and always returned the bottom of the image did only slightly worse than the best evolved program. This was true despite a population size of 10,000 individuals run for 51 generations and a maximum size for individuals of 6000 nodes.

However, when a poorly performing hand written program was used to seed the population, the evolutionary computation was able to successively modify it to produce a very successful algorithm. The best such algorithm achieved an average error of only 2.4 pixels/column (the error in each column was limited to 20 pixels), and got 60% of fitness cases within 2 pixels of the human provided "correct" answer. Subjectively, the vast majority of fitness cases were handled more than well enough for obstacle avoidance, and the errors did not follow any particular pattern, suggesting a simple filter would eliminate most of them.

A navigation algorithm was then written which used the best evolved vision algorithm to control the robot. While it was run several times during development, no formal attempt was made to determine mean time between failures or failure modes. However, it did succeed in navigating the FRC hallway, making turns and avoiding previously unseen obstacles. This experience provided some small changes to the vision framework, such as moving the camera to the front of the robot and rotating it downward a little further, in order to image the area just in front of the wheels. Reflection on the successes and failures of the

entire process were then used to guide a redesign of the vision framework.

Focused Representation

While many aspects of the seed were modified by the evolutionary computation, others were not. In particular, the successful evolved algorithms all iterated vertically, from the bottom of the image to the top or vice versa. Therefore, horizontal iteration was eliminated and the iterate node simplified to take only three arguments: the window size, the horizontal location in which to iterate, and the code to execute at every step. The result producing branch, which had simply returned the result of a single iteration branch, was also eliminated.

To reflect this main difference between the two passes through the outer loop, the first set of experiments were dubbed the *expanded representation,* and the second set the *focused representation.* In the experiments on the focused representation, three separate experiments were run, the first using data from the Field Robotics Center (FRC), the second from Newell Simon Hall (NSH), and the third from both.

All three experiments produced individuals which achieved a fitness of greater than 85%. They did this despite burned out lights and other effects that caused the carpet's average intensity to vary from zero to at least 140s out of 255; despite large gradients caused by imaging artifacts; despite moiré patterns of image noise; and despite the shadow of the robot.

In all three experiments, the best individuals were more than good enough for navigation. Once again the vast majority of columns were interpreted correctly and with one exception errors were transient. The one exception was the stripe of red carpet in Newell Simon Hall, which was uniformly considered an obstacle, at least when near the robot. The best individual from the FRC runs, for example, could distinguish carpet from wall or door at the bottom of the image, and find the boundary between ground and non-ground, despite a burned out fluorescent light at the beginning of the run, carpet intensities that vary from black to at least the 140s out of 255, the shadow of the robot and large artificial gradient intensities on the carpet due to problems in the digitizing software.

The best individuals from the three experiments were simplified to discover how they worked. Evolution had exploited a number of techniques, including a sequence of if-then conditions similar to a decision tree but involving non-linear combinations of up to five different image terminals. In another case, a recurrent mathematical expression was evaluated once at each location of the window. In all cases, the bottom of the image was handled using different code than the rest of the image. This reflects a natural dichotomy in the images: at the bottom of the image a program must detect the presence or absence of an object, but in the middle of an image it must detect the *transition* between ground and non-ground. The mechanisms for this varied; in the FRC experiment it used two different branches for the two conditions, whereas in the NSH and combined experiments a multitude of *if* statements were used to run different code at different locations. Interestingly, the raw image was never used, although the median filtered image was. All directional gradients of the image intensity were used except the vertical.

Online Validation

A navigation subsystem was then created by hand which chose a direction to move based on the output of the vision subsystem. This subsystem was very similar to the one used during data collection, except that its inputs came from vision, not sonar. The three different vision subsystems, one from each experiment, all used the same navigation subsystem.

The robot was then run in the same environment(s) it was trained in. Videos of this *online validation* can be found at www.metahuman.org/Martin/Dissertation. These routes retraced the path of the training set and then continued much further. They included views of the same hallway from the opposite direction, as well as similar areas never seen during training. Objects were present that were not present during training, such as chairs, trash cans and people.

In general, the navigation system worked rather well. It used the same camera that was used for data collection, in the same location and orientation. The evolved algorithms worked well despite months of wear & tear on the carpet. Most errors were transient, lasting 1 or 2 frames, even when the camera was stationary. It worked on a range of iris settings and camera tilts. It did not overfit to column location, and twice as many columns were used during navigation as were used for data collection.

There were a few persistent errors. It was sensitive to small strips of paper or shiny pieces of metal. The red carpet, which was handled poorly even on the training data, was classified as an obstacle when nearby, causing the navigation to consistently treat it as a wall. Other errors would fool the navigation system only occasionally, although these were responsible for most collisions. Overall system performance, specifically time to collision, is roughly comparable to the published performance of hand-written systems.

Not surprisingly, the vision subsystem from the combined data set performed a little worse in each environment than the subsystem developed from only that data set. This produced a measurable but small drop in performance. It did not have a new class of error, but instead was more likely to make one of the errors made by the other vision subsystems.

Reflections

In presentations of this work to date, people have asked “would I recommend my approach to others?” The answer depends on exactly how the question is intended.

For someone whose main goal is to create a working robot as quickly as possible, and wants to reproduce this work with a different robot and environment, the answer is a close no. While the resulting programs work approximately as well as hand constructed programs, hand constructed programs are in all probability easier to write.

However, for a researcher looking for a new way to create visually guided robots, the answer is an unqualified yes. In fact, at least two groups are now interested in refining the ideas in this dissertation. The evolved systems already performs comparably to other techniques, and the future work section is full of suggested improvements, many of them straightforward.

The approach has many attractive properties. For most problems we can think of a number ways to approach them. For example, to visually track a lecturer from a camera we could use background subtraction, segmentation based on skin tone, gait analysis, etc. Given pen and paper we could even write these algorithms out in pseudo code. Genetic programming, as applied here, can be thought of searching the space of such pseudo code. This allows researchers to focus on the representation, leaving the details of how its applied to the evolution. And since existing vision algorithms are typically expressed as programs, incorporation of existing knowledge is easier than other forms of machine learning.

Were the subsystems produced by the evolutionary computing different than those produced by hand? There were certainly superficial differences, most of which were removed by automatically simplifying the evolved individuals. Some differences remained, such as complex nonlinear conditions in the decision tree, or the use of recurrent mathematical expressions. These certainly used more image statistics, and in more complex combinations, then any previous work. Does this extra complexity actually lead to increased performance, or is it simply an awkward way of expressing a simple idea? In other words, could it be simplified further? Until we can analyze these evolved individuals further, we can not know.

However, this dissertation strongly suggests such analysis is possible. The high level structure of the individuals, be it a decision tree or recurrent formula, is straightforward, as is the interpretation of the actions in the decision tree. Thus, great insight can be gained by determining where the various conditions are true or false. Similarly, looking at how the variables in the recurrent expression change over time can shed light on how they work. That the analysis of individuals reduces to such simple tasks is a strong argument that the inner workings of evolved individuals are most likely within our grasp.

Even if no fundamentally new algorithms were evolved in this dissertation, greater computing power or modifications to the evolutionary framework may produce solutions of unmatched subtlety or complexity, beyond the limits of what people could produce.

Perhaps most importantly, this dissertation demonstrates that the proposed research program works. In other words, we can use genetic programming to create a vision subsystem, and such a subsystem can be used for robot obstacle avoidance. We can also learn enough from doing it that we can improve the system, increasing its performance greatly.

Future Work

Several improvements suggest themselves. The errors in the vision subsystem, apart from the red carpet, happened in locations or objects that were not present during training. Therefore, a more diverse training set would likely overcome these.

To use a bigger training set without taking more computer time, only a subset of training images can be used for each evaluation, with the composition of the subset changing over time. The subset can be chosen probabilistically, where the probability of being selected is higher for images that confused the previous generation of vision programs. Such a setup is called *co-evolution* and has been shown to be effective in several problems. In fact, by over-representing difficult

cases, there is reason to believe it may encourage the evolving programs to handle the red carpet.

Currently the algorithms can not maintain state from one image to the next, which means that the resulting algorithms are reactive. Adding a mechanism to maintain state may result in greatly improved, robust algorithms.

As for the navigation subsystem, the similarly between sonar and the output of the vision subsystem suggests using techniques that work with sonar, such as evidence grids, Monte Carlo localization and other Bayesian techniques, not to mention traditional evolutionary robotics frameworks such as evolving neural networks. Sensor fusion should be easy in an evolutionary robotics framework, and combining sonar with vision could prove an easy and useful first step.

Evolving the vision and navigation subsystems simultaneously could lead to the development of active vision, wherein a robot avoids areas where its vision algorithm fails, or views a confusing object from multiple locations. Perhaps even more exciting would be the inclusion of a learning algorithm that operates within a single run. While evolutionary computation is probably too slow for this purpose, the parameters and details of a more traditional learning algorithm could be evolved.

Finally, the analysis and understanding of evolved programs could provide powerful insight into their workings and the workings of the evolutionary process itself. Simplifying programs can make them much easier to understand. While simplifications were partially automated in this dissertation, it would be interesting to fully automate the process and apply it to the ancestors of the best evolved individuals, or to runs that performed poorly, or to other large groups of individuals. The simplifications are close cousins of compiler optimizations. Thus, another advantage of the representation is that decades of work in compiler optimization can be leveraged. In addition, by looking at which parts of the program or memory were used in different image locations, new insights can be gained.

Philosophy and Manifesto

While the author believes that this dissertation stands on its technical merits alone, the most exciting aspect of the work is that it represents a first step in a research program which answers the common critiques of Artificial Intelligence, and thus has the potential to go further than previous research programs, perhaps achieve the original goal, human-level artificial intelligence.

Perception is certainly a key mental ability and human perception is very intricate and subtle. Human vision does not simply recover objective properties of the image. Computer vision has largely focused on bottom up processing, but human vision seems to find the best answer given the sensor data, context, domain knowledge, task knowledge, previous state, etc. From the bottom up point of view, the vision problem is under constrained, but it seems better viewed as ‘over suggested.’ This means it sometimes sees things that are not there, as is demonstrated by optical illusions. When interpreting an illusion that is a mistake, but in natural images it is the right thing to do.

These properties of human perception have been known for a long time, yet have been largely ignored in computer vision of unstructured scenes. Incorporating them is thought to make a vision subsystem much

more complex. Thus, the real problem is complexity. Building a vision subsystem that is task and context free simplifies the design of complex systems allowing separation of labour and knowledge and allows one subsystem to be used in many other systems. It should be noted that the points in this paragraph are a product of research culture and the fact that people are creating programs.

While being task and context free is a great way to manage complexity, it comes at a cost. When one system takes the output of another system at face value, then any errors in the second system mean the program is working under false assumptions, so any action is possible. Also, if you throw away all of the information from your sensors except its best guess of the range to different locations, you are throwing away all the information that can help you decide between the alternatives, and sensor fusion becomes difficult if not impossible.

To summarize, interdependence is good, and readable, conceptually clear design leads to a context free/task independent approach. This leads us to the unorthodox conclusion that messy, unreadable, difficult to understand code is necessary for something with the subtlety and complexity of the human mind. Therefore, since having people write the code adds all these constraints, machines should create the design. This is a relatively novel application of machine learning: designing the architecture.

This suggests that we explore how evolutionary computation can be used to create computer vision subsystems. Others have argued for the importance of embodying intelligence in unstructured real worlds. Thus we have arrived at our starting point: how to use evolutionary computation to create robot vision for an unstructured environment.

How To Read This Dissertation

Readers with different backgrounds and interests will find certain chapters of this dissertation more useful than others. What follows are suggestions of what each group should read.

Those interesting in reproducing these results, presumably with a different robot and environment, should read this overview then the chapters “Technical Framework,” “Offline Learning: Basics,” “Focused Representation” and “Online Validation.” Those wishing to extend this work should read those chapters as well as the “Discussion” chapter. If more details are desired, either type of reader should consult the “Related Work,” “Data Collection” and “Expanded Representation” chapters.

The “Philosophy & Manifesto” chapter should be read and reflected on by anyone interested in the original goal of artificial intelligence: the understanding and creation of intelligence. This hopefully includes most people in the field itself, as well as others in computer science and elsewhere. It can be read without reading the rest of the dissertation or even this overview, in fact, it was the introductory chapter in an earlier draft.

Part I

Motivations

2 Related Work

This dissertation explores the use of evolutionary computing (EC) to produce an embodied vision-based robot. Perhaps the most basic competency that an embodied perception system needs is the ability to move around its environment without bumping into objects. Therefore, the chosen task is obstacle avoidance using vision. In this task, the robot uses the output of a video camera to decide which direction to move in order to avoid any obstacles.

While many people have worked on various vision algorithms, it turns out only a handful have ever tested them by actually constructing a working obstacle avoidance system from them. In this chapter, the existing systems are described, along with other attempts to apply evolutionary computation to robotics.

Evolutionary Robotics

The simulated evolution of executable structures has a long history. In 1950 Turing described the possibility [*Computing Machinery and Intelligence*], and in 1958 Friedberg [*A Learning Machine: Parts I & II*] evolved sequences of machine language instructions that performed modest computations. Fogel, Owens, and Walsh [1966] introduced Evolutionary Programming, evolving finite state automata to predict symbol strings generated from Markov processes, in the 1960s. Holland [1975] also introduced the bitstring Genetic Algorithm in the 1960s. However, it was not until the mid 1980s that even toy problems of any depth could be attempted.

Much of the current work on evolving executable structures follows the work of John Koza and James Rice [Koza 1992, Koza 1994] under the name of Genetic Programming (GP). Using a LISP-like representation, Koza extended the work of Cramer [1985] which demonstrated that parse trees could provide a natural representation for evolving programs. Koza applied this technique to a broad range of problems, including many in robotics, although the two books describe only toy problems.

Koza and Rice evolved controllers, emergent foraging behaviors, and subsumption programs among others. While the results were encouraging, their simulations did not model sensor or actuator noise and were not intended to be realistic. Other work on evolving programs to control simulated robots can be found in [Koza 1991], [Langdon 1987, *Proceedings of Artificial Life*], [Langdon *et al.* 1991, *Proceedings of Artificial Life II*] and [Kinnear 1994b, *Advances in Genetic Programming*].

In 1992, Rodney Brooks discussed the issues inherent in transferring programs evolved in simulation to run on a real robot [*Artificial Life and Real Robots*]. In particular, he suggested using GP to evolve behaviors for behavior based robots, an idea he attributes to Chris Langdon. In summing up the state of the art, he said “There have been no reports to date of programs evolved for embodied robots.” That was soon to change.

Experiments

Evolutionary Robotics is an infant research field that uses simulated evolution to produce control programs for real robots. Recent work can be found in [Husbands *et al.* 1998, *Evolutionary Robotics*] and [Nolfi *et al.* 2000, *Evolutionary Robotics*]. Most work uses bitstring genetic algorithms to evolve recurrent neural nets for obstacle avoidance and wall following using sonar, proximity or light sensors. Significantly, recurrent neural networks, where the outputs of later layers can feed back to earlier layers, are traditionally considered more difficult to train than feed forward nets, since the derivative of the error is difficult or impossible to compute analytically. Evolutionary Computation does not use gradient information, and therefore even exploratory, toy problems use recurrence.

For example, Miglino *et al.* [1995] evolved a four-layer recurrent neural network that allowed a mobile LEGO robot to explore the greatest percentage of an open area within an allotted number of steps. Jakobi *et al.* [1995] evolve arbitrarily recurrent neural networks for obstacle avoidance and light seeking behaviors for a Khepera robot, equipped with eight infrared sensors. Meeden [1996] evolved recurrent neural controllers for a toy car that had to avoid obstacles (walls) while seeking or avoiding light. For sensors the car was equipped with four touch sensors and two light sensors. Despite its large turning radius, lack of position estimation and inability to sense oncoming walls, successful programs evolved in a variety of experiments.

Yamauchi and Beer [1994] evolved a continuous-time fully-connected recurrent neural network to recognize a landmark. The robot moved around the landmark and recorded the readings of a single sonar sensor over time. When transferred onto their Nomad 200 robot, the best evolved net correctly classified the landmarks in 17 out of 20 test trials.

Nordin *et al.* [1998] report the only work that this author is aware of that applies genetic programming (as opposed to any other form of evolutionary computation) to real robots. A Khepera robot is used, in a physical environment constructed especially for the robot. In this environment, obstacles and walls are white so they reflect the infrared sensor's light. They directly manipulate SPARC machine language in symbolic regression to predict the "goodness" of a move given the current sensor values. For obstacle avoidance, the "goodness" of a move is the state of the robot 300ms in the future, specifically the sum of the proximity sensors, plus a term to reward it for moving quickly and in a straight line.

To decide which action to take, they simply predict the goodness of all possible actions, choosing the one with the best score. The training data is collected while the genetic programming is running, i.e. the training set changes during evolution, and the best individual so far is used to steer the robot. They use a population size of 10,000 and find that, in runs where perfect behavior is developed, it developed by generation 50.

Most work evolves robots in simulation, then transfers the best individuals to real robots. Reynolds [1994] has pointed out that without adding noise to a simulation, EC will find brittle solutions that would not work on a real robot. Jakobi *et al.* [1995] discovered that if there is significantly *more* noise in the simulation than on the real robot, new random strategies become feasible that also do not work in practice.

There are no reports of anyone exploring the reverse, i.e. whether entire classes of solution will not be found because they do not work in simulation. For example, dead reckoning error may be different on carpet then on a hard floor, so one possibility is to try to distinguish between them based on, say, their visual appearance. If this difference in error is not modeled in the simulation, such a solution will never be found. Once evolutionary robotics advances beyond its current basic stage, programs evolved in simulation may miss many subtle solutions.

As well, there have been no reports of anyone in evolutionary robotics attempting to simulate CCD camera images, either using standard computer graphics techniques or morphing previously captured images. A group at the University of Sussex in the U.K. has used video images to guide their robot [Harvey *et al.* 1997, *The Sussex Approach*], but every video image was first reduced to three real numbers, the average intensity over three circles. These are significantly easier to simulate than a full CCD image, especially when the only objects are pure white on a black background. The location and size of the circles was determined genetically, as was an artificial neural network to turn the values into motion commands. Using a population of 30 individuals, they evolved movement, target finding and the discrimination of a triangle from a square all within 30 generations. The targets, triangle and square were all white on a "predominantly dark environment." Because of the difficulty of accurately simulating a video image, for example using computer graphics techniques, all of their evaluations took place on their robot, a video camera attached to a 2D gantry. The gantry, which they described as "part way between a mobile robot and simulation," allowed them to accurately position the camera anywhere within a rectangle.

The evolved rectangle/triangle discriminator has an interesting structure. The rectangle and triangle are in fixed orientations against a wall, the rectangle axis-aligned, the triangle pointing "up." The best individuals used two of the three circles, one directly above the other. When looking at the rectangle or background, both sensors would have approximately the same value, but when looking at the triangle, the bottom one would register "white" while the top "dark." The evolved programs rotated in the first case, but drove straight in the second.

Smith [1998] evolves an artificial neural network in simulation for a soccer robot to approach a ball and push it toward a goal. He used a Khepera robot with a one dimensional auto iris vision system that returns 16 pixels of greyscale (intensity) information, arranged horizontally. The 16 pixels were actually derived from 64 pixels; Smith does not say how. This is an important step toward simulating camera images, but again much easier than simulating a CCD image at, say, 160 x 120 pixels or above.

In order for the best simulated robots to work in the real world, he found it necessary to force the robot to ignore certain visual features by making them unreliable. Individuals start scoring around generation 50, but even by generation 3000 the best individuals score only 25% of the time in simulation. Of twenty trials on the real robot, the ball was moved 18 times and the robot scored 4 times.

A few research groups perform all fitness evaluations on the real robot. Floreano and Mondada [1994] evolve recurrent neural networks for obstacle avoidance and navigation for the Khepera robot. It took them 39 minutes per generation of 80 individuals, and after approxi-

mately 50 generations the best individuals were near optimal, moved extremely smoothly, and never bumped into walls or corners.

Naito et al. [1997] evolved the configuration of eight logic elements, downloading each to the robot and testing it in the real world. Finally, the Sussex gantry robot [Harvey *et al.* 1997] mentioned earlier has used evaluation on the real robot. They used a population size of 30, and found good solutions after 10 generations.

The only reported example of evolving visual obstacle avoidance was done by Baluja [1996], who evolved a neural controller that interprets a 15 x 16 pixel image from a camera mounted on a car, similar to ALVINN [Pomerleau 1989, *ALVINN*]. The network outputs were interpreted as a steering direction, the goal being to keep it on the road. Training data came from recording human drivers.

In summary, previous experiments in Evolutionary Robotics have used low bandwidth sensors, such as sonar and proximity sensors. There were typically less than two dozen such sensors on a robot, and each returned at most a few readings a second. This is in contrast to much traditional work in computer perception and robotics which uses video or scanning laser range finders, that typically have tens to hundreds of thousands of pixels, and are processed at rates up to 10 Hz or more. Evolutionary Robotics has much to gain by scaling to these data rich inputs.

In addition, most Evolutionary Robotics has designed algorithms for real but simplified environments that are relatively easy to simulate. While evaluating evolved programs on real robots is considered essential in the field, those environments are typically still tailored for the robot. The current work extends the state of the art by evolving programs to interpret large amounts of data from unstructured environments.

Arguments

Along with systems built, various practitioners have argued for how they should be built. There are two main "philosophical" papers, one by the Sussex group, and another by Rodney Brooks. This section describes the arguments therein, postponing discussion until later chapters.

In [Harvey *et al.* 1992, *Issues in Evolutionary Robotics*], the Sussex group argues against the evolution of computer programs for controlling mobile robots, preferring artificial neural networks instead. They echo Varela, Tompson and Rosch in saying that robots are best viewed as "dynamic systems rather than computational systems that are perturbed by their interactions with the environment." They also argue that "the primitives manipulated during the evolutionary process should be at the lowest level possible," because they believe "that any high level semantic groupings necessarily restricts the possibilities available to the evolutionary process."

The same paper also makes a strong case for evolutionary robotics using vision, and using real, not simulated, camera images. They argue that the proximal nature of touch sensors and sonar force robots to employ primitive navigation strategies such as wall following. In contrast, "The need for more sophisticated navigation competencies, which we take as manifest, is only likely to be overcome fully by an increased reliance on *distal* sensors—in particular, vision." [Emphasis in the original.] They also argue that the computational demands of simulating

vision in a useful manner soon become considerable. After discussing what sort of camera a robot would need, they say "Envisaging what is necessary for the robot is likely only to be possible after some experience with it: a circularity which reveals that some iteration is required between the simulation work and the building of real robots—a pluralist approach will be the most fruitful."

In [Brooks 1992, *Artificial Life and Real Robots*], Brooks argues for the evolution of behaviors in behavior based systems. He argues that behavior based robot programs take three orders of magnitude more memory to represent than programs typically evolved at the time. He points out the vast difference between simulated robots and physical robots and their dynamics of interaction with the environment. He points out the dependence of the structure of the search space on the representation used. "Careful design is necessary." He also argues that nature co-evolved the hardware and controllers in a way that arguably cut down the size of its search space.

Interestingly, Brooks also said "to compete with hand coding techniques it will be necessary to automatically evolve programs that are one to two orders of magnitude more complex then those previously reported in any domain. Considerable extensions to genetic programming are necessary in order to achieve this goal." By Moore's law, two orders of magnitude would have taken about 10 years, and since the paper was written in 1991, this places the projected date at 2001—the year this dissertation will be defended.

With the state of the art in evolutionary robotics now understood, our attention shifts to more traditional work in vision based obstacle avoidance.

Visual Obstacle Avoidance

Somewhat surprisingly, there have been only a handful of complete systems that attempt obstacle avoidance using only vision in environments that were not created for the robot. What is more, it has never been done reliably: the average distance between failures (i.e. hitting something) is around 400 to 500 meters, for the best systems. At typical speeds of 30 cm/s, this is less than 30 minutes between failures.

As an aside, average distance between failures is, of course, a horrible metric. A robot that "explores" by travelling in circles could go indefinitely. And in some environments obstacles may be more common, or harder to detect, than others. But as with "lines of code" as a metric for programmer productivity, it predicts more substantial metrics as long as people's reward is not based on it. And the main point is manifest: with the current state of the art, you certainly could not trust a vision-only robot in an unstructured environment for even a day.

Perhaps one reason for such disappointing performance is that only a handful of such systems have been built. While there has been much work on creating vision systems, we can not know how well they will work or what the failure modes will be until the loop is closed by using them to control a robot. And most systems which do close the loop use some range sensor, typically sonar (for indoor, e.g. Xavier & Amelia) or laser (for outdoor, e.g. Navlabs & Sojourner). Those which close the loop using vision alone either use stereo vision, optical flow, or more ad hoc methods. All three categories are described below.

Stereo Vision

Stereo vision attempts to tell the distance to part of the environment by finding the image of it in two (or more) cameras. If we know the location and orientation of the cameras and their field of view, and if they follow a perspective projection, we can triangulate and find the desired location. If there are only two cameras, they typically share the same orientation, and are displaced horizontally, like the eyes in our head.

To find the corresponding images, a small window in one image (8 by 8 pixels is typical) is compared against a similarly sized window in the other image(s). Typically, the second window is moved along a horizontal line, and at each location is compared to the window in the original image. The window that is the most similar is assumed to be the image of the same object. Similarity is usually some context free mathematical expression, such as the sum of squared differences of corresponding pixels or a cross correlation.

Other than the need for calibration, which is generally considered achievable, the main problems encountered with stereo result from mistaken correspondences. If a given part of the environment does not have much visual texture, such as a wall painted a solid color, then any window on it will look the same as any other window, and stereo will match noise or intensity of specular reflection. If a pattern repeats throughout an environment, such as fence posts or a row of filing cabinets, the wrong one may be selected as a match. Finally, near the boundaries of objects, the background can be seen by one camera and not the other. In this case the matching image simply does not exist, and the object in the mismatch that looks most similar is selected.

Nevertheless, working obstacle avoidance systems have been constructed that use stereo vision as their heart. Two of these are described below.

Larry Matthies' group at JPL has built a number of complete systems, all using stereo vision [Matthies *et al*, 1995, *Obstacle Detection for Unmanned Ground Vehicles*]. They first rectify the images, then compute image pyramids, followed by computing the sum of squared differences, filtering out bad matches using the left-right-line-of-sight consistency check, then low pass filter and blob filter the disparity map.

Their algorithm has been tested on a number of robots. The Mars rover test bed vehicle "Robby" accomplished a 100m autonomous run in sandy terrain interspersed with bushes and mounds of dirt [Matthies 1992, *Stereo vision for planetary rovers*]. And the JPL HMMWV drove 200m continuously at 1 to 3m/sec while detecting half-meter obstacles in time to stop. Figure 1 shows results from a test run that travelled about 200m along a road. From Matthies *et al*. 1995*:*

> The vehicle was instructed to follow its current heading whenever it could, but to swerve to a new heading as necessary to avoid obstacles. Two significant swerves are seen in this run. The first was to avoid a tree on the right side of the road; the second was to avoid a bush on the left side of the road, just beyond the tree. This was a very successful run, and typical of results obtained in this kind of terrain.

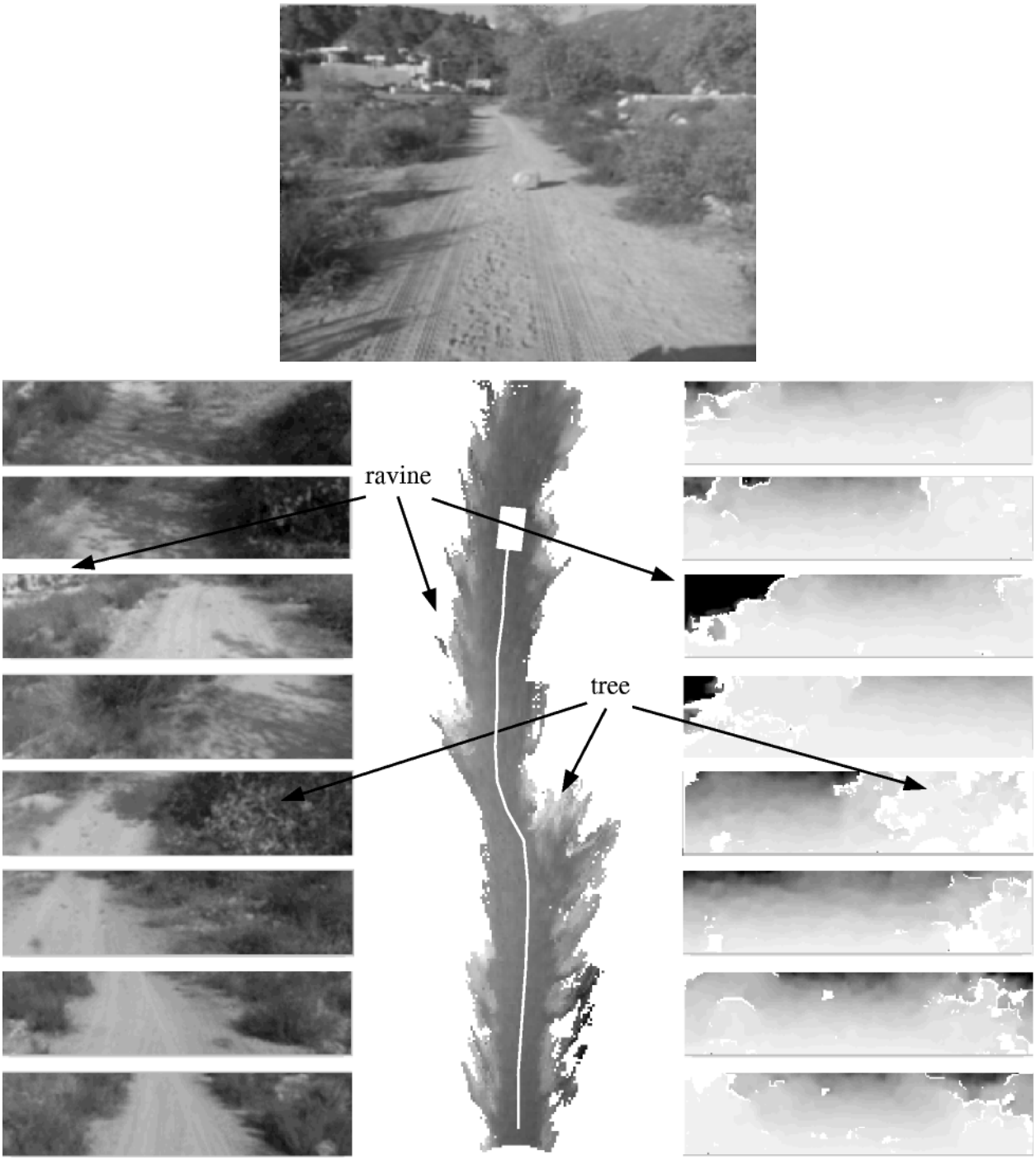


Figure 1: Road run with Ranger, covering about 200 meters. At top is an overview of the road travelled. Shown below are windows of attention from representative intensity images (left side), corresponding range images (right side), and a composite elevation map from the whole run (center). The white curve and rectangle on the elevation map represent the vehicle path and the vehicle itself at the end of the run. From Matthies *et al.* [1995, *Obstacle Detection for Unmanned Ground Vehicles*]

The HMMWV has also accomplished runs of over 2 km without need for intervention, in natural off-road areas at Fort Hood in Texas [Matthies, personal communication]. The low pass and blob filtering mean the system can only detect large obstacles; a sapling in winter, for example, might go unseen. Unfortunately, nothing more is said about the performance of the system, or its failure modes. Sadly, this seems to be the norm in the reporting of such research.

The Ratler trials [Krotkov *et al.* 1995 and Maimone, 1997] are part of the Lunar Rover Initiative [Katragadda *et al.* 1996], aimed at developing technologies for autonomous and teleoperated exploration of the moon. Specifically, the goal is a 200 km traverse of the lunar surface that would visit several historical sites. To accomplish this, a stereo algorithm was developed and implemented on the Ratler vehicle, along with an obstacle avoidance planner and arbiter. After rectification, the normalized correlation is used to find the stereo match. The match is rejected if the correlation or the standard deviation is too low, or if the second best correlation is close to the best.

Using this system, Ratler was run at three different planetary analog sites: the Pittsburgh Slag Heaps, the Robotics Engineering Consortium, and the Moon Yard. Travelling at 50 cm/sec over 6.7 km the system had 16 failures, for a mean distance between failures of 417 m. No information on failure modes is available.

Correlation Based Optical Flow

Correlation based optical flow is similar to stereo, except that the two images are taken by the same camera at different times, and the separation between cameras is no longer strictly a sideways translation. It also is not known very precisely, since even a small wobble in the camera mount can cause motion of a few pixels in magnitude. Therefore, the search for correspondences can not be restricted to a horizontal line in the image and is typically two dimensional. To reduce computational expense, the images are typically much closer together than in stereo and therefore the correspondence problem is easier, but the estimates are less accurate.

The Integrated Systems Division at NIST has succeeded with runs of up to 26 minutes without collision [Camus *et al.* 1999, *Real-time Single-workstation Obstacle Avoidance Using Only Wide-field Flow Divergence*]. Their system uses correlation based optical flow, specifically, divergence of image flow in the direction of the camera's heading, which is inversely related to time-to-contact. Two onboard cameras are mounted one above the other on a single pan motor, and processed separately. The top camera has a 40° field of view, and the lower camera a 115° field of view. The divergence and time-to-contact are computed in three overlapping windows of the central image. In the wide angle image, the process estimates maximum flow in two peripheral visual fields (left and right). Maximum flow and time-to-contact are temporally filtered to reduce errors. Recursive estimation is used to update current flow and time-to-contact estimates and to predict flow and time-to-contact at the next sample time.

Using active gaze control, the cameras are rotational stabilized so that their motion is approximately a translation. If the flow is larger on one peripheral side than the other, then objects in the scene are closer on

that side, and the robot steers away. When the camera points too far away from the heading, a saccade is made toward the heading. From [Coombs *et al.* 1997]:

> Experiments with the obstacle avoidance system were conducted in a laboratory containing office furniture and robot and computing equipment. Furniture and equipment lined the walls and there was free space roughly 5 m by 3 m in the center of the lab. Office chairs provided obstacles. ...
>
> Variability in steering and stopping depends in part on the textures visible from a particular approach. Stopping distances may vary just because the angle of approach varies. As time-to-contact and peripheral flows are computed, the robot moves forward toward open areas while centering itself in the open space. Upon detection of an imminent collision, it turns to avoid the obstacle and continues its wandering behavior. The system has run successfully for up to 20 minutes.
>
> There are three primary factors that lead to system failure. First, there is a variable time delay between detection of an event and the behavioral response to this event. ... Second, the flow computation assumes that there will be sufficient texture in the field of view from which to compute flow. If this is not the case, no flow is detected and the robot collides with the obstacle. The third cause of failure is the rectangular geometry of the robot base and the presence of bumpers which trigger stopping upon contact. There are instances where the robot is turning to avoid an obstacle, and in so doing, brushes its bumper against a nearby object.

The authors point out that the gaze stabilization, which removes camera rotation with respect to the environment, is crucial for their technique. Even small camera rotations can produce rotational flows that are much larger than translational flows.

Ad Hoc Methods

There are also a number of "scruffy" methods that eschew the mathematical rigor of the "neat" methods above, and instead rely on ad hoc algorithms.

Ian Horswill's Ph.D. thesis [Horswill 1994, *Specialization of Perceptual Processes,* and Horswill 1993, *Polly: A Vision-Based Artificial Agent*] was the programming of the robot Polly to give tours of the seventh floor of the AI Lab at MIT. The obstacle avoidance software exploited the texturlessness of the floors, and labeled every area whose texture was less than a certain amount as floor. Starting at the bottom of the image and moving up, the first textured pixel in each column was found, and declared to be an object. The closer such pixels were to the bottom of the image, the closer the object was to the robot. By telling what side

of the image the object was on (left or right), the robot could veer away from it.

From Horswill 1995:

> In general, all low-level navigation problems are obstacle detection problems. Fortunately, most of these are false positives rather than false negatives so the system is very conservative. The system's major failure mode is braking for shafts of sunlight. If sunlight coming through office doors in into [*sic*] the hallway is sufficiently strong it causes the robot to brake when there is in fact no obstacle. Shadows are less of a problem because they are generally too diffuse to trigger the edge detector.
>
> False negatives can be caused by a number of less common conditions. The present system also has no memory and so cannot brake for an object unless it is actually within the camera's field of view. Some of the objects in the lab have the same surface reflectance as the carpet on which they rest, so they can only be distinguished in color. Since the robot only has a black and white camera, it cannot distinguish these isoluminant edges. The edge detector can also fail in low light levels. Of course, most vision systems are likely to miss an object if they cannot even find its edges so this failure mode should not be surprising.
>
> ... Given the current state of vision technology, it is a bad idea to rely exclusively on vision for obstacle avoidance...

A main contribution of the work is the idea of specialization. From [Horswill 1994]:

> We can analyze this relationship [between agents and their environments] formally by deriving an agent that is specialized to its environment from a hypothetical general agent through a series of transformations that are justified by particular properties of the agent's environment. In performing the derivation, the designer divides the agent's specialization into a discrete set of reusable transformations, each of which is paired with an environment property that makes it valid. I call such properties "habitat constraints" because the set of such constraints define the agent's habitat.

Note that these are hard constraints: they are built into the very design of the robot. While these constraints are extremely helpful most of the time, the robot's most troublesome failure modes occur where these constraints are violated. No attempt is made to detect possible exceptions, or anything else that might turn these into soft constraints.

Liana Lorigo's Master's and now Ph.D. work under Rodney Brooks and W. E. L. Grimson follows on Ian Horwill's work [Lorigo 1996, *Visually-guided obstacle avoidance in unstructured environments* and Lorigo *et al.* 1997, *Visually-guided obstacle avoidance in unstructured environments*]. The monocular camera returns 64x64 pixel colour images. The analysis starts by computing a histogram over the 20 (wide) x 10 (high) pixel window in the lower left. The statistic used in the histogram varies; it is either the pair (red/intensity, green/intensity), the pair (hue/intensity, saturation/intensity), or brightness gradient magnitude. This histogram is the "reference histogram" for this column. It is assumed that this area is free of obstacles (or, perhaps, that if an obstacle is that close, nothing can prevent a collision). The window is moved up one pixel and the histogram is recomputed. If it differs from the reference by more than some fixed amount, the new window is said to contain an obstacle and processing stops. Otherwise, the window is moved up one more pixel and the computation repeated.

This gives the location of the obstacle closest to the bottom of the image, for that 20 pixel wide column. The process is repeated for all the columns in the image (that is, starting at horizontal pixels 2, 3, ... 45, not pixels 21, 41, ...). For each column we then have a row number; a flat ground plane assumption is made to translate these into distance to nearest object in that column. Given this, the robot then turns away from the side with the cumulative nearest object.

This system has been tested for more than 200 hours in diverse environments, including test sites at the MIT AI Lab and two simulated Mars sites at JPL. From Lorigo *et al.* 1997:

> For testing in the sandbox, the obstacles were moved into many configurations, additional obstacles were added, and people interacted with Pebbles by stepping in its way. Normally the space between obstacles was only slightly larger than the robot's width. In this situation, the robot navigated safely for large amounts of time. ... The run-time of the robot was limited primarily by hardware concerns and occasionally by the failure modes mentioned below.
>
> Further, the system avoided obstacles in the lounge. Walls, sofas, boxes, chairs, and people were consistently avoided. Corridor following, even at corners, was easily accomplished by the system. Again, moderate lighting variations caused no difficulty. These results were repeated in other rooms where differences included the type and amount of clutter and the pattern of the carpet.

Failure modes include objects outside the camera's field of view, especially when turning. The authors suggest that a wider field of view would solve this problem. Other failure modes:

> [C]arpets with broad patterns or boundaries between distinct patterns resulted in false alarms. Sharp shadows also posed a problem in bright outdoor sunlight when shadows were sometimes classified as obstacles. Similarly, bright specularities on a shiny floor

> occasionally caused the system to falsely report obstacles. Prior knowledge of such patterns or an additional method for depth estimation would be required to resolve these issues.

In this work, an object is anything different from the reference window. This incorporates a kind of domain knowledge, namely that safe ground has approximately the same histogram in all places. However, this is implemented as a "hard constraint": there is no room for exceptions. While this works most of the time, the failure modes above occur when this constraint is violated. One could imagine, for example, trying to recognize exceptional circumstances and developing new rules for them, or having a learning technique that does this automatically.

Illah Nourbakhsh has created a system that uses depth from focus to provide a rough depth map of a scene, and created an obstacle avoidance system based on this [Nourbakhsh 1996, *A Sighted Robot*]. Three cameras are mounted vertically, as close together as possible, and focused at three different distances, namely 0.4m, 1m and 2m. The image is divided into 40 subwindows, 8 across and 5 down. In each subwindow, the sharpness (i.e. how much the intensity changes from pixel to pixel) is computed for the three cameras, with the sharpest giving its focal length to the depth map. The result, then, is an 8 by 5 depth map, where each measurement is "near", "medium" or "far". On flat ground the bottom two rows are medium while the others are far; detecting more than one far in the bottom two rows signals an impending drop off (such as stairs), and the robot turns 180 degrees. Otherwise, the robot turns away from whichever side has the largest number of closes or mediums.

From Nourbakhsh 1996:

> Further indoor tests were conducted in the second and third floor hallways and lounge areas of the Computer Science Department. These tests were the most challenging: lighting conditions and wall texture vary greatly throughout the area. Additional risks included two open staircases and slow-moving students who actively tried to confuse the robot into falling down the stairs.
>
> The robot performed extremely well in this complex indoor domain, avoiding the staircase as well as the students. The robot can reliably navigate from inside a classroom, through the doorway, into the hallway, past the stairs, and into the lounge with perfect collision avoidance in spite of moving students. ... In all three runs [of 20 minutes each], the robot operated fully autonomously and the only environmental modification involved the removal of one coffee table in the lounge that violates our 'beheading' constraint. Average speeds in this domain were approximately 8 inches per second [20 cm/s]. ...
>
> Over several weeks of testing, accumulating more than 15 hours of outdoor time, the robot detected dropoffs and static obstacles with 100% reli-

> ability. Furthermore, false positive detection of steps proved to be essentially nonexistent.
>
> Our final experiment involved an outdoor demonstration of the robot... The robot approached the dropoff and the staircase more than fifteen times, detecting them with 100% accuracy. ... Over the course of the demonstration, the robot came in contact with no static obstacles and contacted a moving obstacle (i.e. a human) only once.

Due to its low resolution depth map, the robot has trouble seeing tables near camera height. As well, in areas of low texture, sharpness is low in all three cameras. Such areas are classified as close for safety, but a large patch of such area (such as a plain wall) could paralyze the robot.

Jimmy (Hajime) Or at the University of Tokyo built what he calls a "biomechatronic robot" [Yam 1998, *Roaches at the Wheel*] while working on his master's degree at the University of Tokyo. After taping down an American cockroach, he inserted fine silver wires into the extensor muscles of the hind legs. The roach was then allowed to run on what amounts to a trackball. The wires picked up the weak electrical signals generated by the muscles, and the signals were amplified and fed to the motorized wheels. In this way, the machine would mimic the speed and direction the cockroach ran. For the relation of this to the present work, see the next subsection.

Analysis

As can be seen from the above examples, the problem of navigating from vision alone has yet to be done well enough to leave a robot unsupervised for hours in an unstructured indoor environment. Note that the planetary analog environments tend to be easier than indoor environments, since obstacles tend to be large and have texture, as opposed to objects like tables at eye level or plain walls. As well, planetary environments are static whereas indoor environments are often filled with people.

Looking back on the above examples, it is not clear that a single depth cue is sufficient for the job. Area based stereo, for example, may simply have too many errors to be reliable for obstacle avoidance. That statement can never be proven, of course, since some day someone may think of a unique twist which makes it work. But from the current state of the art, the sufficiency of stereo is far from certain.

Since ranging devices like sonar (or cockroaches) make the obstacle avoidance problem much easier, they may seem like an obvious way to solve the problem. However, there are two reasons to shy away from them. First, they have many problems of their own. For example, sonar has as specular reflection (many surfaces act as mirrors to ultrasound), poor angular resolution, and an inability to see drop offs. But more importantly, the goal of this dissertation is not simply to engineer a prototype obstacle avoidance system, but rather to figure out how to build a better vision system. It is the new techniques that need to be discovered that are of interest, rather than obstacle avoidance itself. While creating robots as competent as the cockroach would be a breakthrough in vision, using the cockroach without understanding would not.

While reliable obstacle avoidance from vision may be impossible today, there is a simple proof that it is possible: people do it. With the motivation and background in hand, the next chapter describes the high level design of the experiments.

3 Technical Framework

When people design algorithms, much time is spent in *iterative design*, namely instantiating a design, applying it to a problem, and then revising the design based on what did and did not work. This dissertation automates this processes, by using a machine learning framework that manipulates traditional computer programs: Genetic Programming (GP).

To test this idea, one particular embodiment of it was chosen and applied it to visual obstacle avoidance for an indoor mobile robot. Visual obstacle avoidance is the task of using a video camera on a robot to detect and avoid objects.

This chapter introduces the technical framework of the actual experiments. It presents the high level description of system and justifies it. To put it differently, given the goal "use Genetic Programming to create a visual obstacle avoidance system," there are many different ways to flesh it out, and ten different researchers would most likely do it ten very different ways. This chapter describes the aspects of the approach that a researcher in the field could not guess from the one sentence description. These aspects should therefore be considered part of the contribution. In contrast, the contents of the following chapters are more or less typical ways of elaborating these ideas.

The plan is to harness Evolutionary Computation (EC) to construct an embodied vision system. When applying EC to a specific problem, the two main decisions to be made are the method of evaluating individuals, and the method of representing them. These are examined in turn.

The Evaluation Method

It is common practice in Evolutionary Robotics is to evaluate potential robot control programs by having them control a simulated robot in a simulated environment. A virtual world is created, and whenever a camera image is needed, computer graphics techniques could be used to render it. When the robot is commanded to move, its position in the simulated world is updated, modeling actuator error by moving it to a slightly different position than was commanded.

This use of simulation is attractive for a number of reasons. First, recent ray tracing and radiosity techniques can simulate many of the problems of real images, such as specular reflection of fluorescent light bulbs on glossy paint. This technique also allows a large number of candidates to be evaluated quickly, and can make full use of a cluster of workstations or a supercomputer. It doesn't require the constant supervision and resetting by hand that experiments with real robots do.

It also has the potential to discover active vision algorithms. If an unfortunate alignment of objects at a particular location causes problems for a vision algorithm, the traditional computer vision response is to improve the algorithm. An active vision approach would be to simply move to another location. In general, active vision introduces the

notion that the motion of the robot can affect what the robot sees, and therefore should be considered part of the vision algorithm.

However, it was not at all clear how veridical each aspect of the simulation would need to be. There are a number of peculiarities of real world images and CCD cameras, and without some further research or intuition, it seemed well within the realm of possibility that this simulation could require ever more research, becoming worthy of a dissertation by itself.

To ease the computer graphics burden, a simulation could be created as above, with the computer graphics replaced by real images. We could collect images from positions on a tightly spaced grid, under a number of different conditions, and then "interpolate" between them. If we want an image from a certain location, the interpolation can be done, for example, by projecting the nearest image onto the scene, then projecting it back onto the simulated camera.

However, both these approaches have the problem that simulations necessarily embody assumptions. For example, we might add random, Gaussian noise to our dead reckoned position to account for error. In practice, such noise is actually much better behaved than real noise. For example, if each move has k percent error, then after n moves our error is only $k/\sqrt{n}$ percent, in marked contrast to a constant error which can accumulate very quickly. In addition, the characteristics of positioning error are different on carpet, linoleum and concrete. Studying these characteristics and creating a model for each floor type may prove accurate enough, but in general, it is not clear how accurate the model needs to be and whether we could achieve that in practice.

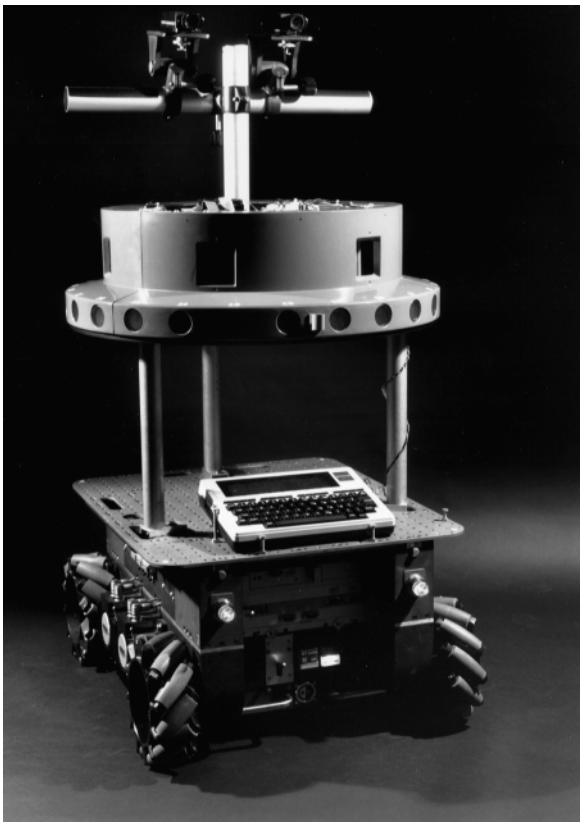
To summarize, simulations are relatively cheap and easy to automate, but the performance of the robot will only resemble that of a material robot in a real environment if one is very careful modeling the appropriate aspects of the environment. Knowing which aspects are appropriate can be difficult if not impossible. Even if the simulated robots perform well in the real world, there is no guarantee that modeling some additional aspect of the environment would not lead to even better performance. In fact, an adequate simulation may involve detailed physical calculations that take longer than real time.

Therefore, it would be beneficial to evaluate the algorithms by running them on the real robot. The robot could use the algorithm in question to navigate, stop when it hit something, and then travel back to where it started. When heading back, it could "cheat" by using a map of the space or additional sensors such as its sonar ring. Even so, this is very slow and requires a person to supervise the robot. As discussed in the Related Work chapter, other groups who have tried this use populations of less than 100 individuals and run for only a few dozen generations. At best it takes a good fraction of a minute for each evaluation, is prone to getting stuck, cannot be done while its batteries are charging, etc. As a rule of thumb, evaluations should take a second or less. This may actually be practical using a number of small, fast, reliable robots, but not with our beloved, lumbering Uranus.

This leaves us at a bit of a crossroads, since we have rejected both simulation and running in the real world. The way out used here is to not navigate during evaluation. Instead, before starting the evolution, we run the robot and record visual, dead reckoning and perhaps other data. During evaluation, we present this data to the algorithm and com-

pare its output to some externally given "correct" output, the *ground truth.* Then the best evolved individuals are tested on the real robot. This is not uncommon is supervised learning, and is the choice made for ALVINN and Baluja's work, as described in the Related Work chapter.

For a summary of these trade-offs, see Table 1.



What to Compute

This immediately presents a number of questions. What exactly should the inputs and the outputs of the evolved programs be? How should the ground truth be produced? How should the robot be controlled during data collection? And how do all these affect the evolution?

A brief description of the robot (at left) and its capabilities is relevant here. A more detailed description can be found at the beginning of the next chapter, "Data Collection."

The rectangular base, which measures 60 cm by 75 cm (2 feet by 2.5 feet), allows a full three degrees of motion. That is, the robot need not move in the direction it is facing (like a car), but can move sideways or any other direction, and rotate arbitrarily at the same time. It can also estimate the distance and direction it has moved and what angle it has rotated through by measuring how much each wheel has turned. In also sports 24 sonar sensors arranged in a circle of diameter 70 cm (2 feet 4 inches) and a single camera mounted at about the height shown, roughly human eye height.

The most basic choice for output would be the direction and speed of travel of the robot. The ground truth could be collected while a user drives the robot, or while driving under an existing, proven obstacle avoidance technique, most likely using sonar. A potential problem is errors in the ground truth, since navigation using a joystick or sonar can be awkward or occasionally make mistakes. Another problem is unrepresentative input. If the robot performs perfectly during data collection, the program will never see a bad situation such as a near collision, and very likely will not know how to handle it. If the robot is intentionally steered into bad situations, then during evolution it will learn to steer into bad situations.

Those problems are not insurmountable. ALVINN collected data while a human operator drove it, and generated images of bad situations by shifting existing images left and right. Something similar might work for office environments. Alternatively, we could collect a representative set of images, for some notion of "representative," and hand label the direction of travel and speed for each image.

A more troubling problem is that the output may be so distantly related to the input, that the mapping is impossible to learn with current techniques and resources. The input, after all, is simply an array of numbers representing the amount of light in various directions. Starting from a complete blank slate, by combining program elements in a randomized directed search, how long will it take to find even the simplest algorithm that does better than moving at random?

Table 1: Pros and Cons of Potential Evaluation Methods

	Pros	Cons
Total Simulation (with later validation in a real environment)	• Control of environment • Can simulate dead reckoning error, specular reflection of lights, sensor and actuator error • Can speed it up by using more computers • No human intervention needed during learning (can run overnight) • Could learn active vision	• Necessarily embodies assumptions • Hard or impossible to model all significant problems of imaging. What works in simulation might not work in the real world and vice versa. • Time consuming to create many environments • May be too time consuming
Simulation w/ Real Images (w/ later validation)	• All advantages of "Total Simulation" • Most peculiarities of imaging are represented	• Necessarily embodies assumptions • Collecting dense set of calibrated images is difficult and time consuming
Evaluation on Robot	• No need to model robot or environment • Can take all nuances of robot and environment into account • Could learn active vision	• Slow evaluations mean smaller population
Offline Evaluation with Recorded Images (w/ later validation)	• Takes nuances of imaging into account • Can speed it up using more computers • No human intervention needed • Multiple environments is easy	• Cannot learn active vision • Ground truth might embody assumptions

Therefore, in order to simplify the problem, the output of the learned algorithm is the distance to objects in various areas of the image. Originally I proposed to estimate the image depth at 1200 points in the image, arranged in a 40 by 30 grid. At each point the individual was to produce three estimates, each with a confidence. For ground truth, rather than create a 3D model of the environment and reliably locate the robot in it (a rather substantial task), I proposed to predict the next image, given the current image, the depth map and dead reckoning. As a second form of feedback, I planned to predict the sonar readings as well.

To predict the depth at a given pixel, I was planning to look at a 24x24 neighbourhood only. In an area with little visual texture (e.g. a blank wall) this is often insufficient. To counter this, the depth map at the previous time step was to be fed back as input, allowing depth information to spread over larger distances through local interactions over time. However, as I implemented that, certain problems became apparent.

When the next image is predicted from the current image and the depth map, it must be compared to the image the camera currently sees. A straightforward way would be to simply subtract intensity values pixel per pixel, but that measures the wrong thing. What's important for depth perception is to compare the predicted and actual locations of each object. This could be done in the image, with displacement measured in number of pixels, or converted to 3D estimates and measured in meters. It's not clear which is the better metric, however they both suffer from the same problem. They're both essentially the same as optical flow, and optical flow is known to have problems with rotation. When there is rotation between the two images, rotational errors completely

swamp translational errors. This is true even with good dead reckoning, better than what I have available. That's why many optical flow systems use a mechanical device to rotational stabilize the camera.

Without stabilization, a better approach is stereo vision. Stereo vision can be used to create the ground truth in the form of a depth map. This depth map could even be corrected by hand. However, even stereo has a problem which ultimately affects optical flow as well: areas of low visual texture. In these areas noise dominates and both stereo and optical flow give no feedback (at best) or incorrect feedback (at worst.) In experiments with typical data, the vast majority of matches were of low or very low confidence.

This is a general problem with using a dense depth map. No existing technique can provide accurate ground truth automatically. The alternatives, such as hand correcting those depth maps, hand constructing a model, and generating ground truth using a different sensor, are all time consuming and perhaps miss the point. A dense 3D depth map is most likely overkill for navigation. Robots navigate very well from sonar, and some of the visual navigation systems described in the Related Work chapter work quite well while producing only a very low resolution depth map. If we force the learning to produce such a dense map, it will never find these "low resolution" approaches.

Part of the original motivation for the dense depth map was the desire for a lot of feedback from each image. The discussion in this section suggests that too much feedback can be a bad thing, lowering the quality of the ground truth as well as forcing the EC to learn the wrong thing.

So what should the output be? The estimate that is most useful for obstacle avoidance is the nearest object in any given direction. Sonar gives six readings in the 90 degree field of view of our camera, so the evolved programs were evaluated by having them predict six depth estimates per image.

Early experiments attempted to predict the values returned by sonar, but had very limited success. Sonar data is rather noisy, and the outgoing cone covers 30 degrees, so the returned distance depends on a large part of the image. Also, things that are easy for the camera to see can be difficult for sonar and vice versa.

Instead, the representation chosen was that used by Ian Horswill's Polly the robot, as described in the Related Work chapter. Here, a direction corresponds to a vertically oriented plane. We represent distance, not in meters, but by something more directly related to the input, the location in the image of the lowest non-ground pixel. If the ground is flat and objects are roughly vertical and touch the ground, as is the case in this dissertation, there is a one to one monotonic mapping between image height and distance of the object.

To keep things simple in this first step, the programs will not maintain state from one image to the next, that is, they will be completely reactive. In fact, the program will be executed once for each of the six columns in the image, and no state will be maintained between executions.

To summarize the evaluation method, the robot is first run in an office environment while using sonar to avoid obstacles. During this *data collection* run, it records camera images, sonar and dead reckoning data. *Offline learning* is where the automated iterative design takes place,

with the computer learning to predict object distance from the camera images. Finally, during *online obstacle avoidance*, the robot uses the learned algorithm to predict the distance to objects and avoid them.

The input to the evolved programs is an image, and the column of the desired estimate. The output of the program is a single real number, interpreted as the vertical location of the lowest non-ground pixel.

With the evaluation method in hand, we now turn to the representation.

Representation

It is seductive to treat evolutionary computation as a black box that automatically finds subtle and intricate uses of the building blocks provided. After all, that is how people look at natural evolution: starting with DNA and amino acids (if not raw atoms), it produced life, including human intelligence. But after working with EC for a while, one realizes that it simply is not that powerful. If the representation has a certain flavour, such as data driven or conceptually driven, it will be difficult if not impossible for EC to develop strategies in a different flavour. As the machine learning community has discovered, representation is king.

What the representation does is make certain algorithms easier to find and others harder. A representation which provides only data-driven atoms for analyzing an image makes data-driven algorithms easy to express, which means they will have greater density in the search space. This means they will be more frequent in the initial random population and more likely to be found later. In contrast, it may require great tricks to implement a conceptually-driven algorithm. It may be stuck with a weak contextual influence, such as choosing which atom to use in a given situation, or may be able to extract some context by applying the atoms at many different locations in the image. In practice, such clever algorithms will be extremely difficult to find, if not impossible.

We also have limited computing power, both because the evolved algorithm must run in real time on the robot, and to make individual learning runs take days, not weeks (or months). Therefore, we cannot just provide some atoms that return the intensity value at any location, some looping atoms and some arithmetic atoms. Most algorithms it would develop would take far too long to evaluate. Instead, the representation must not allow programs that could take inordinately long.

The approach taken here is to borrow our representation and atoms from the existing body of computer vision work, which has several advantages. First, it allows a more direct comparison to traditional methods. Second, it leverages the large amount of work that has already been performed in the field. Any existing techniques which capture some structure of the problem domain will likely be exploited by the EC. And third, it is likely to succeed at least as well as the traditional methods it generalizes from. So while it is a tenet of this work that commonalities in existing bodies of work can reflect the peculiarities of the minds that produce them, the reality is that we must start somewhere. As argued in the Philosophy & Manifesto chapter, the research program proposed here has the right properties to some day generalize beyond the limits imposed in this first step.

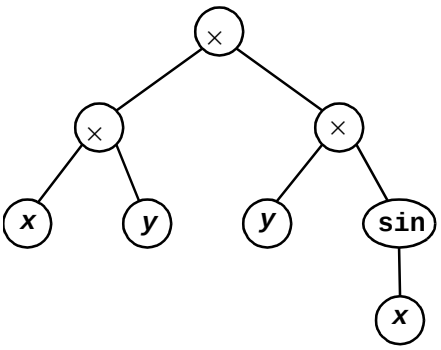


Figure 1: In genetic programming, genes are programs, represented as parse trees.

Genetic Programming

Traditional computer vision uses procedural programming languages such as C, as does most robotics. In contrast, such representations are rare in evolutionary computation and evolutionary robotics, but the representation that comes closest is genetic programming or GP [Koza 1992, *Genetic Programming*]. Genetic programming is a type of genetic algorithm [Holland 1975, *Adaptation in Natural and Artificial Systems*] that individuals are programs, and are represented as a parse tree (see Figure 1). The set of possible nodes is created by a programmer and depends on the problem domain. In GP parlance, nodes with no children (leaf nodes) are dubbed *terminals*, those with children (internal nodes) are called *functions.*

Trees are created at random by randomly choosing a node for the root, then choosing random nodes for each of its arguments (children), and similarly for their arguments, etc. A fixed number of trees (say 1000) are randomly created this way, and each one *evaluated*, that is, assigned a real number that measures how well it solves a given problem. The assigned number is called the *fitness.* Next, individuals are randomly selected, with fitter individuals more likely to be chosen. The selected individuals are either copied verbatim into the new population (*replication*), or undergo *crossover.* Crossover works by selecting a random node in each parent and swapping the subtrees rooted at those nodes (see Figure 2). Once 1000 individuals are created this way, their fitness is measured and the processes repeats for a fixed number of generations. Typically, the individual from the last generation with the best fitness is designated as the result of the GP run. Creation and crossover are constrained so that resulting trees are viable programs.

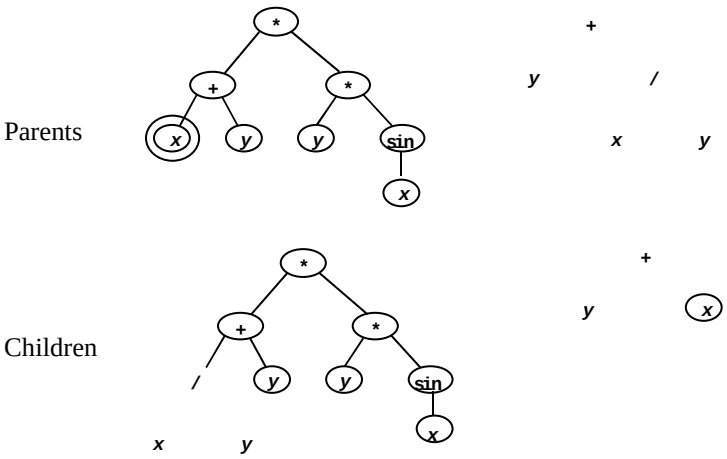


Figure 2: Crossover

Genetic programming (and evolutionary computation in general) have been shown to be efficient enough for practical use [Kinnear 1994a, *A Perspective on the Work in this Book*]. One can conceptualize the solution of a problem as a search through the space of all potential solutions to that problem. Taking this view, genetic programming is considerably more powerful than simple alternatives such as exhaustive or random search. It implicitly utilizes a directed search, allowing it to search the space of possible computer structures and find solutions in hours on tasks that would take random search considerably longer than the life of the universe. While computationally still challenging, genetic programming can produce useful and dramatic results in hours and days instead of millennia.

There has been a lot of work in evolutionary computation, especially in the last decade, but rather than carefully choosing the best settings, I have instead opted for a very simple form of GP, that from Koza's 1994 book *Genetic Programming II.* This is to make a point. The fact that such a naïve form of GP succeeds means the result does

not depend on getting the parameters just right. In other words, the result is *robust* to changes in detailed setup of the genetic programming.

Representation For Vision

What set of nodes should we use? To reiterate the point about representations, any representation makes certain programs easier to find and others harder. Among traditional programming languages, functional, procedural and object oriented styles lead to different solutions to a given problem. What's more, learning methods must be “helped along” by providing a representation close to the problem domain. How, then, can we provide a representation without forcing an architecture? Once again the crucial difference is between hard and soft. We choose building blocks which each do a significant amount of work, but can be put together in a large number of ways, some quite novel. For a non-computational example, LEGO building blocks simplify the process of mechanical design and construction, yet can be put together in many interesting and creative ways. While some high level architectures are easier to construct than others, the architectures that arise in practice cover a limited but interesting space.

Here is a summary of existing, successful visual obstacle avoidance systems from the Related Work chapter. They are summarized with an emphasis on their architecture.

Larry Matthies and company at JPL

This group uses stereo vision, post processed using consistency checks and filters. Stereo vision compares a window in one image against a series of same-sized windows in the other image. The series of windows are typically evenly spaced along a horizontal line.

Ian Horswill and Polly the Robot

Polly’s vision algorithm assumes the ground is untextured, and starting at the bottom of an image, moves a rectangle vertically, stopping at the first location that contains significant texture. This window is assumed to contain an obstacle, and the further up the image this obstacle is found, the further away it is assumed to be.

Liana Lorigo at MIT

Following on from Ian Horswill’s work, a histogram is computed over a series of windows. The windows are evenly spaced along a vertical line. The bottom most window is assumed to represent the ground, and the first window above that with a significantly different histogram is said to contain an obstacle.

Rattler at CMU

The Rattler trials used stereo vision, for our purposes here similar to the JPL work.

David Coombs and company at NIST

This group uses optical flow. Each rectangle in one image is compared with a collection of rectangles in previous images to find the most similar. The robot steers away from the side with the most optical flow.

Illah Nourbakhsh at CMU

Three relatively narrow depth of field cameras are mounted on a robot and focused at three different distances. The three images are divided

into a number of rectangles, and in each rectangle the sharpness of the image is computed.

The common element chosen from these algorithms is the “iterated window.” All the existing algorithms combine local analyses over a rectangular window to produce a handful of numbers. The process is repeated as the window is moved horizontally or vertically.

Therefore, one function node will be a loop, in the computer science sense, that moves a rectangle over an image. This node will have arguments that determine the size of the window, the x and y locations in the image of its initial location, the direction of travel and how far it will travel. The final argument will be a function (i.e. a subtree) that is executed once per iteration, that is, once for every location of the window.

To keep the runtimes practical, there can only be a handful iteration branches executed per frame. This structure drastically reduces computer time compared to arbitrary computation over an image.

Thus, the novel aspects of this dissertation in robotics are the use of a scripting language as a representation for learning, the abstraction of the iterated window, the construction of a practical example, the exploration of some practical issues. The contributions are expounded further in the chapter “Discussion.”

With the major concepts behind the representation and evaluation in hand, we now turn to the details of the experiments.

Part II

Experiments

4 Data Collection

This dissertation uses genetic programming in a standard supervised learning framework. That is, the robot is first run in an office environment while using sonar to avoid obstacles. During this *data collection* run, it records camera images, sonar and dead reckoning data. Then, during *offline learning*, genetic programming iteratively creates and evaluates programs to predict object distance from the camera images. Finally, during *online obstacle avoidance*, the robot uses the learned algorithm to predict the distance to objects and avoid them. This chapter describes the collection of the training data.

The learning framework in this thesis evolves algorithms that determine distances from camera images. To collect a video stream that is representative of what the cameras might see during visual obstacle avoidance, the robot collects data while avoiding obstacles under sonar.

While obstacle avoidance under sonar is easier than under vision, it still took many attempts to get a working system. The method that proved most successful determines speed based on proximity to the nearest object, and determined direction of travel by fitting lines to points on the left and right sides of the robot.

It should be kept in mind that throughout the thesis, all robot motion had some forward component. This is not uncommon in reactive systems, especially those based on vision, since they can not see to either side or behind. The only exception was when the robot was halted by an object directly in front of it, in which case it would turn in place.

Record Video Robot, Real Time

Learn Offline Genetic Algorithm

Build Navigation By Hand

Validate Online Robot, Real Time

The Uranus Robot and Computing Hardware

All experiments were performed on the Uranus mobile robot, pictured in Figure 1 [Blackwell 1991, *The Uranus mobile robot*]. Uranus sported a three degree of freedom base, a ring of twenty four sonar sensors, and one to three black and white video cameras. Controlled by an off-board desktop computer, it existed in more or less its current form when this dissertation was started. It was not modified mechanically except for a few repairs, and the only modification to the electronics was to bypass the onboard 68000 computer, so that the desktop computer communicated directly with the motion control board.

The base uses an innovative wheel design that allowed a full three degrees of freedom. That is, the robot could move forwards or backwards, move sideways, or turn in place. In fact, it could perform any combination of these simultaneously, for example moving along a straight line while rotating about its center.

The sonar ring, originally bought from Denning mobile robotics and refurbished in early 1999, contained twenty four Polaroid sonar transducers equally spaced along a circle of diameter 71 cm. Each sensor points directly away from the center of the ring. Custom electronics allowed a range of 9 to 774 cm (3.6 inches to 2.54 feet.) Sonar sensors have known problems with smooth surfaces that reflect the sound like a mirror, and with crosstalk between different transducers.

Table 1: Video settings during data collection

Focused Representation Runs	Expanded Representation Runs
Resolution: 320×240	Resolution: 320×240
Colour space: 8 bits/pixel grayscale	Colour space: 8 bits/pixel grayscale
Y/C separation: skipped	Y/C separation: skipped
Luminance Coring: 32 or less	Luminance Coring: 16 or less
Chrominance Comb: Enabled	Chrominance Comb: Enabled
Gamma: Enabled	Gamma: Disabled
Error Diffusion: Enabled	Error Diffusion: Enabled
Video Format: NTSC-M	Video Format: NTSC-M
Hardware Type: Bt878	Hardware Type: Bt878
Camera gain: automatic	Camera gain: automatic
Coax Cable Length: 19.5+32 = 51.5 feet	Coax Cable Length: 19.5+5.5 = 25 feet

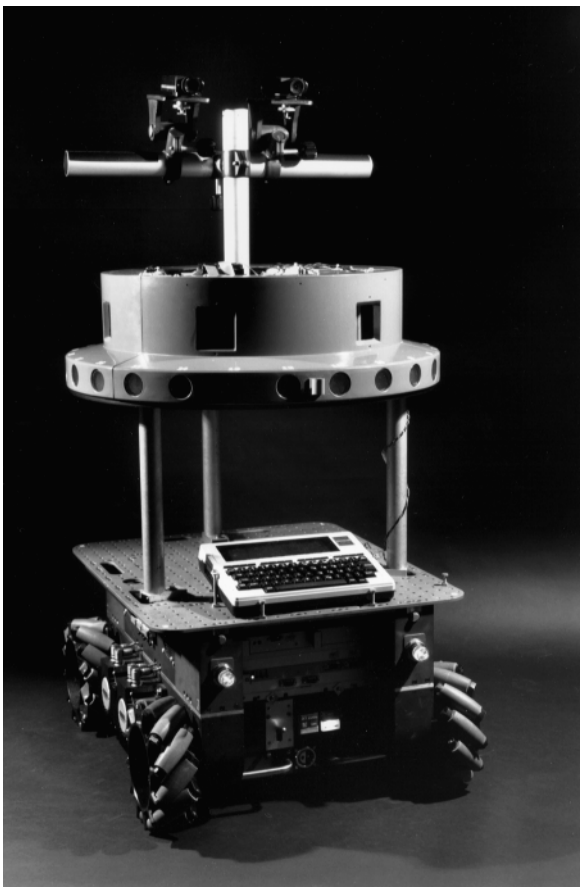


Figure 1: The Uranus mobile robot.

The camera setup consisted of one to three Sony XC-75 black and white analog video cameras in a custom mount. The mount can be tilted or panned by hand before an experiment, but is fixed during a run. The cameras used 3.6 mm lenses that provided an almost 90 degree field of view horizontally and 65 degrees vertically. A typical image is shown in Figure 2. The video digitizer used here is the Hauppauge WinTV 401, based on the BrookTree Bt878 chipset, a later version of the Bt848 chipset. At every pixel it returns a number between 0 (black) and 255 (white). The video settings used throughout the thesis are shown in Table 1.

Luma coring takes all pixels below a threshold and converts them to 16. This reduces the level of noise perceived in dark areas of the image, where it is most notable. For the Bt848 digitizers under BeOS, there are two settings, either 32 and under or 16 and under. Chrominance Comb refers to averaging colour information over two or more lines and should make no difference here, color information is ignored. The digitizer was run in black and white mode, which skipped the Y/C separation with its low pass filtering of the luminance signal. Gamma correction only happened in RGB signals, not in the grayscale mode used here, so the setting should not have had any effect. Similarly, error diffusion only affects RGB15 and 16 modes, and should not have affected this dissertation. The luminance coring had a visible effect on both the image and the error rate of evolved individuals, whereas the chrominance comb, gamma and diffusion controls had no visible effect.

Uranus was run from a desktop dual processor Intel computer running the BeOS operating system. Analog video from the cameras was connected to the off-board computer using two RG-59/U cables. In the expanded representation runs they were 19.5 and 5.5 feet in length for a total of 25 feet. For the focused representation runs, so that the robot could travel farther, the shorter cable was replaced with one 32 feet in length, for a total of 51.5 feet. The signal was properly terminated with a 75-ohm resister just before the digitizers. The offboard computer communicated with the

Figure 2: A typical camera image. Before learning, the fisheye distortion seen here was removed.

sonar ring using a 43 foot RS232 cable at 19,200 baud. Finally, the computer communicated with the four Precision MicroControl Corporation DCX-VM100 motion controllers using another 43 foot RS232 cable at 9600 baud.

There were two offboard computers used at different times in this work. They were used for online activities such as navigating with sonar, digitizing video, and online validation. They were also used for all of the offline learning. The first image set was recorded on the first machine, a dual 333 MHz Pentium II machine with 128 MB running BeOS R4.x. This machine was also used for initial development of the genetic programming system, video digitizing, and robot control. The offline learning of the genetic programming system was run on both that machine and the newer 700 MHz Pentium III machine with 256 MB. The older machine was upgraded to 256 MB and both ran BeOS R5.0.2.

Initial development used the Metrowerks compiler for BeOS, but with the release of R4 Be required the use of the gcc compiler. All code used throughout the thesis was written in C++ under BeOS, except for the simplification of individuals which was written in Lisp and ran under Solaris. There was no source level debugger available until late in the thesis, so most debugging was performed using print statements. The use of assertions, and good software engineering principles, helped significantly here. The code was written to use ISO standard C++ wherever possible in order to be portable, which paid off when it was ported to a Cray supercomputer. (Unfortunately, while the code worked on a single CPU, the multi CPU version was not completed in time, so the Cray port was never used.) The BeOS-specific aspects were mainly the user interface, the digitizing and the serial port interface. The (primitive) user interface used Marco Nelissen's liblayout library [Nelissen 2000, *liblayout*], which allowed easy user interface construction in C++ with a minimum of work.

BeOS was selected, despite its lack of existing applications, because its clean, modern, object oriented API (Application Programmer Interface) promised to be easier to program, and because the needed digitizing functionality already existed. In practice, the digitizing functionality was non standard at the time of the thesis proposal, in BeOS R3. It was completely rewritten in R4, but was so low level that it required a lot of work to get even simple digitizing working. This and other programming was exacerbated by the lack of a source level debugger. By the time the high level API came out with R4.5, many months had been spent getting the low level version working, so the new APIs were not used.

In retrospect, as much time was wasted due to these problems as was saved by using the modern API. This experience, together with the problems the lab experienced with FireWire cameras on the Macintosh, have demonstrated how hard it can be to evaluate a new technology without working with it for months. Even after a month or two of work, it is often unclear whether a working version is just around the corner, or another few months off. At that point, it is often the case that learning another system would take longer, or at least have more uncertainty as to how long it would take and how well it could serve ones needs that switching is most likely a worse move.

Therefore, a methodological lesson learned is that it is impossible to predict the problems that will happen with unfamiliar setups. Therefore, one should never be the first to adopt a new configuration of tech-

nologies. Instead, one should talk to people who have already used it for very similar or exactly the same purposes. Because of this, it is best to stick to common, well understood hardware. Even such simple changes as using multiple cameras in place of a single one can cause problems, if the cameras are new technology. Even if all the components work in isolation, that is no guarantee they will work together. Modern computers are complex enough that seemingly small changes to a setup can cause problems that take months to track down and solve.

The Sonar Baseline

As described in the Related Work chapter, robot obstacle avoidance using vision has traditionally performed worse than obstacle avoidance from sonar. As such, it was important to implement obstacle avoidance from sonar, to compare to the learned vision algorithms. But there was another reason to implement it.

With any learning technique, it's important for the training data to be as representative as possible. For example, if the training data for this dissertation had been collected while moving straight down the middle of a corridor, then the center of the image would almost always be far away. Simply assuming that the forward direction is always far away will get you pretty far. However, when the robot is navigating on its own, as soon as it turns away from straight forward, this assumption will be wrong. It will be in a circumstance it has never encountered during training, and will most likely falter.

Therefore, it would have been best to collect data while the robot was moving and avoiding obstacles using vision. Of course, this is a catch-22: it needed to collect data before it can learn to avoid obstacles, but it needed to avoid obstacles in order to collect the data. Instead, the next best thing was chosen: to save images to disk while avoiding obstacles using sonar.

Images were stored along with the most recent readings from the sonar sensors, and the current global position according to the dead reckoning, although only the images were used. To effect this, the data collection program used two main threads.

The navigation thread read sonar sensors and moved the robot, independent of any data collection. The sonars each made a sound, then listened for its reflection. Crosstalk between sonars, where the sound of one sonar is heard by another, turned out to be a problem in the area just outside the doorway to the lab, where the robot was close to good specular reflectors such as metal filing cabinets and painted drywall. So the program only operated a single sonar sensor at a time, stepping through each of them in turn. Because the robot never travelled backwards, it only needed to see forward and sideways. Therefore, only fourteen of the twenty four sonars were ever used, the sixteen that pointed most forward.

Once all the data is in, the algorithm described below is run, and the new desired speed and heading are turned into motor commands and sent to the motor controller. When navigating this way the system is completely reactive, ignoring all previous input as well as the vision and dead reckoning data.

The recording thread wrote all recorded data to the disk at specified intervals. It recorded the cached position (with timestamp), paused until a sufficient number of video frames had passed, recorded the cached

sonar data, and then repeated. Therefore, there was one position estimate, one image and one set of sonar readings per record. The position and video each had their own timestamps, which in general were slightly different. Because it took up to a second for a full reading from the ring, a precise timestamp was meaningless. Therefore, each record simply contained the last set of sonar readings returned by the ring. As mentioned above, the position and sonar data were unused.

Other threads handled communication with the motion controller and the video driver. In addition, a simple GUI showed the current image to the user and allowed him or her to start and stop recording, to stop the robot, and to move the robot half a meter straight ahead for debugging.

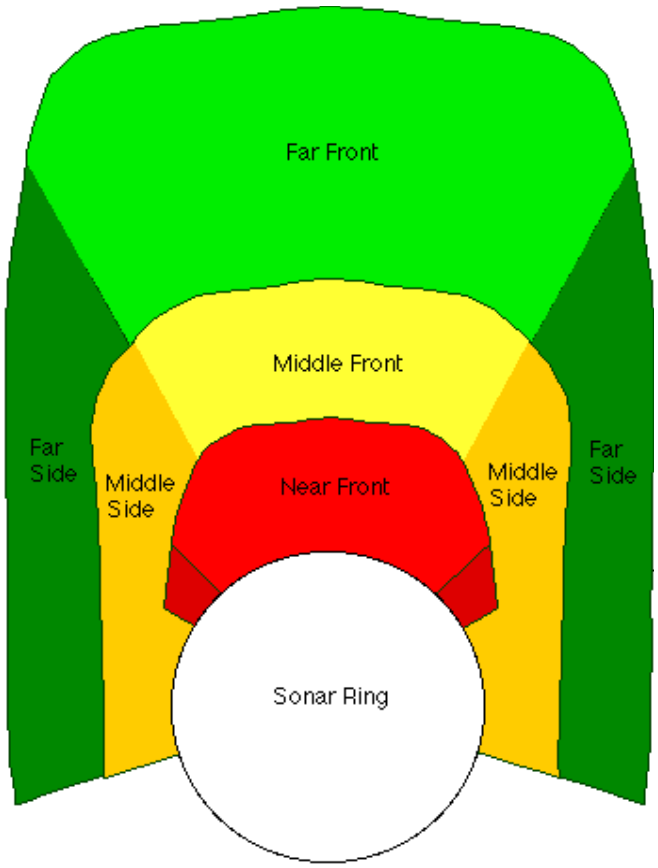


Figure 3: The area of attention for sonar navigation (draw to scale)

The Selected Sonar Algorithm

The navigation algorithm went through several modifications until it worked well in the desired environment. The most successful algorithm was then used during data collection and is described here. Other attempts, their failure modes and the lessons learned are discussed below.

The method that proved most successful looked only at objects in front of or beside the robot, and close to it (see Figure 3.) As described above, to reduce cross talk between transducers, the only sensors fired were those whose output were needed. The union of all the colored areas was termed the *area of attention*. Any objects outside of it were completely ignored.

The selected algorithm starts by dividing the environment into six regions, “front” and “sides” of “near,” “middle” and “far.” For this “cased based” approach I am indebted to Illah Nourbakhsh [Nourbakhsh 2000, *Property Mapping*]. As an overview, the idea behind each region is:

- Far: everything's ok, move at full speed, but move away from anything here so it doesn't enter the “medium” region.
- Medium: move at 2/3 speed, carefully plod your way through the obstacles.
- Near: panic halt.

A detailed description follows.

If the robot receives any readings in the “near” area, it immediately halts. It then waits until there have been no near readings for half a second before moving again. If the robot is panic halted for more than a second, it picks a direction (left or right), then keeps turning in place until the halt condition clears.

If the robot isn't panic halted, it looks for readings in “middle sides.” These are quite close to the robot and appear when the robot goes through a doorway or is in some danger of colliding with an object. If it finds any, then for each side, it chooses the two readings closest to its center line and fits a line to them (see Figure 4). These are the points that, if the robot were to drive straight ahead, it would come closest to. If both lines are diverging from the robot, it assumes the gap it is in is widening and continues moving straight ahead, without turning. If one line is diverging and the other is converging, the robot faces parallel to the converging line. Finally, if both lines are converging, the robot aims for the intersection point. One or no readings on a given side count as a diverging line.

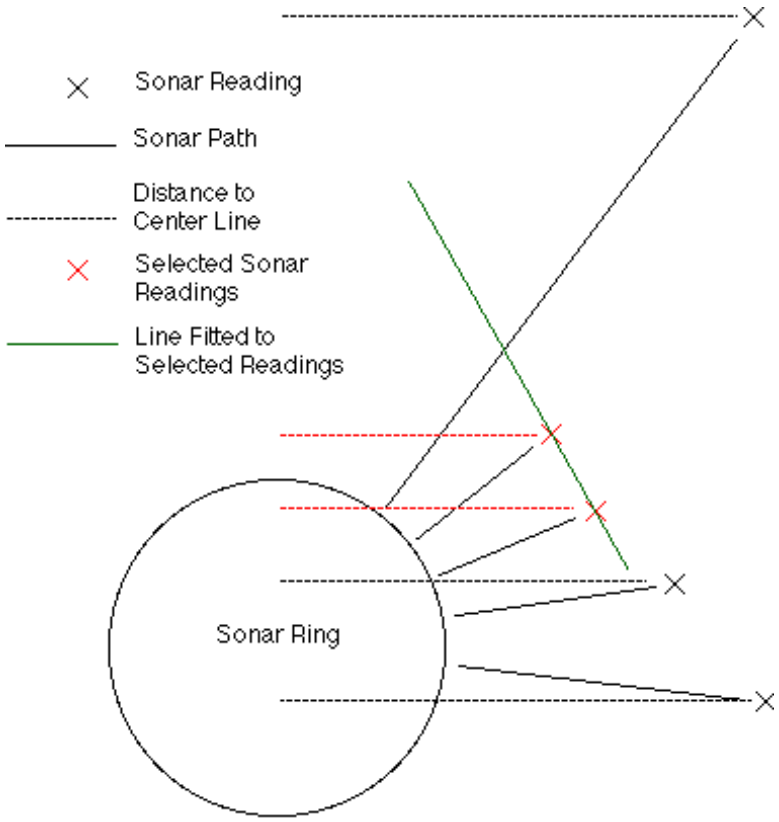


Figure 4: For readings close to the robots sides, a line was fit to the two readings closest to the robot’s center line, to determine the general trend of objects on that side of the robot.

The Uranus robot has a three degree of freedom base. It doesn't have to move in the direction it's facing, but can move sideways or at an angle. In fact, the direction of motion can be chosen independently of the direction the robot faces. In every other case the robot always moves in the direction it faces, but when there are readings in “middle sides” is uses a potential field [Craig 1989, *Introduction to Robotics*] to help guide it. This is a great help when navigating through doorways.

If the middle sides are clear but there's an object either far or middle front, the robot classifies all readings within the area of attention as “obstacles,” and those outside as “non-obstacles.” It faces the largest gap, i.e. the largest number of contiguous non-obstacles. Finally, if the only sonar returns are from the “far sides” area, it fits lines to both sides, finds the closest line, and faces parallel to that.

The “straight line model” of the environment works well when navigating down hallways, quickly aligning itself with the walls, yet moving away from them if it becomes too close (i.e. in the middle sides region.) It also turns corners well, the biggest gap usually corresponding to the direction that the hallway bends. Occasionally it turns the wrong way for a second or so before “changing its mind,” but never gets stuck in a corner.

This straight line model also works well in our cluttered lab or when navigating through doorways, even the doorway out of the lab with a cluttered table on one side. Because the line is fit to only two readings, it seems to work more like a derivative or a local tendency in the area of most danger. The robot can usually navigate out of the lab and into the hallway beyond. It should be kept in mind that this is particularly difficult for our robot, since it's only a little narrower than the door, and when turned at even a slight angle it's rectangular base will bump the door frame.

Other Algorithms

This section summarizes the earlier obstacle-avoidance-from-sonar algorithms which lead to the above. Since obstacle avoidance from sonar has been done before, I do not consider these experiments to be a contribution. As such, I have only recorded the algorithm, observed failure mode and lesson learned for each, and not the details of each experiment.

All these experiments used the same fourteen sonars as the selected algorithm. The humble beginnings of the sonar baseline were to face toward the furthest reading. To add robustness to noise, each reading was replaced by the minimum if it and its two neighbors. In other words, it found the furthest "opening" of three adjacent readings and faced toward it. It had the property that a close reading eliminated not only its own direction, but the direction on either side.

However, a few adjacent specular readings would confuse it. For example, in the hallway of Figure 2, sonars 21 and 22 would give specular readings, causing the robot to turn left when already too close to the wall. This same problem occurred when averaging over four or two readings.

The problem was twofold. First, the largest source of errors (specular reflections) overestimates readings, making long readings less reliable than close readings. Second, it is more useful to avoid close objects than to seek far ones.

The next attempt takes the three smallest readings on each side and adds their reciprocals. This gives a "closeness" number for each side. The turning rate is then proportional to the relative difference:

$$right = \frac{1}{\text{distance}_{r1}} + \frac{1}{\text{distance}_{r2}} + \frac{1}{\text{distance}_{r3}}$$

$$left = \frac{1}{\text{distance}_{l1}} + \frac{1}{\text{distance}_{l2}} + \frac{1}{\text{distance}_{l3}}$$

$$\frac{d\theta}{dt} \propto \frac{right - left}{right + left}$$

This caused too much turning at small angles and too little at large angles, so it was made proportional to the square of the relative difference, but with the same sign:

$$\frac{d\theta}{dt} \propto \text{sgn}(right - left)\left(\frac{right - left}{right + left}\right)^2$$

For reasons discussed below, I next switched to the Property Mapping approach of [Nourbakhsh 2000, *Property Mapping*]. In this approach an algorithm first classifies its sensor readings into a small number of discrete cases, such as *object-on-left* or *goal-far-right*. It then decides which action to take case by case. In the paper, Nourbakhsh argues persuasively for property mapping to make debugging easier for human designers by making the robot's inner state more transparent. There are two other reasons it was helpful in the thesis work.

First, changes are local. Potential fields are canonically described of as a single expression. When implemented this way, after a little tweaking and debugging, the robot succeeds in many situations but fails in a few others. Most changes to the formula affect all cases, often breaking ones that previously worked. This was particularly apparent in my work on the video game *Star Trek: Armada* [Activision 2000], which involved authoring coordinated obstacle avoidance for many simulated spaceships with different speeds and maneuverability.

Second, as a special case of the first, simple code can be created to slow down then panic halt the robot when in immediate danger of collision. This allows other parts of the algorithm to be less conservative, knowing that if they occasionally fail the robot will still probably be safe.

The slowing down and halting does not work perfectly, because sometimes a nearby object is specular or not at sonar height. In the target environment, however, it works rather well. When an object is off to the side, the robot can not panic halt until it is near the minimum detectable distance, or else it will freeze when going through doorways.

As was discovered, there is a common problem in robotic control. If the obvious approach is taken, namely turning away from the nearest object, our action depending only on our distance to it, the robot will tend to weave back and forth as it travels down a hallway. That's because when the robot becomes close enough to one wall it will start turning, but won't diverge from it immediately. Instead, it will get closer to the wall until it's facing parallel to it. However, it's still closer to the wall than its threshold, so it will keep turning. By symmetry, it will usually turn until its facing away from the wall at the same angle it was facing toward it originally. However, it's now facing the opposite wall at that same angle. When it approaches the opposite wall it'll turn until it's diverging from it, and so on, back and forth.

Turning away from the wall only when facing it, and heading straight otherwise, helped somewhat. However, given sensor noise and system latency, the robot would sometimes turn away from it when in fact it should not. It would therefore end up facing significantly away from the wall and again ping pong between the walls, albeit much more slowly.

To avoid the ping-ponging, when the only objects are in "far-sides," the robot was servoed parallel to the wall. By defining "wall" as in the selected algorithm, the obstacle avoidance is fairly conservative and works in other environments as well.

With corridor navigation working well, the next step was readings in "middle sides." The most difficult example is going through a doorway, especially since our robot is only a little narrower than the door. A number of attempts worked at centering the robot between the closest reading on each side, but these were foiled by the particular geometry

around the lab door. The closest reading on the right is a table, not the doorway, so the robot would align itself between the table and the open door—and hit the right edge of the doorway.

The next attempts focused on the two readings, one from the left and one from the right, that define the opening the robot must pass through. To select these points, two different criteria were tried, first the readings closest to the center line, then the pair with the smallest separation. The robot then aimed for the middle of this opening by facing the half way point between the two, using various control laws. When the opening was far ahead this worked well, but when it was in the opening the approach leads to oscillation of various sorts.

Knowing only those two points isn't enough information to know how to proceed because the line joining them may not be parallel to the doorway. Our goal is to head through the doorway, yet the robot doesn't know which way the doorway is aligned. These considerations lead to the final algorithm, which fits lines to both sides to decide which way to face, and uses a potential field to determine the direction of movement.

Discussion

During the development of this algorithm, several unsuccessful algorithms were created before the final, successful one. The advice I would give others attempting to write obstacle avoidance algorithms by hand is:

- Classify your situation into a small number of discrete cases, then handle those independently. This allows the designer to tweak and debug one situation without affecting other, already working situations.
- Decouple translation and rotation. Set translation speed inversely as distance to object.
- When considering nearby readings, avoid those closest to the planned path, not the current position.
- As in traditional linear control, algorithms based only on the distance to an object tend to oscillate.
- Distant sonar readings are less reliable than nearby ones.
- Base an obstacle avoidance algorithm on nearby readings, rather than distant ones. Avoid nearby objects rather than seek far ones.
- The orientation of an opening is important, as well as its center.

The succession of designs presented here is also an example of iterative design, and it should be recalled from the Philosophy & Manifesto chapter that the framework proposed here can be viewed as automating this process. Such an application is a great topic for future work.

Notes on Collected Data

Below are the details of each data set. MPEG movies of each are available from the author. The reduced representation runs used the first two, the expanded runs used the last one.

NSH Hallway

Camera Angle:	51 degrees from horizontal
Total Number of Frames:	328
Frames In Training Set:	65 (every fifth)
Number of Fitness Cases:	65 × 6 = 390
Elapsed Time:	75 seconds
Frame Rate Of Training Set:	65 ÷ 75 = 0.87 fps

While the robot had to travel mostly straight down a hallway, it started out a little askew, so it approached one side. At one point, a person walks past the robot and is clearly visible for many frames. At the end of the hallway it turns right. The carpet is grey with a large black stripe.at one point. The shadow of the robot is visible at the bottom of most frames.

FRC Hallway

Camera Angle:	51 degrees from horizontal
Total Number of Frames:	356
Frames In Training Set:	71 (every fifth)
Number of Fitness Cases:	71 × 6 = 426
Elapsed Time:	82 seconds
Frame Rate Of Training Set:	71 ÷ 82 = 0.87 fps

Starts with robot in lab doorway. Moves straight until its in the hallway, then turns right, travels down hallway, at end turns right, then travels straight to dead end. All doors were closed. The fluorescent light bulbs at the start are burned out, so the intensity of the carpet varies widely. The shadow of the robot is visible at the bottom of most frames.

Combined

Camera Angle:	51 degrees from horizontal
Total Number of Frames:	328 + 356 = 684
Frames In Training Set:	68 (every tenth)
Number of Fitness Cases:	68 × 6 = 408
Elapsed Time:	75 + 82 = 157 seconds
Frame Rate Of Training Set:	68 ÷ 157 = 0.43 fps

This data set was simply the combination of the above two data sets, using every tenth frame instead of every fifth in order to keep the training set size approximately equal.

FRC Hallway - For Expanded Representation Runs

Camera Angle:	31 degrees from horizontal
Total Number of Frames:	226
Frames In Training Set:	75 (every third)
Number of Fitness Cases:	75 × 6 = 450
Elapsed Time:	52 seconds
Frame Rate Of Training Set:	75 ÷ 52 = 1.44 fps

Starts with robot in lab doorway. Moves straight until its in the hallway, then turns right, travels down hallway, at end turns right, then travels straight. All doors closed, except near the end. A person is visible near the end, standing in a doorway, but not enough to affect the results greatly.

Once the data was collected, the next step was to run the genetic algorithm on it. This is the “meat” of the experiments, and is described in the next chapter.

5 Offline Learning: Basics

With the data collected, the next step was to pick out by hand the lowest non-ground pixel in various columns of each image. This *ground truth* was then used to judge the genetically evolved programs, which attempted to estimate that same information in each image. The details of the ground truth extraction, along with the representation, the performance of the runs and the details of the evolved individuals are explained here.

The phrase "ground truth" can have many senses, but here it simply means an externally given answer that is considered correct. Below is discussed how accurate the ground truth really is, but outside that discussion it is assumed that any differences between it and the objective truth are not significant. In particular, if the value returned by any given individual differs from the ground truth, the error is assumed to be with the individual and it will be less likely to be selected as a parent of the next generation.

Record Video Robot, Real Time

Learn Offline Genetic Algorithm

Build Navigation By Hand

Validate Online Robot, Real Time

Ground Truth Determination

For the purpose of ground truth, a column is a single pixel wide. For example, when determining the lowest non-ground pixel in column 105, objects at horizontal locations 104 or 106 would be ignored (unless they were also present in column 105 of course). Thus obstacles in much of the image are not captured by this representation. In practice this was not a significant problem.

While sonar and dead reckoning data were recorded during data collection, only the images were used. Since the representation used during learning only admitted reactive algorithms, the time between images was irrelevant. To keep run times and memory reasonable, only every *n*th image was used, where *n* was chosen so that the total number of images was approximately seventy. At this size the genetic programming runs took approximately 24 hours each on a dual 700 MHz Pentium III. In runs from a single data set, every third to every fifth image was used. This *n* was compiled into the two ground truth programs mentioned below, as well as the actual genetic programming code. Therefore, whenever the data set changed, these programs would need to be recompiled.

Ground truth was stored in a separate file. The file format allowed a separate result to be stored for each of the 320 columns in the image, although in practice exactly six were used for each image. This file was created by a non-interactive program that filled it with a "first guess," based on the Liana Lorigo algorithm described in the Related Work chapter. This program was written and run under BeOS, although it was written entirely in generic C++.

A simple interactive program could then be used to modify the values by hand. A screen shot is shown in Figure 1. This program was written and run under BeOS and for the user interface used Marco Nel-

Figure 1: This program allowed the user to specify the lowest non-ground column in each image. The crosshairs could be moved vertically but not horizontally.

Figure 2: Typical Ground Truth Images

issen's liblayout [Nelissen 2000]. It displayed whichever columns already had values in the ground truth file and did not allow the user to add, delete or move columns, since this capability was never needed. Instead, when the user clicked on the image, the value in the closest column was set to the vertical location of the pointer, and would track the pointer as long as the mouse button was down. Whenever the user chose, the entire ground truth file could be written out by clicking on the "save" button. The user could go to the previous or next frames, or type in the number of a desired frame.

Typical images are shown in Figure 2. Using this method, ground truth could be determined to within plus or minus two pixels. There were a few questionable cases. When objects came within a few pixels of the desired column, but did not enter it, they were considered to be ignored and did not count. However, since the evolved algorithms only got summary statistics over a window, it might be more reasonable to count objects in nearby columns as "close enough," especially since they may be significant for obstacle avoidance. But the problem of deciding how close was close enough when window sizes were under evolutionary control was considered thorny enough to avoid. Another possibility was for each column to have a number of "correct" answers, and reward potential algorithms for getting close to any of them. In practice, most columns would have a single answer, and two columns would take care of most of the rest. However, this would have increased the scope of the thesis for questionable benefit.

Another area of concern was nearly vertical objects, such as walls parallel to the direction the camera was facing. Again because of the larger-than-a-pixel window size, the program could easily decide that the obstacle was higher or lower than it actually was in the desired column, and yet be discerning the right thing, the visual cues of a wall boundary. Again, it was decided not to worry about this for the same reasons as in the last paragraph.

A final judgement call in assigning ground truth was in cases where shadows caused the image to be visually different near an object. For example, there is often a gap between doors and the floor underneath, and the door casts a shadow that shows up a few pixels below the door. While technically not a non-ground pixel, the beginning of the shadow was marked as the ground truth for a number of reasons. First, the shadow is a natural sign for the door, i.e. the presence of a shadow is a good indicator that the door is nearby. In probabilistic terms, the conditional probability of an obstacle given the shadow is very high. Of course, the shadow was marked as the beginning of the obstacle only when there really was an obstacle. Second, the lowest-non-ground-pixel representation is generally only useful when all obstacles have support, that is, when they touch the ground. For example, a table could be very high in the image, so the obstacle avoidance algorithm assumes it is far away, and yet be very close. These shadows typically happened in the space between an object and the ground, and so the beginning of the shadow is actually the most useful indication of the distance of the obstacle, and therefore the most useful result the evolved program could return.

And finally, the beginning of the shadow was selected because the evolved algorithms tended to look for large visual differences as the indicator of transition to non-ground. In other words, the algorithm was most likely to find the shadow boundary anyway, and since it was a nat-

ural sign for the presence of an obstacle, and in the right place, it seemed best not to punish the algorithm for finding it.

Two passes were made using the interactive program, the second to double check and make minor adjustments. Both passes were done on a 19" screen of resolution 640 x 480, and since all images were in 320 x 240 in resolution, that meant each pixel was approximately 0.6mm on a side, and were quite easy to distinguish.

In the ground truth for the seeded expanded representation runs described below, three errors were found out of 450 total fitness cases (75 images). The errors were in the first three images, all in the same column, where the wall goes to the bottom of the image. The correct answer would therefore put the ground truth at the very bottom of the image, yet the indicated position was the next visual feature above that. Since the best individual achieved a 10% error rate, and that three mislabelled points out of 450 is less than one percent, it is unlikely that these mislabellings caused any significant change in the results. In other words, the errors in all evolved individuals swamped the few errors caused by this mislabelling.

The Genetic Programming Engine *Creator*

Following good software engineering practice, the general genetic programming code was kept separate from the application specific code. The general genetic programming engine was initially developed in the six months before the author started at Carnegie Mellon and was his first significant project in C++, and his first significant object oriented design. However, it was very basic at that time, only able to run very simple problems such as symbolic regression or the artificial ant. In particular, checkpointing, constrained syntax and automatically defined functions, described below, were absent.

It was revamped after the thesis proposal, although the high level design has stood up well over time and is largely unchanged. The engine was kept to a higher standard of readability and organizational cleanliness than the dissertation specific code, and will be released open source after the dissertation under the name *Creator.*

Features of *Creator*

Creator supported many "advanced" features of GP that were needed for this thesis, including automatically defined functions and constrained syntax.

An automatically defined function (ADF) [Koza, 1994, *Genetic Programming II*] is a function, or subroutine, or procedure, or module, or program, that is evolved during a run of genetic programming and which may be called by the main program that is being simultaneously evolved during the same run. Each individual consists of one main program and one or more other programs (the ADFs). In the main function, a call to an ADF appears just like any other function (if it takes arguments) or terminal (if it does not).

In vanilla GP, any node can appear as a root, or as the argument to any function. That would not have worked for the dissertation because, for example, the nodes that return image values could not be used to determine where the window starts. Therefore, Koza's *constrained syntax* method was used. In this method, the programmer assigns a type to the return value of every function, and specifies the list of acceptable

types for each argument. In this document these are called GP types, to distinguish them from data types in C++. Creation and crossover of individuals take these restrictions into account.

After the new population was created, but just before it was evaluated, the entire population was written to a file, known as the *checkpoint file.* Whenever the run was interrupted, which happened four or five times for some runs, it could be resumed from the checkpoint file.

With the exception of population size, initial tree size and maximum size after crossover, the parameters for all runs were taken from Koza's 1994 book *Genetic Programming II,* which were themselves largely taken from *Genetic Programming I.* These are summarized in Table 1, with changes from Koza's values rendered in bold.

Table 1: GP Parameters for All Runs

Population Size	**2000, 4000 or 10,000**
Number of Generations	**51 or 101**
Probability p_c of crossover	90%
Probability p_r of reproduction	10%
Probability p_{fn} of choosing function nodes for crossover	90%
Maximum number of nodes in each branch after crossover	**1000**
Maximum depth of each branch in the initial population	**6 to 9**
Generation method for initial random population	ramped half-and-half
Selection method	tournament selection, group size of 7

Code Design and Implementation of *Creator*

Specifying the Data Type Returned by Nodes

Existing publicly available C and C++ implementations of genetic programming used a preprocessor macro to define the return type of nodes. For symbolic regression this is typically a floating point type, although in general it could be any type, including structures and unions. The use of a preprocessor macros is discouraged in general, for a number of reasons. In this case, only one type can be used throughout the entire application which can make co-evolution awkward to implement, the debugger knows nothing of the original form, and it requires the library to #include a user header file, or the user to modify an engine header file.

Algorithms which work on data of many types are sometimes called *parameterized types* or *generic algorithms,* and the C++ feature to support that is templates. Hence, any class which depends on the return type is implemented as a template. This was true for all classes except the node, although revisions after the dissertation will change that, as discussed below. Poor support for templates in existing compilers (mostly gcc) was the largest source of headaches before the proposal.

Random Number Generation

In any probabilistic scientific computing, it is important to have an effective randomizer that is capable of producing a stream of independent random integers. Non uniformities, correlations between consecutive numbers and other biases can cause dramatic shifts in program performance, yet can be nearly impossible to detect and are often maddeningly difficult to track down. Implementations of the standard C

random number generator `rand()` vary in quality and generally can not be trusted. A particularly abysmal example was found on SunOS and Solaris, perhaps the most popular operating systems for scientists in the 1990s. As the manual page for `random()` under SunOs and Solaris pointed out, "`rand`(3C) produces a much less random sequence-in fact, the low dozen bits generated by `rand` go through a cyclic pattern." It turns out that the least significant bit alternated 0, 1, 0, 1, ... on successive calls. Adding to the problems, these facts are not documented anywhere on the manual page for `rand()` itself. For these reasons, it was considered important to implement a generator with known good properties.

According to the survey article [Anderson 1990, *Random Number Generators on Vector Supercomputers and Other Advanced Architectures*], multiplicative congruential randomizers can be very fast yet produce a sufficiently random stream of numbers. In particular the Park-Miller randomizer [Park *et al.* 1988, *Random Number Generators*] is well documented and has come into widespread use due to its especially good randomness by many tests for the low-order bits, its ease of implementation and speed of execution. It generates a new random number from the previous number using the formula:

$$7^5 x \bmod (2^{31} - 1)$$

where x is the previous integer. It should be noted that 2^{31}-1 is prime.

To start the process, the x used to compute the first integer was the result of calling the standard C library function `time()`, i.e. the number of wall clock seconds elapsed since Jan. 1, 1970. Note that the first x must not be either zero or 2^{31} - 1; these were explicitly checked for and never happened.

A Tour of *Creator's* Major Components

Since genetic programming is still an area of active research, Creator is written with extensibility in mind. Therefore, each component such as individual or selection method has its own abstract base class that defines an API. The APIs are carefully chosen to place as few restrictions on the other components as possible. The components are listed and briefly described in Table 2. Detailed descriptions follow.

Representation of Nodes and Individuals

The node component defines the data that is stored in every node, namely the name of the node, a pointer to the C++ function that implements it, the number of arguments, the GP types of the return value and arguments, the floating point value of a constant, and the relative frequency for this node in the initial, randomly created individuals. The per-node memory cost dominates the memory usage of Creator, so storing all of these values in every individual would prove wasteful. Instead, the individual stores a byte (`unsigned char` in C++), the index into an array of the actual data. Therefore, the per-node memory cost of individuals is a single byte. This allows 256 unique nodes. In practice, the set of possible nodes was approximately 90, leaving about 165 values to represent distinct random floating point constants.

During evaluation, a different representation is used. The execution time of nodes such as addition and division is dominated by the time to look up the index of each argument in the array and extract the function pointer, and this cost significantly impacts the run time as well. There-

Table 2: The Major Components of *Creator*

Node	Records data about a single function or node, such as the name, the pointer to the C++ function that implements it, and the number of arguments.
Individual	Stores a collection of nodes and the relationship between them.
Population	A container for individuals, this is simply an array.
Selection Method	Given a population with fitness assigned to each, selects individuals to be used with the genetic operators.
Genetic Operators	Given members of the old population, creates members of the new population.
Top Level	The main loop, this decides between, for example, generational and steady state dynamics.

fore, before evaluating a particular individual, a different representation was constructed, which is deleted once that evaluation is completed. In this representation, the function pointer was stored along with the index. Since a structure that is at least one machine word in size must be a multiple of that word size, three more bytes could be stored "free of charge." It was decided to also store the number of arguments, since this is the only other data of the appropriate size that is generally used during evaluation.

The Individual is responsible for storing the relationship between nodes, that is, which nodes are in which ADFs and which nodes are the arguments to which other nodes. The issues for vanilla GP were explored in *Genetic Programming in C++: Implementation Issues* [Keith *et al.* 1994], which I coauthored. In that paper, the best trade off between size, complexity and speed was found to be a linear array, storing the tree in prefix order, and that is the representation used here.

During evaluation a second representation becomes feasible, where along with each node is stored a pointer to each of its arguments. This is the more traditional representation for tress in computer science. In cases where we need to skip the execution of a subtree the pure prefix approach requires run time proportional to the size of the subtree, whereas this new approach requires no time. However, profiling showed that the time spent skipping subtrees was relatively small, so this optimization was not implemented. It could easily be implemented, and I intend to do so before releasing Creator.

Conceptually, what is in a node is distinct from the relationships between nodes. Therefore, the individual class can work with different nodes. From a software engineering standpoint, the cleanest way to implement this would have each node call virtual functions to get the value of its children. However, this would be far too time consuming. Therefore, the individual takes the node as a template argument. Also, preprocessor macros are used to isolate the programmer from many implementation details without sacrificing efficiency.

Creating the initial, random population

All ADFs of every individual were required to have at least two nodes, that is, the root was required to be a function node, a node that had argu-

ments. This ensured that every individual had at least one function node and one terminal node. When individuals were created for the initial population, a maximum depth was specified; any node at the maximum depth was a terminal. For a quarter of the population this depth was six, for another quarter seven, another quarter eight, and the last quarter nine. What's more, in each quarter, half of the individuals were full trees; all nodes a depth less than the maximum were function nodes. This distribution for the initial random population is known as *ramped half-and-half*.

On top of these restrictions were restrictions on GP types. The programmer provided the set of allowable types for the root of every ADF, and when the programmer specified the list of problem specific nodes, the programmer was also required to specify each node's return type, as well as the types of all arguments.

To achieve these restrictions efficiently, the creation of random individuals for the initial population proceeded as follows. The root was chosen from the set of all function nodes that return an appropriate type. Then, each argument was recursively chosen from among those that return the needed type, and that also satisfy the function vs. terminal constraint for the appropriate depth. If a function node was specified for a particular location in the creation of a full tree, but only terminals fit the type requirements, then a terminal was used instead.

The selection from the set of eligible nodes was not uniform. Instead, the programmer specified a relative frequency for each node. Once the set of eligible nodes for a given argument was determined, the relative frequencies were totaled for all nodes in the set, and each node's probability of being chosen was (that node's relative frequency) / (total of all relative frequencies for nodes in the set).

The method of storing nodes allowed a maximum of 256 unique nodes. There were approximately 90 non-random-constant nodes, which allowed the rest to be used as random constants. During creation, whenever a random constant was needed, if there was an empty slot a new value was chosen randomly and assigned to that slot. If no slot was available, an existing random constant was reused.

Finally, as was standard practice in genetic programming, all the individuals in the initial, random population were unique. This was implemented by checking, after each individual was created, whether an identical individual already existed in the population. If so, the new individual was discarded.

Genetic Operators

The only two genetic operators were reproduction and crossover. Reproduction simply copies its single parent into the new generation. and was used 10% of the time. Crossover, used in the remaining 90% of cases, takes two parents, and selects a random node in the first.

The method of selecting parents used throughout this dissertation was tournament selection with a group size of seven. Whenever an individual was needed for reproduction, seven individuals are chosen uniformly with replacement and the fittest one is selected. When two individuals were needed, for crossover, each was selected independently, that is, two groups of seven were chosen, and the fittest individual in each were used.

Crossover starts by choosing a random node from the first parent. In practice, the number of terminals is comparable to the number of

functions. In fact, in a binary tree, the number of terminals will always be exactly one more than the number of functions. To avoid a preponderance of terminal-only crossovers, 90% of the crossover points are drawn from the function nodes, and only 10% from terminals.

Therefore, crossover started by deciding whether the node in the first parent would be a function or a terminal. It then selected the node uniformly from all such nodes in the individual. For example, if a function was desired and a given individual had 100 function nodes in the result producing branch, and 200 in the only ADF, then the crossover point would be in the ADF 2/3 of the time.

The type of the selected node is determined. The subtree in the second parent was always from the same branch as the first parent, i.e. the the subtree in the first parent was from the result producing branch, the subtree from the second node was required to be from its result producing branch. If there were no nodes of the proper type in the desired branch of the other parent, the entire process was started again. If this failed 100 times an error message would be printed; in practice this never happened. Then, function vs. terminal was decided for the second parent; this choice was independent of the choice for first parent. Thirty attempts were made to find an appropriate node, including both function v.s terminal considerations and GP type. If all thirty failed, the function vs. terminal constraint was abandoned, since a node of the given GP type is known to exist.

At this point, both subtrees had been determined. The total number of nodes in each child were calculated, and if a given child would have been over 1,000 nodes, the child was replaced with one of the parents, not the parent that would have donated the subtree, but the other parent. This was standard GP practice, without it programs grow ever larger as the generations wear on.

Following the common practice in genetic programming, no mutation was used. Because the crossover copies an arbitrary subtree from the second parent, much of the time this subtree will be used out of context. It is therefore similar to inserting a random subtree, although the distribution is certainly not uniform. This is thought to obviate the need for mutation [Koza 1992, *Genetic Programming*].

The Remaining Classes

The population is simply an array of pointers to individuals, implemented as a Standard Template Library vector.

Finally the Top Level class ran the main GP loop. At the beginning of every generation, just before the evaluations started, the checkpoint file was written. This was simply the current seed of the random number generator, the current generation number, then all of the individuals one after another, written out as text in LISP format. To guard against program interruptions during the checkpoint writing, it is first written to a new file with a different name, and only once that is finished is the old checkpoint file deleted and the new file renamed to the old name.

Optimization for Speed

Other than the use of macros explained above in the context of nodes and individuals, two optimizations were affected. First, since the experiments were run on dual processor machines, the evaluations were done in two separate threads. A client server architecture was affected, where whenever an evaluation was completed, the associated thread would claim the next unevaluated individual. In this way, both threads (and

therefore both CPUs) worked 100% of the time, except possibly during the last evaluation. The vast majority of the time was spent evaluating individuals, generally more than a CPU-hour per generation, and as much as six CPU-hours for the largest populations, compared to ten to thirty seconds for everything else. Therefore there was no practical advantage to parallelizing anything else. So, the only BeOS specific code in all of Creator is this multithreading.

The second optimization was in the evaluation of automatically defined functions (ADFs). ADFs can be called multiple times, and because of the dynamics of crossover often have identical arguments. If the ADF is a "pure function," that is its result depends only on its arguments and it is deterministic, then after it is evaluated once with a given set of arguments, both the values of the arguments and the result can be stored. In future, the arguments are always evaluated, but if they evaluate to a previously-seen value, the ADF is not run again; the stored result is simply returned. There are a fixed number of slots for arguments-value pairs; once they are filled, no new values are stored.

Only the Expanded Representation runs used these sorts of ADFs, so only they benefitted from this optimization. What's more, these sorts of ADFs were not used in the body of the iterated rectangle branches where most of the time was spent.

Verification of *Creator*

There were three primary methods used to verify that the implementation worked properly: liberal use of assertions and invariants, comparing test cases to hand computed results, and replicating published results.

Asserts and Invariants

First and foremost, the best way to produce bug free code is to be very careful to avoid bugs in the first place. A number of features of object oriented programming are aimed at this, such as strong typing and data hiding.

Techniques from "programming by contract" also offer a lot of help. In particular, asserts and class invariants were used extensively. As mentioned above, the vast majority of run time was spent in evaluations; therefore, asserts were used liberally throughout the code, during debugging but also left in the production versions of the code. The guiding principle was that, if I could not prove a condition by looking at a small window into the current code, I would add an assert. These assertions helped greatly, often triggered months or years later when some half-remembered condition was violated, substantially cutting debugging time.

An invariant function was created for both the Individual and Node classes, and sprinkled liberally during creation and crossover. For every ADF, the version for individuals would check that the type specified for the root was a subset of all possible types, that the allocated space for nodes was not less than the number of nodes, that each ADF had two or more nodes, and that the type returned by an argument matched the type needed by the calling function. Various checks were also made for the Node class, including that the function pointer was non-null for all non-random constants, that no random constants were NaN, that the number of arguments was non-negative, etc.

Examining program output critically also proved useful. It was noted at the end of the Expanded Representation experiments that the evolution would take a slightly different path if stopped and restarted than if left untouched. This was due to the limited precision used when writing out floating point numbers in the checkpoint file. For the Focused Representation runs, the precision was changed to be enough to represent a double without any loss of precision, plus an extra digit just to be safe.

(× X (+ Y (× X X)))	4.34708
(× Y (× (+ X Y) X))	7.98151
(× Y Y)	4
(× (× X (+ X (+ X (× Y Y)))) (× X Y))	19.6984
(× (× (+ Y (× X X)) Y) Y)	14.091
(× (+ (× (+ X Y) Y) X) X)	9.50427
(× (+ Y X) (× (× Y (+ (+ Y Y) (× Y X))) Y))	83.67
(× (× (+ (+ (+ Y Y) (+ Y X)) (+ (+ Y X) (× Y Y))) X) (+ (+ Y (+ (× X X) (+ X Y))) Y))	156.339

Randomly produced individuals and their values. $x = 1.234$ and $y = 2$.

Test Cases

Each module was tested as it was completed. The Park Miller random number generator was tested by seeding it with the number 1, running it 10,000 times, and verifying that the result was 1,043,618,065, as published in [Koza 1992]. There were some problems in the unused Cray port, which uses 64 bit ints, but other than that no problems have ever been found in the random number generator.

The Node specific code is very simple, its essentially a wrapper for an array. To test it, ten nodes were created at random, and their name, number of arguments and function pointers printed out. It should be mentioned here that Creator's origins date back almost a decade, and that the core tests were written very early on. Whenever a change was made, the tests were rerun to verify that the output had not changed. Therefore, the core functionality of the recent, large version of *Creator* has been tested against the results of a much simpler and smaller version, and found to give the same output.

The Individuals were tested by creating eight random individuals (expressions) using only addition, multiplication, and two variables, and evaluating them. The values were then calculated by hand, and verified to be correct. The expressions and their values are shown at left. Similar tests were performed for individuals with ADFs and individuals using GP types.

The population and ramped half-and-half creation methods were tested by creating a population of ten individuals limited to a maximum depth. This meant that many randomly created individuals would happen to be identical, thus testing the process of ensuring uniqueness of the initial, random population. Tests were also done using individuals with ADFs and random constants, which did not stress the uniqueness.

Replicating Published Results

Three problems from [Koza 1992] and one problem from [Koza 1994] were implemented, and the results compared to the published results. The main variable for comparison was the published success rate. If any individual in a given run met or exceeded a designated level of performance, the run was considered a success. The percentage of successful runs is a very sensitive measure of performance, since even subtle changes to the dynamics of the evolution can affect it. It therefore is both a very powerful test, yet when results differ often gives no clue as to why.

In general, because genetic programming is a randomized algorithm, even in two identical setups the percentage of successful runs will vary. Therefore, the statistical technique of hypothesis testing was used to determine whether two setups were "close enough." In particular, since each run is classified as "success" or "failure," it can be consid-

ered a sample from a binomial distribution. Different runs of the same setup are truly "independent and identically distributed," in our case this is not an approximation. There is only one parameter that affects the distribution, the probability of success. Thus, for the purposes of comparing success rates, the only data needed from each setup was the number of times GP was run, and the number of those that were successes. The null hypothesis was that the two sets of runs came from the same distribution. To test this, the chi-square test of homogeneity was used [Rice 1988, p. 436]. The Mathematica script that did the comparison is shown on the next page.

```
(* This script is for Mathematica 3.0 and 4.0

   When comparing two different setups A and B, set
   "as" to the number of successes with setup a, "na"
   to the number of runs with setup a, and similarly
   with "bs" and "nb" for setup b. *)

af = na - as;  (* Number of failures in setup a *)
bf = nb - bs;  (* Number of failures in setup b *)

ns = as + bs;  (* Total number of successes *)
nf = af + bf;  (* Total number of failures *)

n = na + nb; (* Total number of runs in both setups.*)

(* ns / n is the maximum likelihood estimate of prob-
   ability of success under the null hypothesis. *)
eas = na ns / n;(* expected no. of succ. under null*)
eaf = na nf / n;(* expected no. of fail. under null*)
ebs = nb ns / n;(* expected no. of succ. under null*)
ebf = nb nf / n;(* expected no. of fail. under null*)

chiSqrAs = (as - eas)^2 / eas;
chiSqrAf = (af - eaf)^2 / eaf;
chiSqrBs = (bs - ebs)^2 / ebs;
chiSqrBf = (bf - ebf)^2 / ebf;

chiSqr =
   Simplify[chiSqrAs + chiSqrAf + chiSqrBs+chiSqrBf];

<<Statistics‘ContinuousDistributions‘

p = 1 - CDF[ChiSquareDistribution[1], chiSqr];

(* "p" is the probability that you would get differ-
   ences this big or bigger, assuming that the proba-
   bility of success is the same for both methods.  If
   it’s < 0.05, then the difference is significant at
   the 5% level, similarly for 0.01 and 1% *)

N[p/.{as->"a successes", na->"Number of a", bs->"b
   successes", nb->"Number of b"}]
```

The first test was the symbolic regression of $x^2/2$, described on page 706 of [Koza 1992]. This particular setup came with a complete LISP implementation, facilitating the tracking down of differences. Page 718 reports the results of 190 runs which lead to a success rate of 67%, meaning 127 or 128 successes.

The *Creator* implementation achieved 3434 successes in 5100 runs; the probability of getting this at random, if the two implementations had exactly the same success rate, is 88.7%. If the runs from the unused Cray port are included, the numbers are 9520 successes out of 14,100 runs, for a probability of 84%. The estimated success rate is 67.5% $\pm$ 0.77% (95% confidence interval).

This comparison caught an earlier, very subtle bug. At one point, the success rate was 56.6% out of 1240 runs, which if the implementation was correct, would happen only 0.78% of the time. The problem turned out to be what happened when crossover produced an individual that was too large. In the early *Creator*, both children were replaced with their parents, whereas in Koza's code, only the "too big" child was replaced. This demonstrates the power of comparing probabilities of success, it would most likely have been impossible to discover this without such a comparison.

The second test compared the implementation of ADFs, using Koza's example of determining the 5 input boolean even parity function. This is a symbolic regression problem where the variables are boolean, i.e. have one of two values, "true" or "false." The designated problem was to compute the even parity of five such values, that is, the

negation of the exclusive-or of all of them. Again, this problem came with the Lisp code in the 1994 book, so a reference implementation was available.

Koza achieved 25 successes in 32 runs for a success rate of 78%. The *Creator* implementation succeeded in 59 out of 79 runs, for a success rate of 75%. If the two distributions were in fact identical, a difference this large or larger would happen by chance 70% of the time.

It should be noted that attempts to replicate performance on another problem, symbolic regression of the sextic polynomial $x^6 - 2x^4 + x^2$ met with failure. In 13 runs using a different implementation, Koza found 7 successes or 54%. The *Creator* implementation succeeded in 63 out of 200 runs, for a success rate of 31.5%. If these were from the same distribution, a difference that great or greater would only happen 9.6% of the time.

There are many possible reasons why. In the book, 50 test cases were randomly selected and used for all runs, although they were not reported; perhaps they were a "lucky set." Or perhaps slight differences in the implementation lead to it. Also, Koza does not mention some of the details of his experimental design, in particular how the total of 13 runs was decided. And finally, bugs in either implementation cannot be ruled out.

The last test was of the constrained syntax code. While Koza describes a number of constrained syntax examples in chapter 19 of the 1992 book, he only gives performance numbers for one, the symbolic regression of the cross product, phrased as a vector valued function of four scalars: $f(x_1, x_2, x_3, x_4) = (x_1x_3 - x_2x_4, x_2x_3 + x_1x_4)$. His success rate from 300 runs was 13%, while *Creator* achieved only 3.5%. It was reimplemented twice, the last time as a completely separate program; that program achieved 370 successes out of 10,478 runs, or 3.53%. Finally, a public domain GP implementation that supports constrained syntax was run; it came with the desired problem already implemented as an example. In one thousand runs it achieved 38 successes or 3.8%. If this and separate program came from the same distribution, a difference this large or larger would happen 66% of the time.

To track down the error, the descriptions in Koza's book were examined in detail, and different possible interpretations were tried. In desperation, the fitness function was changed and tournament selection were used, but the highest observed success rate was 8.67% (26 successes / 300 runs). Given that 3 different implementations by two different people agreed with each other but disagreed with Koza's, it was concluded that any error was most likely not in Creator. It would be interesting to implement this problem in Koza's own GP implementation and see how the success rate compares.

It should be noted that the statistical test described here can be applied to Koza's claims in [1994] that ADFs help the success rates of certain problems. It turns out, many of the differences are not statistically significant, although some still are and his main point is still valid. I was a reviewer for the book and pointed this out to him before it was published, but no changes along these lines were made to the book.

It should also be noted that these tests are much weaker than the norms in the software industry. Activision, for example, devoted an entire floor of its four story office building to quality assurance, filled with

rows upon rows of people amazed that they could actually get paid to play games all day. It is considered essential for the testing to be done by people outside the development group. However, the efforts reported here are believed to be above the norm for academic programming.

With the basics of data collection and the genetic programming engine explained, the details of the experiments can now be described.

6 Expanded Representation

As mentioned previously there were to sets of experiments, one for each pass through the outer loop described in the Overview chapter. This first set of experiments is described in this chapter; the second in the next. The representation used here, while more general, proved challenging for the genetic algorithm. This chapter describes that challenge and the methods used to rise to it.

The Representation of Learned Programs

It will be recalled from the Technical Framework chapter that each execution of an evolved program will be given an image and the x location of a column, and its single real-valued output will be interpreted as the y location of the lowest non-traversable pixel in that column. In particular, there is no state maintained from one execution to the next, and therefore all programs are purely reactive. To measure fitness, the individual is executed six times on each image, in six different columns, on sixty to seventy five images.

Record Video — Robot, Real Time

Learn Offline — Genetic Algorithm

Build Navigation — By Hand

Validate Online — Robot, Real Time

The list of all possible nodes is summarized in Tables 1 and 2, and an example individual is shown in Figure 1. For the most part they are elements from traditional programming languages, such as arithmetic operators, if statements, and reading and writing of registers. Those elements have been used in many GP applications and everything but the image iteration nodes and reading and writing of registers could be considered standard in the GP community. The only obviously application specific nodes are the iterated rectangle and image reading nodes, which are described in the next section.

A Tour of the Representation

This section presents a high level overview of the representation and explains the working of an example in detail. The next section describes the implementation of each node in detail.

The chapter "Technical Framework" describes the iterated rectangle as a building block of existing algorithms. The expanded representation experiments permitted these rectangles to move either horizontally or vertically. These were represented by a pair of nodes, `Iterate-Horiz` and `Iterate-Vert`, which took five arguments.

The first argument was one of 17 tokens representing various rectangle sizes from 2×2 to 8×8, as well as 20×20. Thus, the subtree for this argument always consisted of a single terminal, and these rectangle size tokens were never used in any other part of the individual. The second and third arguments were an expression for the *x* and *y* locations of the initial location of the center of the window, in pixels. The fourth argument was the *x* location (for horizontal movement) or the *y* location (for vertical movement) of the final location of the center of the window. The final argument was a block of code to be executed for every loca-

Table 1: The Node List for the Iteration Branches

Image Iteration	Iterate-Vertically, Iterate-Horizontally
Window Size	2x2, 2x3, 3x2, 3x3, 2x4, 4x2, 4x4, 5x5, 2x6, 6x2, 3x6, 6x3, 6x6, 7x7, 2x8, 3x8, 8x8 and 20x20.
Mathematical	+, -, ×, % (protected division), square
Flow Control	if-le (if-less-than-or-equal-to), prog2, prog3, break
Miscellaneous Terminals	desired-x, random constant, window-area, image-width, image-max-y, first-rect, x, y
Registers	set-a, set-b, set-c, set-d, set-e, read-a, read-b, read-c, read-d, read-e
Image Operators	raw, raw-squared, median, median-squared, median-corner, median-corner-squared, gradient, gradient-squared, moravec-horiz, moravec-horiz-squared, moravec-vert, moravec-vert-squared, moravec-slash, moravec-slash-squared, moravec-backslash, moravec-backslash-squared

Table 2: The Node List for the Result Producing Branch

Mathematical	+, -, ×, % (protected division), square
Flow Control	if-le (if-less-than-or-equal-to)
Miscellaneous Terminals	desired-x, random constant, image-width, image-max-y
Final Register Values	mem0a … mem0e, mem1a … mem1e, mem2a … mem2e
ADFs	ADF0, ADF1

tion of the window. Thus, for each execution of this branch, all arguments except the last would be evaluated once, and the last argument would, in general, be evaluated many times.

This last argument could access the *image reading terminals.* There were 16 of these. Half of them were the average value over the window of one of eight common image statistics, as illustrated in Figure 2. The other half were the average of the squared value of the same statistics. This allowed the computation of second order statistics, including variance and standard deviation.

From the work of Teller and Veloso [1997] on PADO, the idea of memory locations which can be read from and written to were borrowed. Each iteration branch had five floating point registers, which were accessed through five write nodes and five read nodes. The write nodes each took a single real valued argument, while the read nodes took no arguments.

As mentioned before, a `break` node would halt the current loop, freezing the values of the five registers. The remaining nodes were the usual suspects in genetic programming: standard mathematics functions, control flow provided by `if-le` (if less than or equal to), `prog2`

and `prog3`, random constants, and terminals to provide the image size, the area of the rectangle, the *x* location of the desired column, and the current location of the window. `If-le` took four arguments; if the first was less than or equal to the second, then the third was evaluated and returned and the fourth was not evaluated, otherwise the fourth was evaluated and returned, the third going unevaluated. `prog2` and `prog3` took two and three arguments respectively. Each argument was evaluated in turn, and the value of the last one returned.

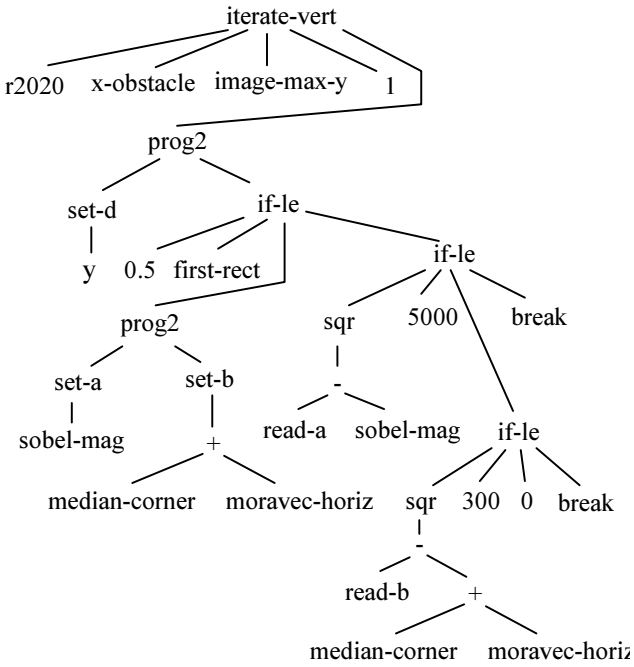


Figure 1: An example individual in the expanded representation.

In a modification of Koza's "automatically defined functions" [Koza 1994, *Genetic Programming II*], each individual consisted of three `Iterate` branches, one result producing branch, and two automatically defined functions that were usable only by the result producing branch. The return values from the three `Iterate` nodes were discarded; instead, the the final values of the five registers were used as the result of each branch. Thus, the job of the result producing branch was to combine the fifteen such values into a final value, representing the vertical location of the first non-ground pixel.

The result producing branch took as input (i.e. leaves of its tree) only the final register values and fixed numerical constants, and could not access the image directly. Its functions were standard mathematical functions and Automatically Defined Functions taking one argument each, as described in [Koza 1994]. They were callable only by the result producing branch.

The result producing branch used the subset of nodes that had meaning outside of window iteration. Since this eliminated the set and break nodes, which were the only nodes with side effects, there was no longer a need for the `prog` nodes. The only nodes added were the final values of the registers in the three iteration branches, and the ability to call the two ADFs.

The node list for the ADFs was similar to that for the result producing branch, except they could not access the final register values, could access their arguments, neither could call ADF1, and ADF1 could only call ADF0.

As an example, a simple version of Liana Lorigo's algorithm is shown in Figure 1. This example was created by hand to demonstrate the possibilities of the representation and was not produced by any GP run. The root of the tree (`iterate-vert`) says the window will be moving vertically. The first four arguments mean it will be 20×20 pixels, whose center starts at the coordinate (`desired-x`, `image-max-y`) and whose end is at (`desired-x`, `1`). Therefore, this window will move from the bottom up, and at every location it will execute its fifth argument, the rest of the tree. The value returned by this argument is discarded.

The "`prog2`" executes each of its arguments in order. In this case, it first sets register "`d`" to the *y* position, then checks to see if 0.5 ≤ `first-rect`. Since `first-rect` equals 100,000 the first time through (i.e. at the bottom of the image) and -100,000 thereafter, the left branch is executed the first time, and the right branch all subsequent times.

The left branch simply sets registers "a" and "b" to values representing summaries of the image over the 20×20 rectangle. For example, register "a" is set to the average of the Sobel gradient magnitude over those 400 pixels.

Finally, on subsequent iterations, the remaining branch compares the stored values to freshly computed ones at the new location of the rectangle. If the difference is small, i.e. if (a - sobel-mag)² ≤ 5000 and (b - (median-corner + moravec-horiz))² ≤ 300 we keep iterating, otherwise we execute a break which stops the iteration. Note that the value in register "d" will be the *y* coordinate of the window where it stopped, or the *y* coordinate of the highest rectangle if no break statement was executed.

Figure 2: A dewarped image and the image operators applied to it: truncated median, median corner, Sobel magnitude, and four directional Moravec interest operators.

Details of Nodes

The list of all nodes was the same for all expanded representation experiments and is shown in Table 1 and Table 2. Protected division, a standard GP function, returns one if the denominator is zero, thus avoiding divide by zero. This extension also means a÷a equals one even when a is zero. The other functions are already defined for all real valued inputs. As described above, if-le took four arguments; if the first was less than or equal to the second, then the third was evaluated and returned and the fourth was not evaluated, otherwise the fourth was evaluated and returned, the third going unevaluated. Prog2 and prog3 took two and three arguments respectively. Each argument was evaluated in turn, and the value of the last one returned.

Desired-x was the x coordinate, in pixels, of the column that the program would be judged on. That is, desired-x was the column where the lowest non-ground pixel was desired. If the random constant node was selected during the creation of individuals, a random real number was selected uniformly between -5 and +5 and used in its place. Window-area was the area, in pixels, of the window size, that is, the product of the two digits in the first argument to iterate-up or iterate-down. Image-max-x was the maximum *x* coordinate of any pixel; in this thesis that was always 319. Similarly, image-max-y was the maximum *y* coordinate, 239. First-rect only appeared in the third argument to the iterates, and was one when the window was at its startling location and minus one after that. X and y were the coordinates of the center of the rectangle, in pixels, rounded up.

Set-a through set-e took a single argument and set the given register to that value. Read-a through read-e had no arguments and returned the value of the given register. The "set" nodes returned the *previous* value of the memory location. This is the convention used by Teller and Veloso [1997] and is also the convention used in "atomic"

operations common in parallel computing. Returning the argument is the convention in the C programming language.

The image operators came in two flavors, straight up and squared. The straight up versions would return the average value of the operator over the window, and the squared version would return the average of the squared value over the window. `Raw` was simply the raw image pixels, `median` was the truncated median filter of [Davies 1997, *Machine Vision*, pp. 48-55] in a 5×5 window (a kind of low pass filter), `median-corner` was the raw minus the median (and therefore a kind of high pass filter), `gradient` was the magnitude of the 3×3 Sobel gradient operator, and the Moravec interest operators were simply the squared difference between adjacent pixels, either horizontally, vertically, or diagonally.

The random constant node actually represents a family of nodes each with a different value. In Creator, the total number of unique nodes is always 255. Each individual had six branches, each branch was limited to 1000 nodes. This gives an upper bound to the search space of $255^{6000} \approx 10^{14,439}$. While there are some restrictions, this nevertheless gives an indication of the order of the search space.

Other Details

Fitness measures in evolutionary computing have two goals. First, they should select individuals whose children, grandchildren, etc. are likely to do well. Second, because the best solution found is generally not an optimal solution to the problem, it must say how desirable each individual is as a final result.

These are commonly considered to be the same, by assuming that individuals who solve the problem well are more likely than others to have some component that would be useful in a full solution. However, as described in the "Early Experiments" section below, that proved problematic and Illah Nourbakhsh suggested the solution.

The fitness measure used was rather simple. Each column of each image was a separate fitness case. The evolved program was run, and the value stored in the "a" register compared to the hand created ground truth. The fitness was the absolute value of the difference in number of pixels. For each fitness case, this number was capped at 20. The fitness of an entire individual was simply the some of the capped differences over all fitness cases. Lower is better. The sum of absolute differences is the standard error measure in symbolic regression using GP, although it is not usually capped.

The camera was placed near the center of the robot and pitched down 31°, the same setup used by deliberative stereo research that took place previously on the robot. It turned out that the best algorithm developed in these experiments worked for a wide range of camera pitch, including the steepest possible before viewing part of the robot.

After an image was taken it was dewarped using the image dewarping software described in [Moravec 1996, *Robot Spatial Perception by Stereoscopic Vision and 3D Evidence Grids*]. This version replicates pixels, that is, every pixel in the resulting image is the same value as some pixel in the original image, and that adjacent pixels are sometimes equal, even in areas of the image that otherwise have high spatial gradient.

Examples of all image operators are shown in Figure 2. The experiments in this chapter used a truncated 5×5 median filter as described in

[Davies 1997, *Machine Vision*, pp. 48-55], and the median corner operator was the original image minus the median filtered image. The median filter in these experiments was implemented using the C function `qsort()`. The C++ Standard Template Library function `nth_element()`, which is designed for exactly this purpose and should be O(n), was tried, but the implementation used in BeOS, which comes from SGI, performs poorly when called from multiple threads, even on data that is many pages apart in RAM. In fact, it takes longer when called from two CPUs than a single CPU version using `qsort()`! This is mostly likely due to caching issues, where one CPU writes to a memory location needed by the other CPU, so the first CPU's cache must first be written to main RAM before the other CPU can continue. This bug was reported to Be, but never seems to have been fixed. It nevertheless produces the correct answer, although it was only ever used during testing.

Simplifying Evolved Programs

While it is unnecessary to know how a particular program works in order to use it, such knowledge can shed light on several questions. This includes practical questions: Are these solutions that could be implemented by hand, or are they far too subtle and complex? How will they generalize to other environments? Are there any obvious failure modes that we have not seen? Other questions arise when trying to improve the evolutionary algorithm. What features did it make use of? How did it use them? How do those differ from ways humans use them?

Unfortunately, programs created by simulated evolution are often large and convoluted. Thankfully, they also often have branches that are never executed, calculations whose results are never used, etc. Therefore, the first step to understanding evolved programs was to simplify them.

Two main methods present themselves: simplifying by hand and automatic simplification by computer. Simplifying by hand can be time consuming and error prone, although a person can often notice simplifications that would be difficult to teach a computer. However, writing a program to simplify them can also be time consuming, and may contain undiscovered bugs. The choice was even less clear because computers are better at different kinds of simplification than people are.

The path used in this work was to start simplifying by hand, and whenever a pattern was noticed that was straightforward to automate, do so. Since the programs are represented as expressions in tree form, Lisp is the natural language in which to write the simplifier. The simplification code was that in an appendix of [Koza 1992]. This code is an "engine" written in Lisp that requires rules to be added specific to a particular application.

To validate the system, each new simplification rule was tried on a canonical case, and every hint of a bug tracked down. As well, a third of the interesting branch of one individual—approximately 300 nodes—was simplified by hand, and compared to the automated system. Finally, the simplified individual was re-run on the training data to verify that its score had not changed. It should be pointed out that all of these verifications were repeated many times as the simplification rules were added one by one.

The first step was to keep track of which nodes were never actually evaluated. Branches can go unevaluated when the condition of an `if-le` statement always evaluates to the same value, or as the result of a `break` statement. For example, `(if-le 0 300 x y)` will always take the `x` branch. In this case, the entire expression can simply be replaced with `x`. In general, since the first two branches may have side effects, we must keep them along with the always-evaluated branch, turning the `if-le` into a `prog3`.

In the example above, this happens because both branches of the conditional are constant, and will therefore evaluate the same way on any data set. However, we could go further and also eliminate branches that are never executed on the training data set. In this case, on the training data the `prog3` and `if-le` forms will be exactly the same, executing the same operations in the same order and achieving the same results. However, they may perform differently on new data. Which generalizes better? The question could also be asked of existing `prog3`s where the first branch always evaluates less than or equal to the second. In that case it seems arbitrary to convert it into an `if-le` and add a clause that does not affect the learning.

Table 3: Simplification Rules

Rule	Example
Rules that Eliminate Computation	
Remove branches that were never executed on training data	`(if-le a b x y)` → `(prog3 a b y)` if `a` is always less than or equal to `b` when run on the training data.
Remove initial branches of `progn` that do not have side effects.	`(prog3 x y z)` → `(prog2 x z)` if `y` does not have any side effects.
If the value of an arithmetic expression is not used, convert it to a `progn`	`(prog2 (+ x y) z)` → `(prog2 (prog2 x y) z)`
Eliminate calls to "set" if the value will be overwritten before it is used.	`(set-b (+ x (set-b y)))` → `(set-b (+ x (prog2 y read-b)))` if y does not contain `set-b`.
Rules for Human Readability	
Convert two successive instances of `prog2` into `prog3`	`(prog2 (prog2 x y) z)` → `(prog3 x y z)`
Raise `prog2` if it is the second branch to an arithmetic operator, if the reordered branches do not contain `break` and do not read anything the other one sets.	`(+ x (prog2 y z))` → `(prog2 y (+ x z))` if x does not read any register that y sets and vice versa, and if `break` does not appear in either x or y.
Change a double subtraction into a subtraction and an addition.	`(- x (- y z))` → `(+ (- x y) z)`
Raise prog2 or prog3 as the first argument to any operator	`(op (prog3 x y z) a)` → `(prog3 x y (op z a))` if op is not `prog2`, `prog3` or `set-a` … `set-e`

In other words, the genetic programming does not know the difference between `prog3` and `if-le`, except in how they influence evaluation. An `if-le` with a branch that is never executed, and whose three remaining branches are always executed in order, is equivalent to a `prog3` as far as the GP is concerned. The never-executed branch does not affect fitness in any way, so we have no inductive reason to suspect it would do anything worthwhile on new data. Treating it as a `prog3` is more natural.

Therefore, the first step in editing the evolved programs was to run the program on training data, keeping track of which subtrees were never executed on the training set, then eliminate those subtrees. That was done in C, and the rest of the simplifications were done in LISP. The set of simplifying rules is described in Table 3.

The rules were carefully chosen not to produce any loops, where the effects of one rule could be undone by some combination of other rules. Therefore, the rules could be continuously applied until no more could be applied anywhere in the tree. The order of application of the rules could affect the outcome, since the application of some rules render others inapplicable. This could be remedied by suitably generalizing some of the above rules, but in practice was not a problem.

As a side note, it is possible to apply these simplifications during evolution itself. The value of this is a topic of debate in the genetic programming community, and was not tried here.

Finally, the individual was rewritten into a syntax similar to a traditional procedural language and simplified further, for example noting that a variable does not affect the results so that all references to it can be removed, or that the code might usefully be reorganized in some way. Any further simplifications were converted back into Lisp form and verified on the training set. Often, a hand simplification would allow more automated rules to apply, so hand simplifying and automated simplifying were alternated.

Early Experiments

A number of informal experiments were performed before deciding on the details of the actual experiments. These informal experiments are detailed here. They all produced results that were no better than the best constant approximation, i.e. the best evolved programs did not even examine the image, at least in a useful way. These experiments were various combinations of the modifications described in the next paragraph.

All of these experiments ran to 100 generations, and used a population size of either 2000 or 10,000. In some of them, the fitness measure was not capped at 20 for each fitness case. That is, the total fitness was computed using the traditional method for symbolic regression, namely as the sum of the absolute value of the difference between what the individual returned and the ground truth. The worst of the initial population was therefore routinely “Infinity” (in the IEEE floating point sense), and even in final population the worst fitness would exceed 1e+77 in some cases.

The individuals in the initial, random population were examined and found to be very shallow. Any program that examined the image in non-trivial ways would need to combine a number of affordances of the language, and it was hypothesized that the initial programs were simply

too small. Originally, the initial tree size followed that of [Koza 1992], namely a minimum depth of four and a maximum of nine, so the minimum was eventually changed from four to six. Also, the frequency of function nodes was quintupled, making the total probability of choosing a function node 59:25, which exceeds 2:1.

Table 4: Summary of Informal Experiments

Large Initial Size	Fitness Capped	Seed	Population Size	Number of Runs
N	N	N	2000	6
N	N	N	10,000	3
Y	N	N	2000	5
Y	Y	N	10,000	2
Y	Y	Y	2000	1

The example shown in Figure 1, the simple Lorigo algorithm, was at one point used as a seed. That is, it was used as one individual in the initial population, and all of the remaining individuals were chosen randomly as before. Since this individual did better than constant, these runs broke the "constant barrier." They improved on the seed a little, but not significantly.

The final change, the one that resulted in greatly improved performance, was to the allowable size of an individual after crossover. For all these informal experiments it was 200 nodes per branch. However, after some investigation, it was determined that the randomly created individuals often had branches with over 800 nodes, occasionally over 2000. This would cause crossover to almost always fail, degenerating into reproduction. When the maximum size after crossover was changed to 1000, the runs all of a sudden started performing better than the best constant approximation. After that, there was no turning back. The various experiments are summarized in Table 4.

Results

The summary of the training data that appeared at the end of the Data Collection chapter is repeated here:

Field Robotics Center Hallway

Camera Angle: 31 degrees from horizontal
Total Number of Frames: 226
Frames In Training Set: 75 (every third)
Number of Fitness Cases: 75 × 6 = 450
Elapsed Time: 52 seconds
Frame Rate Of Training Set: 75 ÷ 52 = 1.44 fps

Starts with robot in lab doorway. Moves straight until its in the hallway, then turns right, travels down hallway, at end turns right, then travels straight. All doors closed, except near the end. A person is visi-

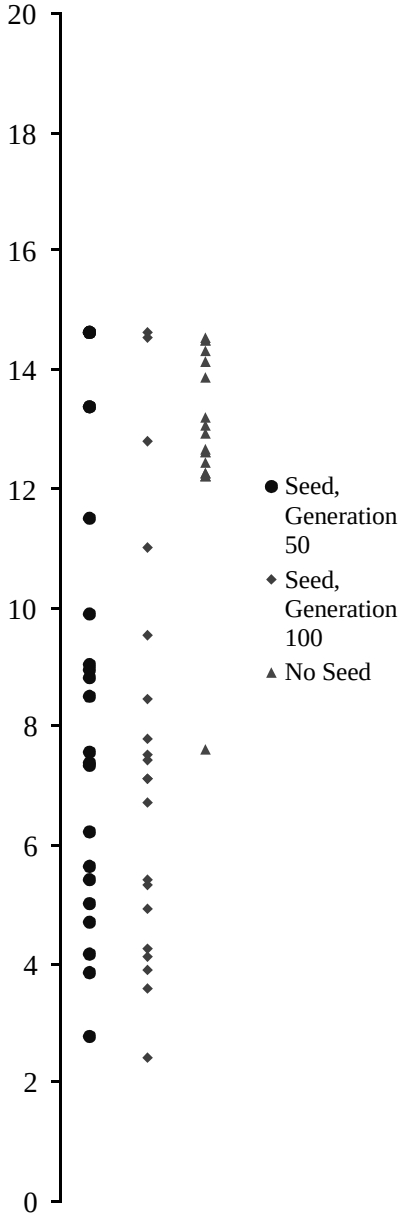


Figure 3: A scatter plot of the best-of-run individual by generation 50 and 100 from the experiment with the seed, and generation 50, the final generation, from the experiment without.

ble near the end, standing in a doorway, but not enough to affect the results greatly.

The two experiments in this chapter used the same training data, but differed in population size, number of generations and the use of a seed. The first experiment used a population size of 10,000, 51 generations and no seed individual, while the second experiment used a population size of 2,000, 100 generations and the seed. A scatter plot of the best fitness from every run is shown in Figure 3.

All performance measurements in this chapter are on the training data itself. Performance on other images, as well as performance when used to control the robot, is discussed in the chapter "Online Validation." The best constant approximation returned `image-max-y`, i.e. the very bottom pixel, for an average error per pixel, i.e. fitness, of 14.65, and achieving 113 out of 450 fitness cases within two pixels. Since the fitness measure in this chapter penalizes for the amount of error, an individual that returns a slightly higher value does worse. The seed did only slightly better at 14.12.

No Seed

On the dual 700MHz Pentium III, the time to evaluate a generation ranged from 5 seconds (1.2 CPU milliseconds/eval) to 4 hours (2.8 CPU seconds/eval), averaging 31 minutes (0.37 CPU seconds/eval), or 26 hours per run of 51 generations.

In all runs but one, the best individual in the initial, random population did no better than that. The better run did only slightly better, 14.54. This amounts to a random search for a solution through 200,000 individuals. Clearly, a random search here is harder than in the previous chapters, as expected.

These runs did universally poorly, despite the larger population size. Only one run achieved an error below 12 pixels/fitness case, achieving 7.59, close to the median score for the other experiment. This run was extended for another 50 generations and the average error of the best-of-run dropped by one pixel to 6.58. This setup is clearly too difficult for the evolutionary computation to have much success, at least with the number of runs used here.

With Seed

There are many ways in which a researcher can provide information to an evolutionary computation system, such as the elements of the representation, rules on how they must be combined, etc. A novel method is to provide an example of how the elements fit together in the form of a "seed" individual.

The seed used in these runs is the example individual in Figure 1. It was created initially simply to assure myself that nothing crucial had been left out of the representation, i.e. that it really could represent existing algorithms naturally. It was later used as an example in an earlier version of this chapter, and then implemented in order to validate the system. At this time, the two threshold values were determined using a crude GUI.

This individual was never intended to be a good solution to the problem. In fact, one of its basic assumptions, that the bottom of each column represents ground, is violated in almost a quarter of the fitness cases. As well, the image statistics were chosen for their explanatory power with no concern given to their appropriateness.

Figure 4: Performance of the seed on the training data.

Images of the performance of the seed on the training set are shown in Figure 4. It only achieved all six answers correct on the few frames where the bottom of all columns was ground, as in the upper left and upper middle. The upper right shows the most common form of error, missing a wall or door that goes to the bottom of the image. As the bottom left demonstrates, strong edges, such as those produced by the specular reflection at the bottom of the baseboards, stop the window early, whereas more gentle edges, such as those at the bottom of the filing cabinets, allow it to continue further. The last two images demonstrate greatly varying performance on consecutive frames. The middle two columns are roughly correct on the first frame, but wildly wrong on the second. Despite these flaws, overall it performed marginally better than the best constant approximation. In every run it outperformed all other individuals in the initial population.

Figure 5 graphs the best, median, average and worst fitness vs. generation for the run that attained the median best-of-run fitness. As in the last chapter, there was always one individual in every generation that achieved the worst possible score. Also, when the median fitness became roughly equal to the best for a few generations, only incremental improvements took place.

Figure 6 graphs the fitness of the best-of-generation vs. generation for the four best runs. As can be seen, the change after generation 50 was usually minor and did not affect the relative standing of the four runs. In particular, of these four runs, the best could be identified by generation 50.

The best individual from all runs achieved an average error per pixel of 2.42, getting 272 of 450 fitness cases within two pixels of the correct answer. Some example images are shown in Figure 7. It got the vast majority of the fitness cases more than close enough for obstacle avoidance, including obstacles at the bottom of the image. The upper left and upper middle images demonstrate examples. The errors were

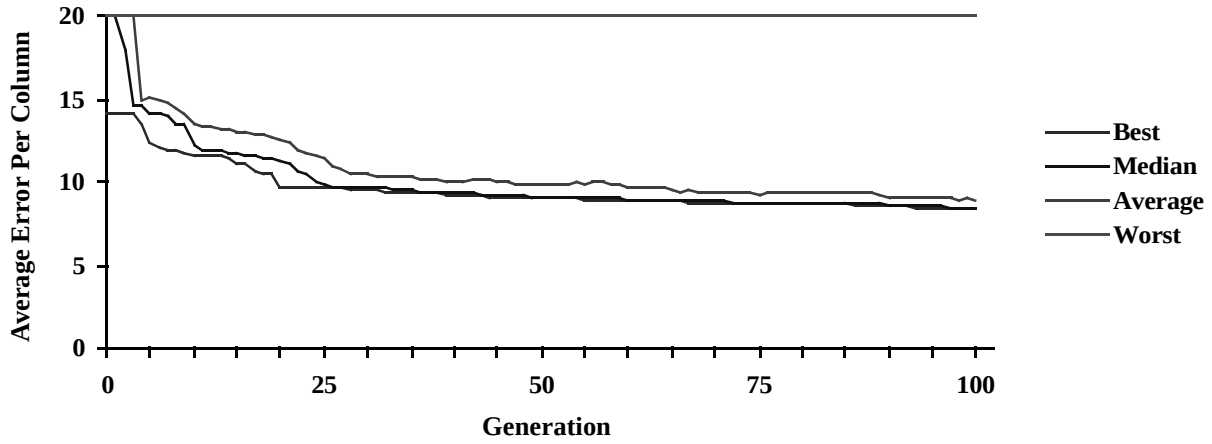


Figure 5: Average Error per Column (i.e. Fitness) vs. Generation for the run that produced the median best-of-run individual.

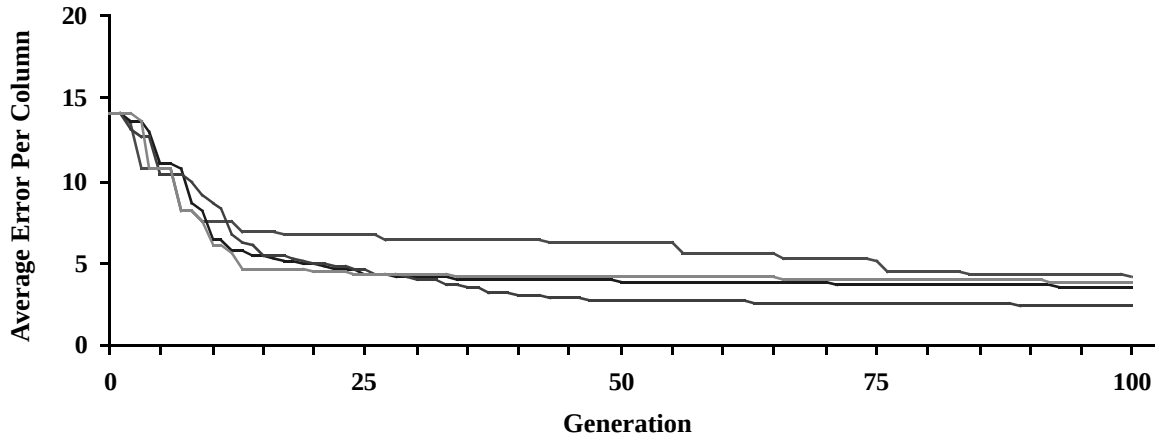


Figure 6: Average Error Per Column (i.e. Fitness) v.s Generation for the best-of-generation individual in the four runs that produced the fittest best-of-run individuals

more or less evenly distributed among remaining cases such as missing walls at the bottom of the image, missing the boundary between floor and door or filing cabinet, and thinking the floor was an obstacle. It would also occasionally miss small objects at the bottom of the image, such as the corner of a filing cabinet or a wall. However, errors were definitely the exception, and the vast majority of fitness cases were handled accurately, more than good enough for navigation, despite the large change in intensity in the carpet and other effects.

For this data set, ground truth was prepared for the non-training data using the same process, and with the same care, as for the training data. Therefore, direct comparisons of error measures are possible. The non-training data used the same images as the training set, but different column locations. On this data the simplified version of the individual achieved an average error of 2.77 pixels. Example images are shown in Figure 8.

Examining the images by hand, of the 375 fitness cases, 352 of them were close enough for navigation. Of the 23 remaining, 13 were missing a wall at the bottom of the image, five were missing a door,

Figure 7: Performance on training data of the best individual from all seeded runs.

Figure 8: Performance of the best individual from all expanded representation runs on test (non-training) data.

three were counting the ground as an obstacle, and two were missing the file cabinets.

Simplified Best Individual from all Expanded Representation Experiments

The original and machine only simplifications of the best individual are virtually unreadable. The final version, using both hand and automated techniques, is displayed in Figure 9. For verification this version was

Iterate-Vertically

An 8 by 8 window from bottom to top in the desired column. At each window location execute:

```
if first-rect then
  d := image-max-y;

  if gradient ≤ 70.71 or (desired-x - median-corner)² ≤ 239 then
    break;

  if median-corner + moravec-horizontal > 300 then
    d := y;
    break;

  if (2×median-corner + moravec-horizontal - desired-x)² ≤ 239 or
      (median-corner + 239 - desired-x + gradient)² >239 then
    c := median-corner + moravec-horizontal;
  else
    c := median-corner + 239;

  if (c - desired-x + median-corner)² > 239 and c² > 300 then
    break;

  a := gradient;
  b := - median-corner;

else   // first-rect

  if (a - gradient)² ≤ a then
    b := a;
    a := y²;
  else
    if b > 300 or
        (b > image-max-y and (gradient - b)² ≤ 5000) then
      d := y;
      break;

    b := y;
    a := gradient;
```

Figure 9: Simplified version of the best individual from all Expanded Representation runs.

implemented in the genetic programming representation and re-run on the training data to verify that the result had not changed.

The best evolved individual shows some distinct similarities to the original. It did not modify the result producing branch, which still simply returns the "d" register or the first iteration branch unmodified. Therefore, the ADFs and the other iteration branches went unused. Also, the iteration branch had the same structure as the seed, namely iterating vertically from the bottom of the image to the top, centered on the desired column. The body of the loop started the same, by first assigning y to register d, then branching based on the value of `first-rect`. It also stored the gradient magnitude in register a, at least some of the time, and used `median-corner + moravec-horizontal`, although often in a "mangled" form. However, the similarities ended there.

As with the individuals in the focused representation experiments, this individual handles the problem of recognizing objects at the bottom of the image differently than that of recognizing the transition between floor and object. To find objects such as walls and doors at the bottom of the image, it checked for a small gradient or a high frequency component that rises across the image. It then checked another high frequency measure, and if found, returned the y location of the center of the rectangle, four pixels above the bottom of the image. It is likely that this detects objects very near the bottom of the image, as opposed to objects that extend below the image. If the above tests fail, it uses a more complicated test, then finally sets registers a and b in preparation for the next iteration.

Subsequent iterations make heavy use of the registers to remember information between iterations. To understand how it worked, it helps to keep in mind that if the `break` statement is never called, the program will return `image-max-y`, indicating that the object is at the bottom of the column. There is only one `break` statement, and it can only be reached if the value of b from the previous iteration was greater than `image-max-y`. Note that this can not happen if the previous iteration took the else clause, since then b is set to y, which will

always be less than `image-max-y`. Therefore, the iteration will only stop on the iteration after the if clause is taken. It should also be noted that when "a" or "b" equal a gradient value from a previous iteration, the difference effectively takes the average for the top line of the window and subtracts it from the average from the line just below the window. In other words, it acts as a second order derivative. It should also be noted that after the bottom of the image, the decision about the location of object boundaries is based purely on gradient information.

Observations

The presence of the seed helped enormously, and extending runs for an extra fifty generations provided only incremental improvement. In future, if done at all should only be done for the most successful runs. It should also be noted that many of the individuals in the initial, random population were too big to have been created by crossover, although these always were eliminated in the first two generations.

Finally, the best individual in this chapter made some use of state. This may have been due to the presence of the seed, which already had the machinery in place, or the fitness measure, which encouraged delaying the call to `break` by an iteration or two, until the window was actually over the desired location.

With these results and observations in hand, the next chapter examines the other experiments, that simplified the representation used here to the point where a seed was not needed. In the chapter after that, the best individuals from all experiments are applied to a live video stream and used to guide the actual robot in real time.

7 Focused Representation

After reflecting on the experiments of the previous chapter, many aspects of the experimental setup were modified. The most significant were to hard code the beginning and ending locations of the window, and to dispense with the result producing branch. The representation in this chapter only allows windows that move vertically and that start and stop at the edges of the picture, unless a `break` statement is encountered. As well, the result is the value of the first memory register, eliminating the need for the result producing branch and its automatically defined functions.

The Representation of Learned Programs

Record Video Robot, Real Time

Learn Offline Genetic Algorithm

Build Navigation By Hand

Validate Online Robot, Real Time

It will be recalled from the Technical Framework chapter that each execution of an evolved program will be given an image and the x location of a column, and its single real-valued output will be interpreted as the y location of the lowest non-traversable pixel in that column. In particular, there is no state maintained from one execution to the next, and therefore all programs are purely reactive. To measure fitness, the individual is executed six times on each image, in six different columns, on sixty to seventy five images.

In evolutionary computation, the greatest influence on the results comes from the representation of the individuals. Previous chapters argued for genetic programming, and to flush in the representation, the list of all possible nodes is described and explained. The list is summarized in Table 1, and an example individual is shown in Figure 1.

For the most part they are elements from traditional programming languages, such as arithmetic operators, if statements, and reading and writing of registers. Those elements have been used in many GP applications and everything but the reading and writing of registers could be considered standard in the GP community. The only obviously application specific nodes are the iterated rectangle and image reading nodes, which are described in the next section.

A Tour of the Representation

This section presents a high level overview of the representation and explains the working of an example in detail. The next section describes the implementation of each node in detail.

The chapter “Technical Framework” describes the iterated rectangle as a building block of existing algorithms. The expanded representation experiments permitted these rectangles to move either horizontally or vertically, but it was noted that the best-of-run individuals from the successful runs only used vertical iteration, never horizontal. Therefore, in the focused representation, only vertical iteration was

supported, although the iteration could be either “going up” or “going down.”

These were represented by a pair of nodes, `Iterate-Up` and `Iterate-Down`, which took three arguments. The first argument was one of 17 tokens representing various rectangle sizes from 2 by 2 to 8 by 8. Thus, the subtree for this branch always consisted of a single terminal, and these rectangle size tokens were never used in any other part of the individual. The second argument was an expression for the *x* location of the center of the window, in pixels. The final argument was a block of code to be executed for every location of the window. Thus, if the iteration was not terminated early by hitting a `break` node, the first argument would be examined once, the second argument evaluated once, and the last argument evaluated 240 minus window height times.

This last argument could access the *image reading terminals*. There were 16 of these. Half of them were the average value over the window of one of eight common image statistics. The other half were the average of the squared value of the same statistics. This allowed the computation of second order statistics, in particular variance and standard deviation.

From the work of Teller and Veloso [1997] on PADO, the idea of memory locations which can be read from and written to were borrowed. Each iteration branch had five floating point registers, which were accessed through five write nodes and five read nodes. The write nodes each took a single real valued argument and returned the value of that argument, i.e. the value written to memory, as is the convention of assignment operators in C. The read nodes did not take any arguments.

As mentioned before, a `break` node would halt the current loop, freezing the values of the five registers. The remaining nodes were the usual suspects in genetic programming work: standard mathematics functions, control flow provided by `if-le` (if less than or equal to), `prog2` and `prog3`, random constants, and terminals to provide the image size, the area of the rectangle, the *x* location of the desired column, and the current location of the window. `If-le` took four arguments; if the first was less than or equal to the second, then the third was evaluated and returned and the fourth was not evaluated, otherwise the fourth was evaluated and returned, the third going unevaluated. `prog2` and `prog3` took two and three arguments respectively. Each argument was evaluated in turn, and the value of the last one returned.

In a modification of Koza’s “automatically defined functions” [Koza 1994], each individual consisted of two `Iterate` branches. One branch was designated the result producing branch; the value of its first register at the end of execution was used as the return value of the individual, i.e. the estimate of the location of the lowest non-ground pixel. This individual could access the final values of all memory locations of the other branch. Thus, each loop would be performed once at most.

An example individual is shown in Figure 1. This example was created by hand to demonstrate the possibilities of the representation and was not produced by any GP run. Rather than finding the lowest non-ground pixel, it find the highest patch that looks like the bottom middle of the image. While this violates the spirit of what we intend it to find, it may be better than the literal interpretation of finding lowest part of the image that doesn’t look like ground. This will be true if most objects are large enough to occlude the ground from their bottom to the horizon (in which case the two definitions are roughly equivalent) and if

the ground contains patches which look significantly different that the majority of the ground. This is the case in the Newell Simon Hall data set, where most of the carpet is flat grey, but contains a few stripes of red. In this situation, the literal interpretation may mistakenly mark the carpet as obstacle, whereas the "backwards" interpretation will get the correct answer most of the time. This would be an example of using domain constraints to help disambiguate the visual input.

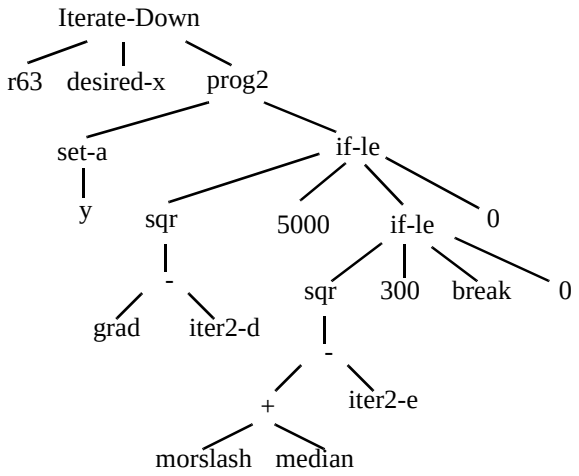


Figure 1a: The result producing branch works down from the top, quitting when a window is found that is significantly similar to the average computed in the other branch.

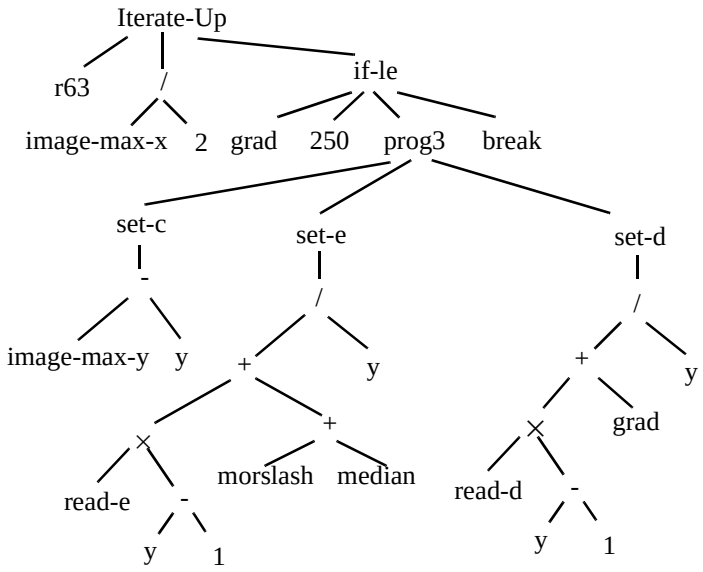


Figure 1b: Computes the average gradient and average of another statistic over a portion of the image that is very likely ground.

The second branch starts at the bottom of the image and iterates up, using a 6 by 3 window. No matter which column the individual would be judged on, it only looks at the center of the image; the result producing branch will look at the intended column. At each location, if the average gradient magnitude over the window is more than 250, the execution of this second branch stops. Otherwise the execution will continue, since there are no other `break` statements. If the bottom middle of the image always sees ground, and the ground has low gradient, this would cover a portion of the image that is most likely ground.

At each location in this swath, the following statements are computed:

$$c = \text{image-max-y} - y$$

$$d_{i+1} = \frac{d_i(c-1) + \text{grad}}{c}$$

$$e_{i+1} = \frac{e_i(c-1) + \text{morslash} + \text{median}}{c}$$

Because the window has a height of three, *y* starts out as one less than image-max-y and decreases by one each iteration. Therefore, c is the iteration number, i.e. it starts as one on the first iteration and on each iteration becomes one larger.

Locations d and e accumulate an average of two image statistics. The average of *n* numbers is sum of the numbers divided by *n*. Given the average, we can calculate the exact sum by simply multiplying by *n*. Therefore, to include one more element in the average, we simply multiply the old average by the number of items in that average, add the new value, and divide by the new number of items. Whenever the branch breaks, the two locations will contain the most up-to-date averages.

The result producing branch uses a rectangle of the same size, 6 by 3, but starts from the top of the image and moves down. On every iteration, it starts by writing the current y position to the a register; therefore, the final value of the a register, which is

the value for the function as a whole, will be the y location of the center of the rectangle on its last iteration. It then compares the average gradient over the window with the average computed by the second branch, and if they are not close, returns zero. The return value is discarded, the window is moved and the next iteration is started. However, if the current gradient *is* similar to the stored average, the second statistic is compared. If it is close as well, then the window we are looking at matches a particular description of the "average ground" and we stop.

Table 1: The Node List

Image Iteration	Iterate-Up, Iterate-Down
Window Size	2x2, 2x3, 3x2, 3x3, 2x4, 4x2, 4x4, 5x5, 2x6, 6x2, 3x6, 6x3, 6x6, 7x7, 2x8, 3x8, 8x8
Mathematical	+, -, ×, % (protected division), square, sin, cos, tan, atan, atan2, min, max
Flow Control	if-le (if-less-than-or-equal-to), prog2, prog3, break
Miscellaneous Terminals	desired-x, random constant, window-area, image-max-x, image-max-y, first-rect, x, y
Registers	set-a, set-b, set-c, set-d, set-e, read-a, read-b, read-c, read-d, read-e
Other Iterator	iter2-a, iter2-b, iter2-c, iter2-d, iter2-e
Image Operators	raw, raw-squared, median, median-squared, median-corner, median-corner-squared, gradient, gradient-squared, moravec-horiz, moravec-horiz-squared, moravec-vert, moravec-vert-squared, moravec-slash, moravec-slash-squared, moravec-backslash, moravec-backslash-squared

Details of Nodes

The list of all nodes was the same for all three experiments and is shown in Table 1. Protected division, a standard GP function, returns one if the denominator is zero, thus avoiding divide by zero. This extension also means a/a equals one even when a is zero. The other functions are already defined for all real valued inputs. As described above, `if-le` took four arguments; if the first was less than or equal to the second, then the third was evaluated and returned and the fourth was not evaluated, otherwise the fourth was evaluated and returned, the third going unevaluated. `Prog2` and `prog3` took two and three arguments respectively. Each argument was evaluated in turn, and the value of the last one returned.

`Desired-x` was the x coordinate, in pixels, of the column that the program would be judged on, that is, the column where the lowest non-ground pixel was desired. If the `random constant` node was selected during the creation of individuals, a random real number was selected uniformly between -10,000 and +10,000 and used in its place. `window-area` was the area, in pixels, of the window size, that is, the product of the two digits in the first argument to `iterate-up` or `iterate-down`. `Image-max-x` was the maximum *x* coordinate of any pixel; in this the-

sis that was always 319. Similarly, `image-max-y` was the maximum *y* coordinate, 239. `First-rect` only appeared in the third argument to the iterates, and was one when the window was at its startling location and minus one after that. `x` and `y` were the coordinates of the center of the rectangle, in pixels, rounded up.

`Set-a` through `set-e` took a single argument and set the given register to that value. `Read-a` through `read-e` had no arguments and returned the value of the given register. `Iter2-a` through `iter2-e` were only available in the result producing branch; they returned the value of the final value of the given register for the other branch.

The image operators came in two flavors, straight up and squared. The straight up versions would return the average value of the operator over the window, and the squared version would return the average of the squared value over the window. `Raw` was simply the raw image pixels, `median` was the result of replacing every pixel with the median of all pixels in a 5 by 5 window centered on it (a kind of low pass filter), `median-corner` was the raw minus the median (and therefore a kind of high pass filter), `gradient` was the magnitude of the 3 by 3 Sobel gradient operator, and the Moravec interest operators were simply the squared difference between adjacent pixels, either horizontally, vertically, or diagonally.

The random constant node actually represents a family of nodes each with a different value. In Creator, the total number of unique nodes is always 255. Each individual had two branches, each branch was limited to 1000 nodes. This gives an upper bound to the search space of $255^{2000} \approx 10^{4,813}$. While there are some restrictions, this nevertheless gives an indication of the order of the search space.

Other Details

Fitness measures in evolutionary computing have two goals. First, they should select individuals whose children, grandchildren, etc. are likely to do well. Second, because the best solution found is generally not an optimal solution to the problem, it must say how desirable each individual is as a final result.

The common approach is to consider these to be the same, by assuming that individuals who solve the problem well are more likely than others to have some component that would be useful in a full solution. However, as described in the “Early Experiments” section of the next chapter, that proved problematic and Illah Nourbakhsh suggested the solution.

The fitness measure used was rather simple. Each column of each image was a separate fitness case. The evolved program was run, and the value stored in the “a” register compared to the hand created ground truth. If the register value was within 10 pixels of the ground truth, it declared “correct.” The fitness of an individual was the number of correct answers.

This measure penalizes all answers that are more than one window’s height away from the correct answer, penalizing them equally no matter how far away, under the assumption that whatever caused them to give the wrong answer was completely off. Rewarding individuals for achieving a better answer within the 10 pixel boundary could result in selecting individuals which get one pixel closer on a number of fitness cases at the expense of getting a few more fitness cases wrong. An example of a somewhat convoluted routine which resulted in such incre-

mental improvement is discussed in the next chapter. Getting the answer correct to within ten pixels is generally more than good enough for obstacle avoidance.

During online validation for the previous experiments, it was found that the robot could not see obstacles near it, because these were below the field of view of the camera. Because the vision and obstacle avoidance algorithms are reactive, they cannot avoid obstacles outside of the field of view. Therefore, the camera was moved to the front of the robot and pitched 51° from horizontal, such that the wheels were just beyond the field of view of the camera.

Images were first dewarped using the dewarper in [Moravec 2000], an updated version of the dewarper used in the previous chapter. The `median` image operator was a simple median of all image values in a 5×5 window. The implementation of the median filter in these experiments was optimized to take much less time than a simple quicksort, but computed exactly the same result. It was validated by median filtering all images in a given data set using both methods and comparing the result.

Simplifying Evolved Programs

While it is unnecessary to know how a particular program works in order to use it, such knowledge can shed light on several questions. This includes practical questions: Are these solutions that could be implemented by hand, or are they far too subtle and complex? How will they generalize to other environments? Are there any obvious failure modes that we have not seen? Other questions arise when trying to improve the evolutionary algorithm. What features did it make use of? How did it use them? How do those differ from ways humans use them?

Unfortunately, programs created by simulated evolution are often large and convoluted. Thankfully, they also often have branches that are never executed, calculations whose results are never used, etc. Therefore, the first step to understanding evolved programs was to simplify them.

Two main methods present themselves: simplifying by hand and automatic simplification by computer. Simplifying by hand can be time consuming and error prone, although a person can often notice simplifications that would be difficult to teach a computer. However, writing a program to simplify them can also be time consuming, and may contain undiscovered bugs. The choice was even less clear because computers are better at different kinds of simplification than people are.

The path used in this work was to start simplifying by hand, and whenever a pattern was noticed that was straightforward to automate, do so. Since the programs are represented as expressions in tree form, Lisp is the natural language in which to write the simplifier. The simplification code was that in an appendix of [Koza 1992]. This code is an "engine" written in Lisp that requires rules were added specific to a particular list of nodes.

To validate the system, each new simplification rule was tried on a canonical case, and every hint of a bug tracked down. As well, a third of the interesting branch of an individual in the next chapter—approximately 300 nodes—was simplified by hand, and compared it to the automated system. Finally, the simplified individual was re-run on the

training data to verify that its score had not changed. It should be pointed out that all of these verifications were repeated many times as the simplification rules were added one by one.

The first step was to keep track of which nodes were never actually evaluated. Branches can go unevaluated when the condition of an `if-le` statement always evaluates to the same value, or as the result of a `break` statement. For example, `(if-le 0 300 x y)` will always take the `x` branch. In this case, the entire expression can simply be replaced with `x`. In general, since the first two branches may have side effects, we must keep them along with the always-evaluated branch, turning the `if-le` into a `prog3`.

In the example above, this happens because both branches of the conditional are constant, and will therefore evaluate the same way on any data set. However, we could go further and also eliminate branches that are never executed when run on the training data set. For the purposes of the training data the `prog3` and `if-le` forms are the same. Which generalizes better? The question could also be asked of existing `prog3`s where the first branch always evaluates less than or equal to the second. In that case it seems arbitrary to convert it into an `if-le` and add a clause that does not affect the learning.

Table 2: Simplification Rules

Rule	Example
Rules that Eliminate Computation	
Remove branches that were never executed on training data	`(if-le a b x y)` -> `(prog3 a b y)` if `a` is always less than or equal to `b` when run on the training data.
Remove initial branches of `progn` that do not have side effects.	`(prog3 x y z)` -> `(prog2 x z)` if `y` does not have any side effects.
Eliminate nested calls to `set`	`(set-a (set-a y))` -> `(set-a y)`
Eliminate a call to `set` if the value will be overwritten before it is ever read	`(set-e (+ (set-e x) y))` -> `(set-e (+ x y))` if neither x nor y calls break, and if y does not contain a `read-e`.
Eliminate a call to `set` in the result producing branch if that register is never read.	`(set-e x)` -> `x` if no `read-e` ever appears. Only applies to the result producing branch.
Rules for Human Readability	
Convert two successive instances of `prog2` into `prog3`	`(prog2 (prog2 x y) z)` -> `(prog3 x y z)`
Raise `prog2` if it is the second branch to an arithmetic operator, if the reordered branches do not contain `break` and do not read anything the other one sets.	`(+ x (prog2 y z))` -> `(prog2 y (+ x z))` if x does not read any register that y sets and vice versa, and if `break` does not appear in either x or y.
If the argument to a `set` is an `if-le`, push the `set` into the last two arguments of `if-le`.	`(set-b (if-le a b x y))` -> `(if-le a b (set-b x) (set-b y))`

In other words, the genetic programming does not know the difference between `prog3` and `if-le`, except in how they influence evaluation. An `if-le` with a branch that is never executed, and whose three remaining branches are always executed in order, is equivalent to a `prog3` as far as the GP is concerned. The never-executed branch does not affect fitness in any way, so we have no inductive reason to suspect it would do anything worthwhile on new data. Treating it as a `prog3` is more natural.

Therefore, the first step in editing the evolved programs was to run the program on training data, keeping track of which subtrees were never executed in practice, then eliminate those subtrees. That was done in C, but the rest of the simplifications were done in LISP. The set of simplifying rules is described in Table 2.

The rules were carefully chosen not to produce any loops, where the effects of one rule could be undone by some combination of other rules. Therefore, the rules could be continuously applied until no more could be applied anywhere in the tree. The order of application of the rules could affect the outcome, since the application of some rules render others inapplicable. This could be remedied by suitably generalizing some of the above rules, but in practice was not a problem.

As a side note, it is possible to apply these simplifications during evolution itself. The value of this is a topic of debate in the genetic programming community, and was not tried here.

Finally, the individual was rewritten into a syntax similar to a traditional procedural language and simplified further, for example noting that a variable does not affect the results so that all references to it can be removed, or that the code might usefully be reorganized in some way. Any further simplifications were converted back into Lisp form and verified on the training set. Often, a hand simplification would allow more automated rules to apply, so hand simplifying and automated simplifying were alternated.

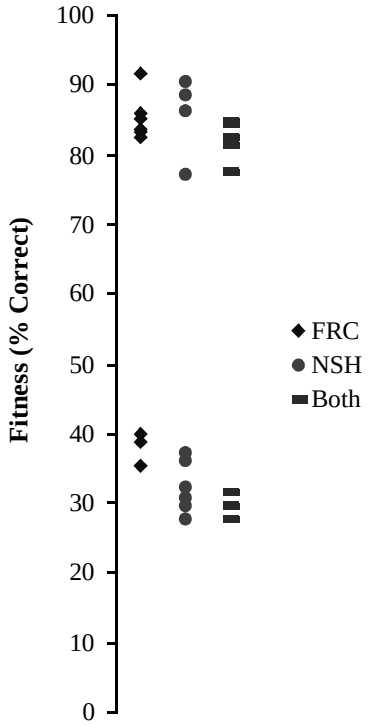


Figure 2: A scatter plot showing the fitness, i.e. the percentage of fitness cases estimated correctly, of the best-of-run individual in all three experiments.

Results

The three experiments in this chapter differed only in the training data used. The first used the data from the Field Robotics Center (FRC), the second from nearby Newell Simon Hall (NSH), and the last used every other frame from both of the first two. A scatter plot of the best fitness achieved from every run is shown in Figure 2. The most notable feature is the bimodal distribution of the first two runs.

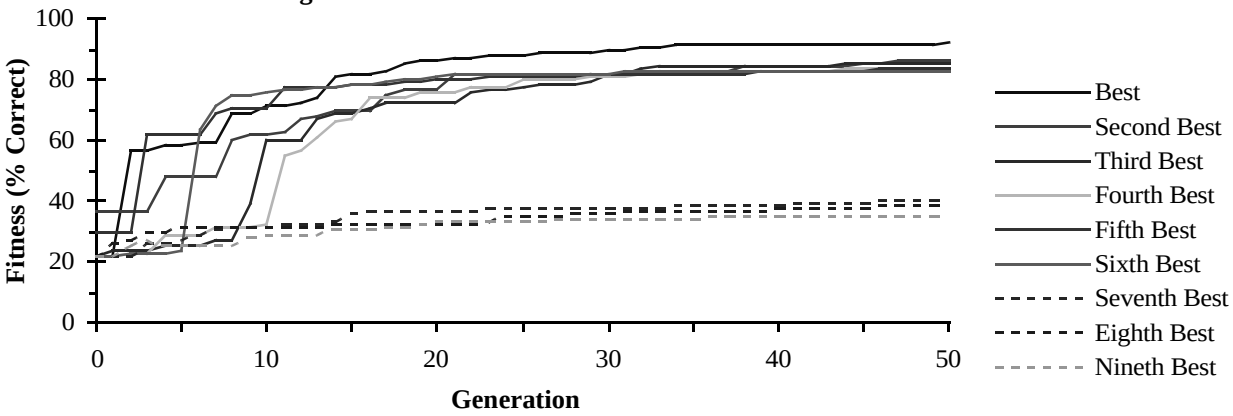
All performance measurements in this chapter are on the training data itself. Performance on other images, as well as performance when used to control the robot, is discussed in the chapter "Online Validation."

Field Robotics Center

The FRC runs are examined first. The summary of the training data at the end of the Data Collection chapter is repeated here:

Camera Angle:	51 degrees from horizontal
Total Number of Frames:	356
Frames In Training Set:	71 (every fifth)
Number of Fitness Cases:	71 × 6 = 426
Elapsed Time:	82 seconds
Frame Rate Of Training Set:	71 ÷ 82 = 0.87 fps

Figure 3: Fitness of Best vs. Generation for FRC Runs



Starts with robot in lab doorway. Moves straight until its in the hallway, then turns right, travels down hallway, at end turns right, then travels straight to dead end. All doors were closed. The fluorescent light bulbs at the start are burned out, so the intensity of the carpet varies widely. The shadow of the robot is visible at the bottom of most frames.

A random search was implemented by simply generating random individuals using the same distribution as the initial generation. Of the 2,042,079 individuals created this way, only 56 (0.0027%) did better than the best constant approximation. The best individual achieved 246 out of 426 fitness cases, or 57.7%, within 10 pixels of correct.

A graph of the fitness of the best-of-generation vs. generation for all FRC runs is shown in Figure 3. On the dual 700MHz Pentium III, the time to evaluate a generation varied from 5 minutes (0.15 CPU seconds/eval) to 77 minutes (2.3 CPU seconds/eval), averaging 35 minutes (1.04 CPU seconds/eval), or 30 hours per run of 51 generations.

The best constant approximation, ignoring both the image and the desired column, was to return `image-max-y - 10`, i.e. 228, to get 101 answers correct, for a fitness of 23.7%. Because of the camera's wide field of view, it often imaged walls on the left or right, which made for a preponderance of answers at the very bottom of the image. Since there was no penalty for returning an answer up to 10 pixels off, the constant approximation got all columns that were within 20 pixels of the bottom.

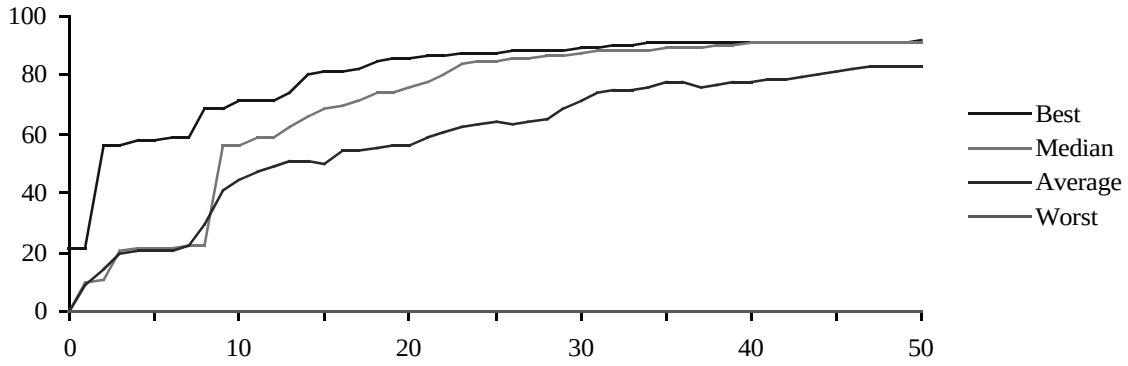
In all but two of the runs, the best individual in the initial population did worse than that, returning `image-max-y - 3` and achieving 92 correct answers for a fitness of 21.6%. The other two runs achieved 29.3% and 36.4% in the initial population.

Interestingly, the graph shows that if an individual was going to end up in the higher scoring mode, the fitness of the best-of-generation would already be there by generation 11.

The best individual from all runs succeeded in 391 out of 426 fitness cases, for a fitness of 91.78%. Some example images are shown in Figure 4. In the upper left and upper middle images, all six fitness cases were determined correctly. In the upper right, one of the two on the filing cabinets was higher than desired, however, a difference this small would not affect obstacle avoidance. The other three each contain one more significant error. MPEGs of the best individual from all three experiments, run on both the training data and other data as described below, can be found at http://www.metahuman.org/Martin/Dissertation/.

Figure 4: Performance on the training data of the best individual from the FRC hallway runs.

Figure 5: Best, Worst, Median and Average Fitness vs. Generation for the Best FRC Run



This individual can distinguish carpet from wall or door at the bottom of the image, and find the boundary between ground and non-ground, despite a burned out fluorescent light at the beginning of the run, carpet intensities that vary from black to at least the 140s out of 255, the shadow of the robot and large gradient intensities on the carpet due to luma coring.

Figure 5 graphs the fitness of the best, median, average and worst of each generation in the run that produced the best individual. A few trends are apparent. First, significant innovation happened only while the median was significantly different than the best. Once half the population had substantially similar performance, significant change stopped, suggesting that the population had largely converged and diversity was lost. The average kept improving however, suggesting that the median is a better measure of convergence than the average.

Only a subset of collected images were used in the training data, which left others to be used as test data, to asses over fitting; see Figure

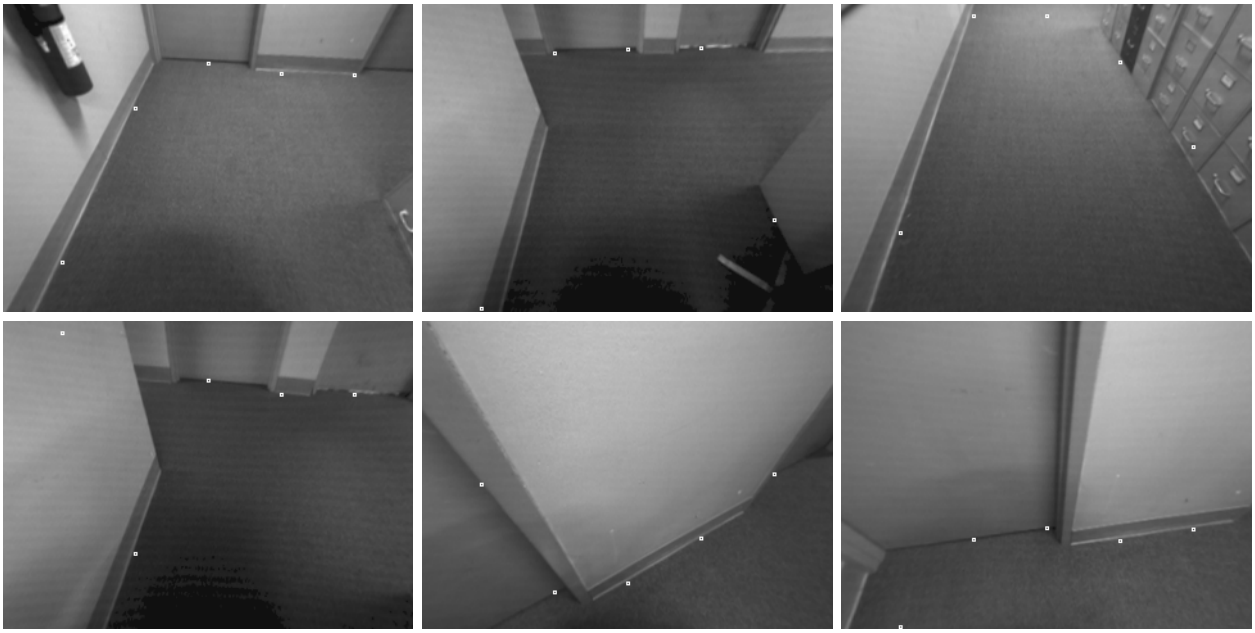


Figure 6: Performance on non-training data of the best individual from the FRC hallway runs.

6. As well, the locations of the columns were changed to in between the ones used in training. The program used during this validation was a simplified version, using the simplification techniques described above. The simplified program returns exactly the same answers on the training data, but may perform slightly differently on non-training data. Validation on an entirely new, live video stream is discussed in the chapter "Online Validation."

The performance on the test (non-training) data was judged using a weaker criterion than that used during the genetic programming, namely the author's opinion of what error was significant for obstacle avoidance. To give a conservative estimate, any questionable result was considered an error. Thus, an value was considered correct whenever it would very clearly not affect obstacle avoidance, in the author's opinion. On fifty images of non-training data, five columns per image, the best individual achieved 239 out of 250 columns correct, or 96.8%. The eight errors were: missing the coat stand, missing a wall, thinking the ground was an obstacle (x2), and missing a door (x4).

Simplified Best Individual from FRC Experiment

The final hand and machine simplified version of the best FRC individual is displayed in Figure 7.

In the first nine iterations, when y is less than or equal to nine, the return value is set to nine. This ensures that if no other edge is found, nine will be returned, which works for all values at the top of the image.

If the final value of b in the second branch is greater than nine, then "a" will be set on every iteration. Therefore, the final value of "a" will represent the bottom of the image. Since the other conditions require a large gradient, this case must catch walls, doors, and other objects that extend to the bottom of the image. In the training set, these were all a solid colour, and therefore had a low gradient.

Result Producing Branch:

Iterate-Down, 3x3 window, centred on the desired column:

```
if y ≤ 9 then
  a := y;

if iter2-b > 9 then
  a := y;

if median > 35.4444 then
{
  if gradient > 413.96 then
    a := y;

  if gradient > 239 and
  (gradient / 239)^4 > moravec-slash then
    a := y;
}
```

Second Branch:

b (initial value) := 3.40282e+38

Iterate-Up, 3x3 window, in column desired-x:

$$b = \frac{b(1+h) - (1+h+h^2+h^4)\text{desired-x}}{h^5}$$

Where *h* is moravec-horizontal

Figure 7: A simplified equivalent of the best individual from all FRC runs.

If neither of the other conditions are met, the median image value is checked. If it is too low, the other conditions are ignored. It will be recalled that luma coring was on during the data collection run, and that luma coring converts any intensity value less than 32 to 16. This causes large gradients at the boundaries of such regions, which are almost always on the carpet. Thus, it is reasonable to assume this check avoids those problems.

Finally, the gradient is checked. If it is greater than 413.96 a non-floor pixel is noted; in other words, this code embodies the assumption that, in areas of the image with a median pixel value of 36 or higher, the carpet does not have a strong gradient. Similarly, if the gradient is too low, the evidence for non-carpet is considered inconclusive at best. If the gradient is between these two bounds, the non-linear condition is checked.

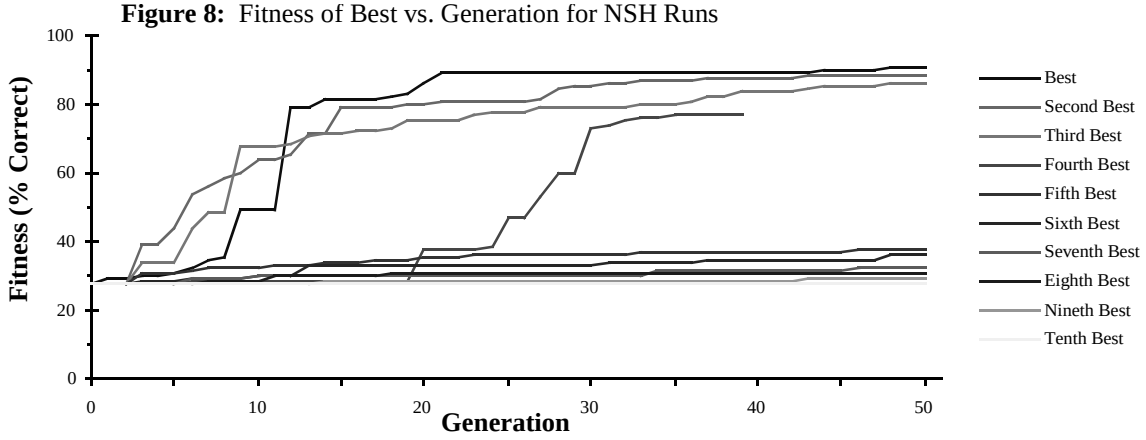
Based on the role that the second branch plays in the result producing branch, we can conjecture that it detects walls and doors. Given that it only looks at horizontal differences in the image, we can conjecture that it detects areas with little texture. If `moravec-horizontal` is ever zero, a divide by zero is called for, and by the rules of protected division, the result will be 1. Since b's initial value is so high, we might also conjecture that this formula simply detects whether the operator ever becomes zero. Whether or not these are true are best answered, not by simplification, but using other techniques suggested in the Discussion chapter.

A few other features deserve comment. The two branches represent two very different styles of algorithm. The result producing branch is essentially a decision tree, although one of the decisions is a non-linear boundary in the space spanned by two image operators. The other branch, that plays the role of detecting a certain kind of feature, is a recurrent mathematical function involving both a single image operator and the location of the column.

This division into two cases—gradient based boundaries and objects that touch the floor—was discovered automatically. Nothing in the representation suggested the two different cases, nor that the two branches should each tackle a separate case. This is an example of the genetic algorithm simultaneously exploiting regularities in both the problem domain and the representation.

It should also be noted that in the automatically simplified version, no random constants appeared. Instead, when numerical constants were needed, the area of the window (9) and the image dimensions (319, 239) were used, often in combination with mathematical functions. This suggests that the range of random constants was too large (-10,000 to 10,000); numbers in the hundreds might work better. Also, the trigonometric functions went unused, suggesting that they are simply distractions. The column of iteration was always desired-x, suggesting that this is best hard coded. If borne out by examining other runs, these conjectures could be tested in a controlled experiment.

Figure 8: Fitness of Best vs. Generation for NSH Runs



Newell Simon Hall

The summary of the training data at the end of the Data Collection chapter is repeated here:

Camera Angle:	51 degrees from horizontal
Total Number of Frames:	328
Frames In Training Set:	65 (every fifth)
Number of Fitness Cases:	65 × 6 = 390
Elapsed Time:	75 seconds
Frame Rate Of Training Set:	65 ÷ 75 = 0.87 fps

While the robot had to travel mostly straight down a hallway, it started out a little askew, so it approached one side. At one point, a person walks past the robot and is clearly visible for many frames. At the end of the hallway it turns right. The carpet is grey with a large black stripe.at one point. The shadow of the robot is visible at the bottom of most frames.

A random search was implemented by simply generating random individuals using the same distribution as the initial generation. Of the 2,000,964 individuals created this way, only 68 (0.0034%) did better than the best constant approximation. The best individual achieved 225 out of 390 fitness cases, or 57.7%, within 10 pixels of correct.

A graph of the fitness of the best-of-generation vs. generation for all NSH runs is shown in Figure 8. On the dual 700MHz Pentium III, the time to evaluate a generation varied from 3.5 minutes (0.11 CPU seconds/eval) to over 10 hours (19 CPU seconds/eval), averaging 43 minutes (1.28 CPU seconds/eval), or 36 hours per run of 51 generations.

The best constant approximation, ignoring both the image and the desired column, is to return 10, which achieves 109 correct answers, for a fitness of 27.95%. This hallway was wider than the FRC hallway, so the best bottom-of-image score was 28 hits or 7.18%. Since there is no penalty for returning an answer up to 10 pixels off, the constant approximation gets all columns that are within 20 pixels of the top.

In all runs, the best individual in the initial population did as bad or worse than that. Because the computer kept performing runs until halted by human intervention, an eleventh run was started, which happened to achieve a fitness of 32.56% in the first generation.

One run broke the pattern of achieving over 50% early on if it was ever going to achieve it. The individuals in this run took an inordinate

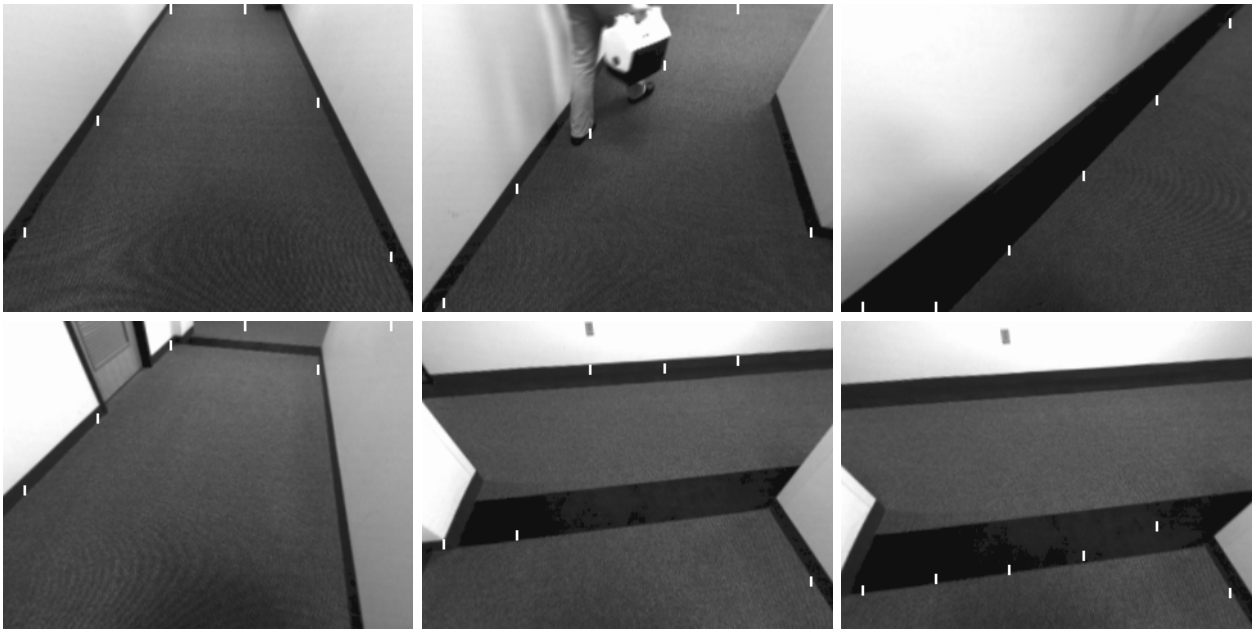


Figure 9: Performance on the training data of the best individual from the NSH hallway runs.

amount of time to evaluate, 19 seconds each or over 10 hours per generation, 15 times the average. For this reason, the run was halted with 11 generations left, which at that rate would have taken almost five more days to finish. Nevertheless, this run will be completed by the defense. As can be seen from the graph, it was performing more poorly than the other three runs in the higher scoring group.

The best individual from all runs succeeded in 353 fitness cases, for a score of 90.51%. Some example images are shown in Figure 9. In the images in the top row, all six fitness cases were determined correctly. In the lower left, a wall is mistakenly identified as ground although the black carpet is correctly labeled as ground. The majority of errors were incorrectly classifying the black carpet as non-ground. When the black patch was far away (and brighter) it was generally classified correctly, although the second column from the left classified it correctly only at extreme distances. At closer distances, where the intensity fell to less than 32 and was therefore changed to 16 by the luma coring, it was reliably and rather uniformly misclassified. This is more than enough to foul obstacle avoidance code.

Again, the best evolved individual succeeded in over 90% of cases, correctly classifying people, doors and baseboards with enough fidelity for obstacle avoidance, despite the shadow of the robot and moiré patterns in the images. Misclassifying the black patch was the only problem that would significantly impact obstacle avoidance.

It was hoped that the best-of-run individuals from other high scoring runs might have got the black carpet right while encountering less significant problems in other areas. However, that was not the case; whereas the individual described above worked when the black patch was far away, the other individuals consistently misclassified it at all distances.

Figure 10: Best, Median, Average and Worst of Generation vs. Generation for the Best NSH Run

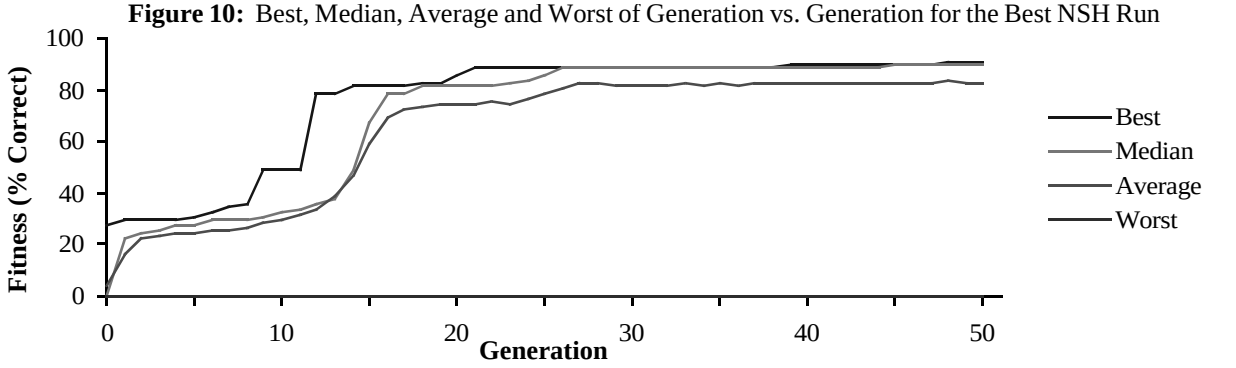


Figure 11: Performance on the test (non-training) data of the best individual from the NSH hallway runs.

Figure 10 graphs the fitness of the best, median, average and worst of each generation in the run that produced the best individual. A few trends are apparent. First, the median equaling the best again appears to be a good indicator of convergence. Again, the average kept improving, suggesting that the median is a better measure of convergence than the average.

The performance on test (unused training) data, using different columns, is shown in Figure 11. The program used during this validation was again simplified in a way that may perform slightly differently on non-training data. On forty five images of non-training data, five columns per image, the best individual achieved 215 out of 230 columns correct, or 93.5%. The criterion here was more forgiving than that used during evolution, counting only errors relevant for obstacle avoidance.

Of the fifteen errors, twelve were classifying the black carpet as non-ground, two were classifying the carpet at the bottom of the image as non-ground, and one was classifying a door as ground. There were also a number of examples of classifying the baseboards as ground but

finding the wall above them, and these were considered not significant for obstacle avoidance.

Simplified Best Individual from NSH Experiments

The final simplified version of the best NSH individual is shown in Figure 12.

NSH Best, Result Producing Branch:
Iterate-Up, 2 by 8 window, centered on the desired column:

```
if first-rect then
{
  a := y;

  if tan(moravec-horizontal) ≤ 1 + y / desired-x and
       (median-squared - y) × median-corner-squared ≤
        moravec-horizontal-squared then
    break;
}
else
{
  if gradient-squared × median-squared > 10,000,000,000 then
    break;

  a := y;

  if (median-squared - y) × median-corner-squared ≤ y then
    break;

  if gradient-squared > 4656.07 then
    temp := 2;
  else
    temp := 1 + y / morslash × gradient-squared² /
      (gradient-squared/median-squared - desired-x + image-max-y - y);

  if tan(moravec-horizontal) ≤ temp then
  {
    if (median-squared - y) × median-corner-squared ≤
         moravec-horizontal-squared then
      break;

    if (median-squared - desired-x + image-max-y - y)
         × median-corner-squared ≤ moravec-horizontal-squared then
      break;

    if (median-squared-y) × median-corner-squared ≤ 239 then
      break;
  }
}
```

Second Branch:
Not used.

Figure 12: Simplified version of the best individual from the NSH runs.

The individual uses the approach of starting at the bottom of the image, setting the "a" register to the current *y* position, then breaking wherever it detects the ground/non-ground boundary.

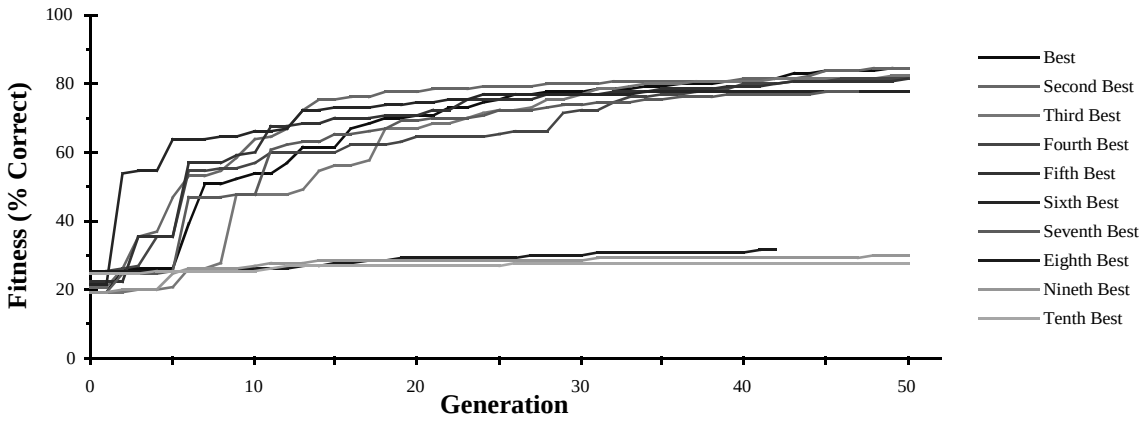
At the first location of the window, i.e. the bottom of the image, *y* will always equal four. The return value is tentatively set to four, and the given condition is evaluated to determine whether the bottom of the image represents ground or non-ground. Since walls and baseboards were generally caught at the very bottom of the image, this test must be the one that decides.

In subsequent iterations, if the geometric mean of the average of the gradient magnitude squared and the average median pixel value squared is greater than 100,000, the *y* location from the previous location is returned. Otherwise, the return value is updated to the current y location, and series of tests are performed to decide whether or not the window is over a ground/non-ground boundary.

In common with the best individual from the FRC hallway runs, this individual effectively performed different computations at the bottom of the image than on the rest of it. However, the test at the bottom is very similar to another test in the main branch. Before simplification the original individual did not have "if first-rect" as the first statement, but instead used the same code for both cases, with only a few values changed by "if first-rect" conditions. In contrast, the FRC individual used a completely different branch for the bottom test.

This individual does not maintain any state from one window location to the next, except for the

Figure 13: Fitness of Best vs. Generation for Combined Runs



previous location of the window which could easily be computed. Looking at only the pixel values without maintaining such state, it is most likely impossible to distinguish baseboards from black carpet. Even people, if only shown a 2 by 8 pixel window, would find it impossible to distinguish between them. What's more, the poor black-carpet-finding performance of the column second from the left may be related to one or the other of the desired-x tokens.

Given this lack of state, the approach of working up from the bottom and breaking at the first potential ground/non-ground transition is equivalent to the best individual from the FRC runs, which would work down from the top, setting the "a" register at every potential ground/non-ground transition.

This individual is still similar to a decision tree, but uses many more image statistics in many more non-linear combinations. It also uses the *x* and *y* locations of the window in these decisions.

Combined

The last experiment used every other training image from both of the previous data sets. The summary of the training data that appears at the end of the Data Collection chapter is repeated here:

Camera Angle:	51 degrees from horizontal
Total Number of Frames:	328 + 356 = 684
Frames In Training Set:	68 (every tenth)
Number of Fitness Cases:	68 × 6 = 408
Elapsed Time:	75 + 82 = 157 seconds
Frame Rate Of Training Set:	68 ÷ 157 = 0.43 fps

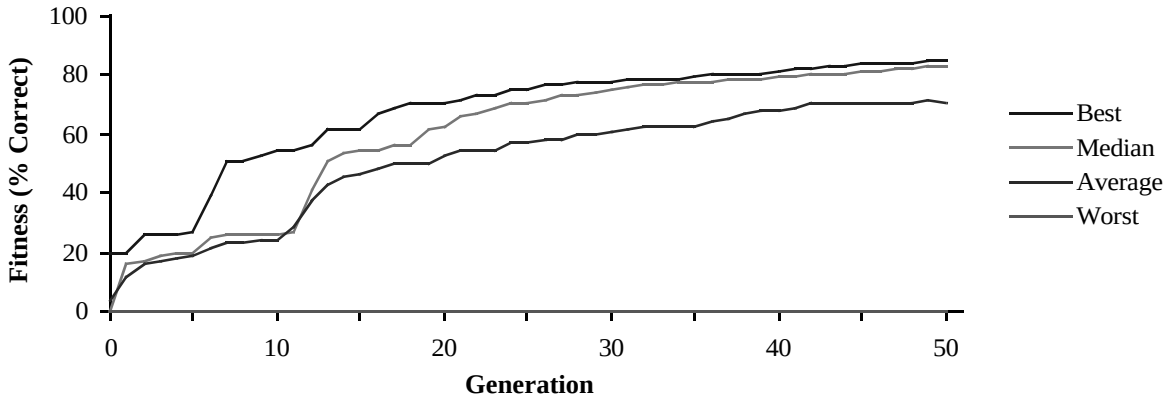
This data set was simply the combination of the above two data sets, using every tenth frame instead of every fifth in order to keep the training set size approximately equal.

A random search was implemented by simply generating random individuals using the same distribution as the initial generation. Of the 2,009,923 individuals created this way, only 345 (0.0172%) did better than the best constant approximation. The best individual achieved 236 out of 408 fitness cases, or 57.8%, within 10 pixels of correct.

A graph of the fitness of the best-of-generation vs. generation for all runs in this experiment is shown in Figure 13. On the dual 700 MHz

Figure 14: Performance on training data of the best individual from the combined data set runs.

Figure 15: Best, Median, Average and Worst Fitness vs. Generation for the Best Combined Run



Pentium III, the time to evaluate a generation varied from 50 seconds (0.025 CPU seconds/eval) to over 2 hours (4.3 CPU seconds/eval), averaging 30 minutes (0.928 CPU seconds/eval), or 26 hours per run of 51 generations.

The best constant approximation, ignoring both the image and the desired column, is to return 10, to get 79 out of 408 fitness cases correct, for a fitness of 19.36%. Of the ten initial populations, for of them achieved this fitness; the others achieved higher. The run that eventually produced the best-of-experiment individual was one of the four, that is, at the initial, random population it was tied for last place. The run that eventually came in second was the lowest scoring of the other six, with a score of 20.83%. The best individual from all initial populations 25.74%. Once again there was a bimodal distribution, and once again the fate of a run was sealed early on, this time by generation 9.

Best Combined, Second Branch:
Iterate-Up, 5 by 5 window in the desired column:

$$\text{let hybrid1} = \frac{\text{median-corner-squared}}{319\,\text{gradient-squared}};$$

$$\text{let hybrid2} = 1 + \frac{\text{moravec-backslash}}{\text{median-corner-squared}};$$

```
if moravec-slash ≤ y then
{
  if moravec-slash > b then
    e := image-max-y;
    break;

  e := y;

  if moravec-slash ≤ median-corner then
    break;

  b :=
```

$$\frac{y}{\left(\frac{\text{med-cor-s}}{319} + \frac{\text{mor-bkslh}}{\text{raw}}\right)\left(1 + \frac{\text{mor-bkslh}\left(\text{med-cor-s} + \frac{\text{mor-bkslh}}{\text{raw}}\right)\text{hybrid2}}{319\,\text{gradient-squared}}\right)};$$

```
  if moravec-slash > b
    break;

  if y ≤ 319 moravec-backslash then
  {
    b :=
```

$$\frac{y}{\left(\text{hybrid1} + \frac{\text{moravec-backslash}}{\text{raw}}\right)\text{hybrid2}};$$

```
    if moravec-slash ≤ moravec-backslash/b then
      break;

    b :=
```

$$\frac{y}{\left(\text{hybrid1} + \frac{y}{\text{raw}}\right)\text{hybrid2}};$$

```
    if moravec-slash ≤ moravec-backslash/b then
      break;

    temp :=
```

$$\frac{y}{\text{hybrid1}\left(1 + \frac{\text{mor-bkslsh}}{y}\left(\frac{\text{med-cor-s}}{\text{mor-bkslsh}} + \frac{\text{mor-bkslsh}}{\text{raw}}\right)\text{hybrid2}\right)};$$

```
  } else {
    temp := median-corner-squared;
  }
```

(continued on next page)

The best individual from all runs succeeded in 346 out of 408 fitness cases, for a fitness of 84.8%. Some example images are shown in Figure 14. In the upper left and upper middle images, all six fitness cases were determined correctly. Errors were concentrated almost entirely on the first few images and the on misclassifying the black carpet. In the upper right, the second image of the training set shows errors on very dark carpet, as well as misclassifying walls that go to the bottom of the image. The lower right image shows a misclassification of dark carpet, in this case the robot's own shadow, as obstacle. Other errors that would be significant for obstacle avoidance are rare. The filing cabinets, for example, were always classified more than well enough for obstacle avoidance. The lower left shows a misclassified carpet, and the lower middle a missed door.

Outside of the two significant types of error, the individual performs very well. The small errors appear transient and would not affect navigation in either environment. In the vast majority of cases it correctly classifies people, doors and walls despite the combination of hurdles in the previous two data sets: a burned out fluorescent light on part of the run, carpet intensities that vary from black to at least the 140s out of 255, the shadow of the robot and moiré patterns of noise in the images.

Figure 15 graphs the best, median, average and worst of each generation of the run that produced the best-of-experiment individual. The theory that there is no improvement after the median equals the best is supported, if only because the condition never happens and there is continuous improvement. Again, there seemed to be no clear relationship between the average individual and future performance.

The simplified version of this individual was also run on non-training data, i.e. the frames in between those that were collected, and on columns between those used during evolution. This was again judged using the author's opinion of

(continued from previous page)

```
  if y > 319 × temp then {
    b := median-corner-squared;
  } else {
    b := y / ((hybrid1 + moravec-backslash / raw) hybrid2);
    if moravec-slash ≤ moravec-backslash then {
      temp-b := b;
      b := y / ((hybrid1 + hybrid2 / raw) hybrid2);
      if moravec-slash ≤ moravec-backslash / b then
        break;

      b := y / (hybrid1 (1 + mor-bkslsh / temp-b ((med-cor-s / mor-bkslsh) + (mor-bkslsh / raw)) hybrid2));
    } else {
      b := y / (hybrid1 × hybrid2);
    }
  }

} else {
  e := y;
  b := y / (((median-corner-squared / 319) + (hybrid2 / raw)) hybrid2);
  if moravec-slash > b then
    break;

  if moravec-slash ≤ moravec-backslash / median-corner-squared then
    break;

  if moravec-slash ≤ moravec-backslash then
    e := median-corner-squared;
    b := y / (hybrid1 (1 + mor-bkslsh / med-cor-s ((med-cor-s / mor-bkslsh) + (mor-bkslsh / raw)) hybrid2));
  else
    b := y / (hybrid1 × hybrid2);
}
```

Figure 16: Simplified version of the best individual from the combined data set experiment.

what error was significant for obstacle avoidance.

On forty eight images of non-training data, five columns per image, the best individual achieved 208 out of 240 columns correct, or 86.7%. The success rate on the FRC subset was 92%, and on the NSH subset 81%. On the FRC subset there were 7 conservative errors, i.e. labeling the ground as non-ground, two where it missed a wall, and one where it missed a filing cabinet. On the NSH subset there were 22 conservative errors, and all but 2 of them were mistaking the black carpet for an obstacle. The black carpet was labeled as ground every time.

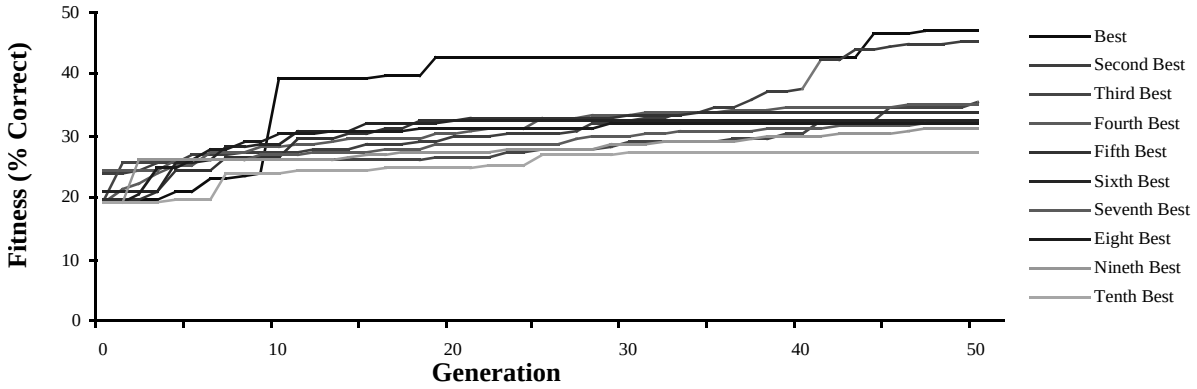
Simplified Best Individual from Combined Experiment

The result producing branch simply returns the *e* register of the second branch and isn't shown here. The final hand and machine simplified version of the second branch is displayed in Figure 16.

Two expressions occurred multiple times, denoted as *hybrid1* and *hybrid2* in the figure. They were image statistics combined with constants, and did not otherwise depend on the location or size of the window, etc. Therefore, they can be viewed as genetically discovered image statistics. They were never used on their own, always as a part of a larger mathematical expression.

This individual used the technique of writing the *y* location of the center of the current window in the result register during every image, then breaking when it determined that there was a transition between ground and obstacle. The only exception is writing *image-max-y*, i.e. the bottom of the image, at one point, and writing *median-corner-squared*. Because there is no break statement after *median-corner-squared* is written, the value would be overwritten by the *y* register on the next iteration. The only way the final result of the branch could be affected would be if this statement were executed on the final iteration. It is not known whether that happened on the training data.

Figure 17: Fitness vs. Generation for All Combined Runs With Depth Limiting



This individual uses the *b* register to maintain state from one window location to the next. The value from a previous iteration is only ever compared to the *moravec-slash* image statistic.

In this simplified form it is again organized as a decision tree, however the decisions are highly non-linear, in some cases involving as many as five image statistics along with the current *y* location of the window.

Unlike the best-of-experiment individuals in the two previous experiments, here it is not clear which branches detect objects at the bottom of the image, and which detect the transition between ground and non-ground.

As was the case with the-best-of-experiment individual from the FRC runs, in this simplified version no random constants appeared. Instead, when numerical constants were needed, the image height (319) and expressions that evaluate to constants, such as a number minus itself, were used. This again suggests that the range of random constants was too large (-10,000 to 10,000); numbers in the hundreds might work better. Also, the trigonometric functions again went unused, suggesting that they are simply distractions. The column of iteration was always desired-x, suggesting that this is best hard coded. If borne out by examining other runs, these conjectures could be tested in a controlled experiment.

Combined with Depth Limited Crossover

A final experiment used the combined data set but a different method of limiting the size of individuals produced by crossover. Instead of the total size of each branch being the criterion, the maximum depth was limited to 17. That is, if the depth of an offspring was greater than 17, it was discarded and replaced with a copy of a parent. The maximum size of trees (both iteration branches combined) seen in each generation was smaller, 623 vs. 984 for the NSH runs and 912 for the FRC runs. That is the average of each generation's maximum, over all generations of all runs. Because this experiment was not part of the original experimental design (it was run by accident), it is not referred to in the rest of the dissertation.

A graph of the fitness of the best-of-generation vs. generation for all runs in this experiment is shown in Figure 17. On the dual 700 MHz Pentium III, the time to evaluate a generation varied from 3 minutes (0.083 CPU seconds/eval) to 103 minutes (3.1 CPU seconds/eval), aver-

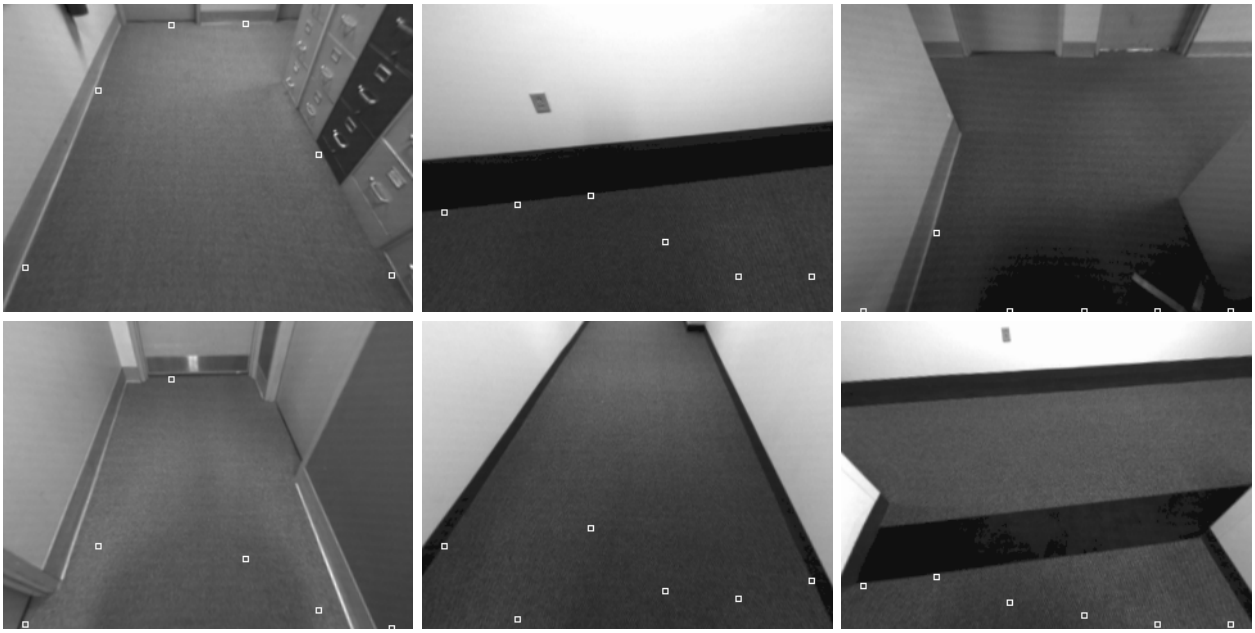


Figure 18: Performance on training data of the best individual from the combined data set runs with depth limiting.

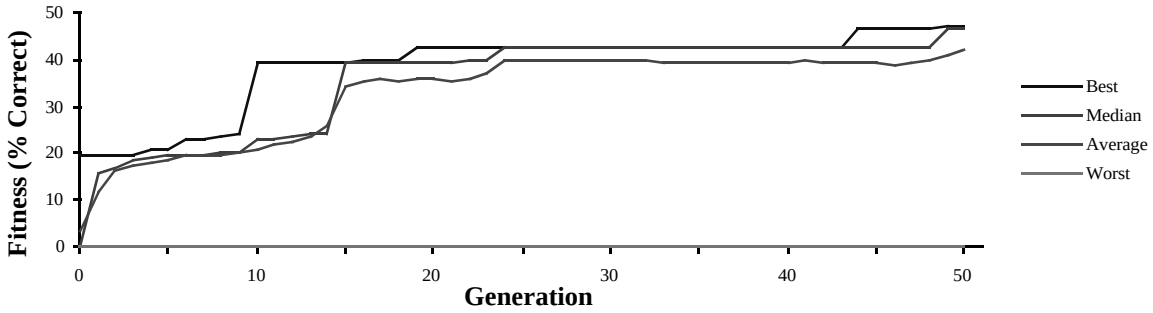
aging 28 minutes (0.84 CPU seconds/eval), or 24 hours per run of 51 generations.

The best constant approximation, ignoring both the image and the desired column, is to return 10, to get 79 out of 408 fitness cases correct, for a fitness of 19.36%. In four of the ten runs, the best individual in the initial population got exactly that fitness. In three other runs, the best initial individual achieved 19.85% or 81 fitness cases correct. The remaining three runs achieved 21.08%, 24.02% and 24.26% in their initial random populations. Thus, in a random search of 40,000 individuals, the best score achieved was 24.26%. This was worse than the best-of-run individual from every single run.

The evidence for a bimodal distribution is less strong, since the gap between the potential groups is much smaller, there were only two runs in one group, and there were a number of best-of-generation individuals scoring in the gap. What's more, one run left the lower group around generation 40, eventually becoming the second best run, and the best run saw significant improvement after generation 40. This suggests that the population still contained significant diversity, and had the runs been continued past generation 50, further improvements were possible. This is in marked contrast to the size limited experiments.

The best individual from all runs succeeded in 192 fitness cases, for a disappointing score of 47.06%. Some example images are shown in Figure 18. The individual generally performed better on the FRC images than the NSH images. Images where all six fitness cases succeeded were rare, and only happened on the FRC subset. One such image is shown in the upper left. From a casual inspection, the best performance on a single NSH image was three out of six cases correct, as in the upper middle image. With the exception of missing a few walls at the bottom of the image, the errors were all mislabelling ground as non-ground, i.e. thinking the boundary between ground and non-ground was

Figure 19: Best, Median, Average and Worst Fitness vs. Generation for the Best Combined Run



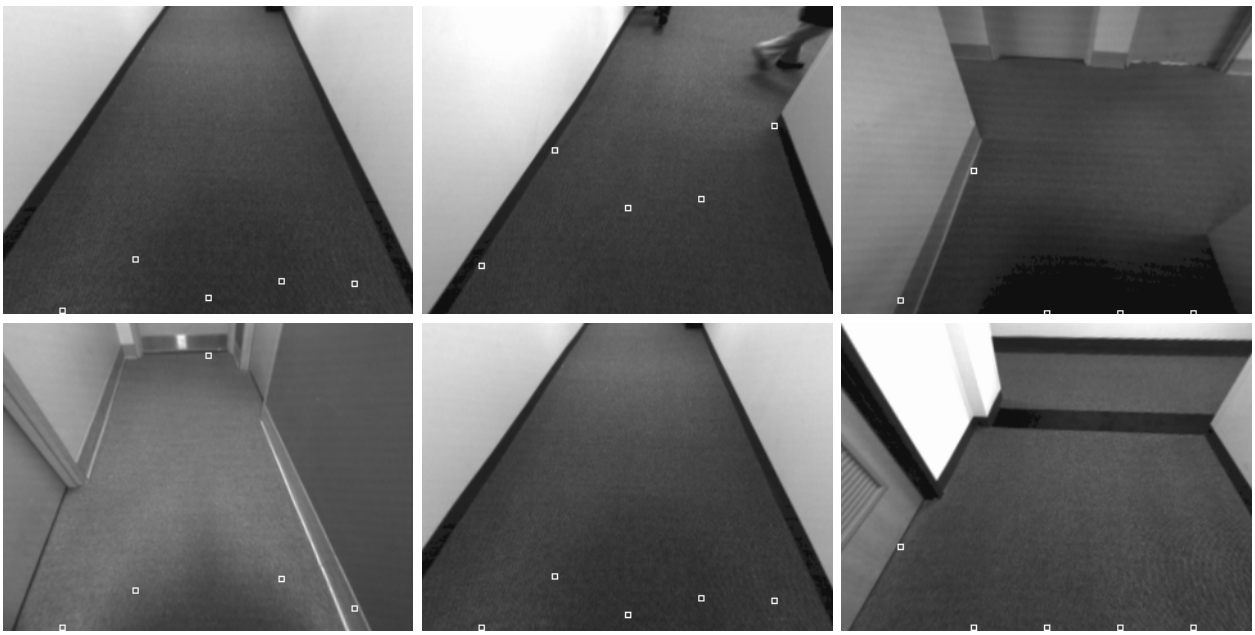


Figure 20: Performance on test (non-training) data of the best individual from the combined data set runs.

below the real boundary, somewhere on the carpet. The remaining four images show examples of that. This is far too poor to be used for obstacle avoidance.

Figure 19 graphs the fitness of the best, median, average and worst of each generation in the run that produced the best individual. These results challenge the idea of the median as an indicator of convergence. Substantial improvement happened twice while the median was substantially equal to the best.

The performance on test (unused training) data, using different columns, is shown in Figure 20. The program used during this validation was again simplified in a way that may perform slightly differently on non-training data. On forty eight images of non-training data, five columns per image, the best individual achieved 111 out of 240 columns correct, or 46%. The criterion here was more forgiving than that used during evolution, counting only errors the author deemed relevant for obstacle avoidance.

Again the performance was better on the FRC data than the NSH data, scoring 67% correct on the former but only 23% on the latter. The errors were again almost entirely classifying ground as non-ground.

Observations

This section records trends and observations, most of which are unsubstantiated yet suggestive. They are discussed further in the Discussion chapter.

In almost every run of all three experiments, the best individual in the final generation outperformed the best individual from the initial generation, and in the more successful runs by a wide margin. Profiling data shows that later individuals spent more time examining the image. In evaluating the initial, random population, about 15% of CPU time was spent calculating the averages of image operators, whereas overall for a run, about 40 to 50% of CPU time was recorded there. This was true across all three experiments. As well, the more successful runs generally took more CPU time to complete, suggesting that creating programs at random emphasizes the wrong part of the space, and that the evolutionary computation is not finding the best trivial solution, but rather a substantial solution. This is further suggested by the workings of the simplified individuals, which perform non-trivial computations over the image.

The choice of image operators should be noted. In particular, the average of the raw image values is never used in the three individuals described here, although the average median value is used several times. The other operators were all used, except the vertical component of the Moravec interest operator.

Interestingly, all distributions were bimodal, and with one exception, runs that finished in the second group were already there by generation 11. The one exception was a strange run that took large amounts of computing power, and did worse than the other three runs in its category. While more runs would need to be performed to verify this trend, it suggests a strategy of rejecting runs that have not scored over, say, 50% by, say, generation 20.

As noted, in the best runs, no more improvement seemed to take place after the median fitness became substantially similar to the best fitness for a few generations. The data from the other runs could be explored to determine the value of this condition.

The successful individuals analyzed here all performed very different computations to decide between non-ground at the bottom of the image, and a ground/non-ground transition in the middle of the image. These tasks are very different, since the first looks at the type of image, whereas the second looks for a change in the image.

Finally, the best individuals here made less use of state than those in the expanded representation. This may have been due to the lack of a seed. The seed provided a framework that already used state for the evolution to modify. It may also have been due to the change in fitness measure, since the state was used to delay calling `break` by an iteration or two in order to get a few pixels closer to the ground truth.

The experiments of the last two chapters have applied genetic programming to robot visual obstacle avoidance in various ways, in a number of environments, and shown that the solutions achieved generalize well to

non-training data taken from the same run as the test data. However, algorithms that work well on canned data are one thing, but the ultimate goal of this work is algorithms that work well on a real robot, in the real world. That is the subject of the next chapter.

8 Online Validation

Previous chapters evolved vision algorithms that succeeded both on training data and on other test data taken at the same time. However, the real test—and the point of this dissertation—is how well they work when controlling a mobile robot.

There are many reasons to be pessimistic. The evolved algorithms might be sensitive to the tilt of the camera, the iris setting, and the exact location the images were taken from. It might be confused by objects it had never seen during training. Months or years of wear and tear on the carpet might subtly change the observed gradient, causing problems. This chapter reports on the exploration of these issues. First, the experimental setup is described, then the results are reported, and finally the main observations are recorded.

Experimental Setup

Record Video Robot, Real Time

Learn Offline Genetic Algorithm

Build Navigation By Hand

Validate Online Robot, Real Time

For reasons described in the previous two chapters, *a priori* arguments suggest that if there are any differences in performance, the simplified individuals should generalize better to new data than the individual as evolved. In addition, these simplifications can speed up the computation by an order of magnitude, especially when it eliminates an image operator which is time consuming to compute, such as the median based filters. For these reasons, the simplified individuals were used for these tests.

The individual is executed using the same implementation of functions and terminals as during learning, with the exception of the image operators. During learning the texture values are precomputed for areas of all images, but during a run they are computed only when and where needed.

The next step was to run the algorithm on a live video stream and display the results in a window. Not only did this facilitate debugging, but it allowed the properties of the evolved algorithm to be explored in a qualitative way and to estimate what would be needed in a navigation algorithm. To the great delight and relief of the author, all algorithms generalized well to the new data.

A *validation run* was a single run of the robot, i.e. lasting from the moment the navigation algorithm was given control until either it collided with some object, or was shut off by hand. A *validation set* was the set of all validation runs for a single individual in a given environment. In this dissertation, all validation sets contained five validation runs.

With the exception of adding a chair and replacing the burned out lights in the FRC hallway, and creating barriers as discussed below, the hallways were not modified for the validation. Office doors were not opened or closed, nor were the environments cleaned up in any way. The tether can be seen in approximately one half of the images, and was kept to the side and out of the robot's way as much as possible. People were not excluded, and the author occasionally walked through the scene. The robot moved slower than people walked, so in practice people were not a significant concern. In the FRC runs, only one person

other than the author appeared. The NSH hallway was more active, with people visible in most runs. People were not given any instruction, and naturally avoided walking in front of the robot. Those who asked were told it was quite ok to walk around it. The people and the tether were the only dynamic elements in the scene.

All validation runs used the same camera as was used during data collection, at the same location and orientation on the robot. The iris was adjusted before each validation set by running the individual on the live video stream, and choosing the setting that seemed to give the best results. A rough setting was all that was needed. The camera's automatic gain was on, but this only partially compensates for changing iris settings.

Because the evolved programs take in a column number and return the closest non-ground object in that column, they can be run on any number of columns at any horizontal locations. While six columns were used in offline training, that seemed inadequate to localize doorways during navigation, so twelve columns were used instead.

The returned values are very similar in spirit to sonar readings, in that they represent the estimated distance to the nearest obstacles in a fixed set of directions. As such, the navigation algorithm used here is virtually identical to that used in the chapter "Data Collection," and is not considered a contribution.

The ground in front of the robot was occasionally misclassified as an object, in a way that would signal a panic halt. However, it was observed that most errors were momentary, lasting only one or two frames, even when the camera and environment were static. Therefore, the readings were replaced with the furthest of the current and previous readings in the same column. Such temporal filtering was the only major modification needed to adapt the sonar navigation algorithm to the data returned by the evolved individuals.

The main loop proceeded as follows. The program asked the video digitizer for the most recent frame. If no such frame was in memory, the program waited for the one currently being digitized. The image was then dewarped into a separate array, and the original frame returned to the digitizer. The dewarped image was then padded by copying it into an array with two extra pixels on every side; the border pixels were set to the value of the closest image pixel. Because the image operators use at most a five by five window at every pixel, this allows them to be run even at the boundaries of the image, although in practice this was not needed. The individual is then executed on twelve different columns of the image and the results stored in an array. The array is then handed to the navigation algorithm, which sets the speed, heading and curvature of the three degree of freedom robot. Finally, the image and its columns are displayed and the motion commands sent to the robot. Then the main loop repeats.

The images can be stored to disk, with the twelve estimated locations marked with boxes. This mechanism was used to produce the images in this chapter as well as the MPEG videos.

The Navigation Algorithm

The navigation algorithm used estimates of the various robot parameters, such as the height of the camera and the width of the robot, to roughly translate locations in the image to locations on the ground, and vice versa. It started by comparing, in each column, the current reading

to the previous, and used the highest of the two, i.e. the furthest away. This filtering removes transient errors that would otherwise cause the robot to halt or take evasive action.

Following the Property Mapping approach of [Nourbakhsh 2000], the filtered object locations were first classified as *near*, *medium* or *far*, and different code executed in each case.

The Uranus mobile robot consisted of a rectangular base 76cm by 61cm, with a circular sonar ring of diameter 71cm centered on the base. Thus, the front of the robot extended 2.5cm beyond the sonar ring. All three areas, i.e. near, medium and far, were rectangular. An obstacle was classified as "near" if it was within one foot in front of the sonar ring and 0.2 feet on either side. Medium was defined as 2 feet in front of the ring and 0.6 feet on either side. Finally, far was defined as 4 feet in front of the ring and 1.2 feet to either side.

The speed was determined first, independently of the direction. If any filtered reading was near or if the robot was *boxed in*, the robot immediately halted. "Boxed in" was defined as at least 11 out of twelve readings closer than a horizontal line 4 feet from the robot, and without a *gap*. Intuitively, the gap is an opening on the sides of the image large enough for the robot to turn through; it is defined precisely below. This sort of halt is termed a *panic halt*. Once panic halted, the robot waited until there were no near readings for one half second, in case the missing readings were transient. If it was halted for more than two seconds, it started turning in place. It picked a direction when it first started to turn, and kept turning in that direction until there had been no panic halt condition for one half second.

When it was not panic halted, it then checked to see if there were any medium readings; if so, it slowed down to 2/3 of its maximum speed. Otherwise, it traveled at full speed.

Next, the direction to travel was determined. If there were any medium obstacles in front of the robot, it would curve away from them. The rate of curvature was proportional to the horizontal location of the most central medium reading. That is, if there was an reading directly in front of the robot it would turn sharply, if not then more slowly depending on how far to the side a reading was found. The direction was chosen by starting from the outside and moving in, looking for the first medium reading; the robot would turn away from such a reading, turning left in case of a tie.

If the robot was not panic halted and there were no medium readings in front of the robot, the rest of the medium area was examined. If there were any such readings, the two readings on each side of the robot that were within 6 feet of the sonar ring and closest to the center line of the robot were used; a line was fit separately on the left and right. As long as there were at least two such readings, the line was considered valid if the line sloped toward the robot, if the slope was less than 2.5 (in world coordinates, not image coordinates), and if there were any medium readings on that side. If neither line was valid the robot went straight; if only one side was valid the robot turned to be parallel to it; and if both were valid, it turned toward the intersection of the two.

If none of the above conditions applied but there were far readings in front of the robot, it would turn to avoid them. The magnitude was determined using the same criterion as the medium front case. To find the direction, the robot first searched for a *gap*.

To find a gap, the readings median filtered horizontally. That is, the median filtered estimate at a given column was the middle distance of it and its two neighbors. At the ends, the maximum reading, i.e. closest to the robot, of it and its single neighbor was used. At each end of the array, an extra "virtual" reading was added that was 15 pixels below the bottom of the image. Then, the differences between adjacent readings were computed, and if any difference was greater than 90 pixels, it was considered a gap. In this case, the robot would turn to the side with the largest gap. This gap determination algorithm was also used when determining whether to panic halt, as described above. If there was no gap, the robot headed for the median filtered reading that was furthest up the image.

Finally, if none of the above cases applies but there were far readings that were not in front of the robot, it again fit a line to each side and used the same algorithm as in the medium sides case, with the exception that if both lines were valid, it turned to become parallel to the closer line. If there were no readings in any of the three areas, the robot headed straight.

The same algorithm was used with all best-of-experiment individuals. The performance of each one, both from looking at raw video and during subsequent navigation, is discussed below.

Results

Movies of all validation runs can be found on the web site for this dissertation, www.metahuman.org/Martin/Dissertation. These movies show the images used during navigation, with the output of the evolved individual drawn on them. The output was temporally filtered as discussed at the beginning of the navigation algorithm above.

The best-of-experiment individuals generally performed very well. An inspection of their performance on live video suggests that the performance is more than good enough for navigation.

As with data collection, two environments were used. The two best-of-experiment individuals from the FRC and NSH experiments were validated in their own environments, whereas the individual from the combined experiment was validated in both. The discussion below is organized by environment.

Field Robotics Center

Due to the limited reach of the tether, if a robot went too far down the hallway it would have to be stopped. Therefore, a waste paper basket and an empty water jug were placed in the environment to create a closed area in which the robot could run continuously until it collided with an object in the environment. A chair was added to see how well it avoided obstacles that were not present in the training data.

The robot started in the same location as the data collection run and retraced the same route, through a turn to the end of the hallway. Upon reaching the end, the robot would halt, turn in place, then retrace the entire route. Thus, the first half of the lap repeats the trip made during data collection, while the second half views the same area from the other direction. The ends of the hallway, where the robot turns around, were not seen up close in the training images.

The individual trained only on the FRC data is discussed first. During the adjustment of the iris setting, it was noted that the individual worked well on a large range of settings; the best iris setting was approximately the same as that used for the training data, about a quarter of the way from fully open. Objects that had not been seen during training, such as chairs and people, were handled well as were ten degree differences in camera tilts. It did not do well with the floor of the lab, which is much brighter with a higher gradient. It nearly universally marked it as an obstacle.

Transient ground errors were more common (and less transient, lasting two or three frames) on small strips of white paper and shiny pieces of metal on the carpet a couple pixels wide. This was enough to cause a momentary panic halt but not otherwise affect navigation.

On the 700 MHz Pentium III, when only the image processing and display are running, the FRC individual ran at 7 to 10 frames per second. This used an iterpreted form of the individual, the same process used during evolution; rewriting it in C would most likely speed it up.

During the actual validation runs, the majority of time was taken up communicating with the robot and saving images to disk, so that the frame rate was between 1.7 and 2.0 frames per second, averaging approximately 1.87 fps. Of course, this could have been sped up greatly.

In operation most of the individual's failure modes were transient, even when the robot was stationary, and did not cause the navigation to fail. The persistent errors were at the end of the hallway near a closed door, where the limited light and the robot's shadow made the ground much darker, and near the side of the filing cabinets. Most errors classified obstacle as ground, i.e. they were not conservative.

Table 1: Validation Runs of the Best FRC Individual

Number of Laps	Length of Run Min:Sec / Meters	Failure Mode
5.5	20:46 / 150 m	Did not see door at the darkened end of the hallway.
0.75	2:56 / 21 m	Collided with chair.
1.75	8:33 / 48 m	Collided with chair.
6.75	27:59 / 185 m	Collided with chair.
9.5	39:32 / 260 m	Did not see door at the darkened end of the hallway.

The five runs of the validation set are summarized in Table 1. A *lap* took the robot from its starting position through a turn to the end of the hallway, then retraced the entire route, leaving it where it started. Two runs ended because of the aformentioned problem that the evolved individual has at the end of the hallway. The other three ended when the robot collided with the chair. The evolved individual had clearly identified the chair as an obstacle in several columns of several frames.

The output of the evolved individual on the chair danced around the legs. This was more than sufficient to cause a panic halt. Once the robot had panic halted and started turning, eventually only one column that imaged the chair would be in the panic halt region, and if this did

not see the bottom of the chair on two consecutive frames, the robot would start moving again. This is what caused the robot to hit the chair in three of the five runs.

The combined individual generally performed worse than the FRC only individual. Again during the adjustment of the iris, only a rough adjustment was needed since a wide range of positions worked well. The best setting was approximately the same as the training data and the FRC only individual. Objects not seen during training, such as chairs and people, were also identified well. As with the FRC only individual, it did not do well with the floor of the lab, often mistaking it for an obstacle.

During operation, errors were mostly transient, although objects a few pixels wide would again confuse it for two or three frames. The combined individual ran at 5 to 8 frames per second. Profiling shows that it spent two thirds of its time computing the median filter. The code was not optimized, and in fact computes the median twice at every pixel. Therefore, simply reorganizing the code to only compute it once could speed it up by a third. During the actual validation runs, while storing images to disk and communicating with the robot, the frame rate dropped to 1.80 to 2.15, averaging approximately 1.83 fps. This could be greatly sped up as well.

In operation most of the individual's failure modes were transient, even when the robot was stationary, and did not cause the navigation to fail. Persistent errors were mislabeling doors as ground, and mislabelling doors as ground when close to them and looking at them face on, which only happened while turning around at the end of the hallway. Finally, it would also label a dark shadow at the bottom of the image as obstacle for a few frames. Most errors classified obstacle as ground, i.e. they were not conservative.

Table 2: Validation Runs of the Best Combined Individual in the FRC

Number of Laps	Length of Run Seconds / Meters	Failure Mode
4	15:07 / 110m	Evolved individual missed bottom of filing cabinet while turning around.
3	11:07 / 82m	Collided with filing cabinet behind it while turning in place. A cylindrical robot would not have the same problem.
2	7:17 / 55m	Evolved individual missed bottom of filing cabinet while turning around.
2.5	9:28 / 69m	Evolved individual missed bottom of filing cabinet while turning around.
2.75	6:23 / 75m	Collided with chair.

The five runs of the validation set are summarized in Table 2. One run ended when the robot collided with the filing cabinets behind it, out of the view of the camera. Such a case is difficult for a reactive system to avoid, although a cylindrical robot would not have had the same

problem. Three runs ended because the evolved individual misclassified the bottom of the filing cabinet as ground in two consecutive frames. When the robot was nearly done turning, this was the only column of the filing cabinet in the "panic halt" zone, so the robot would start moving prematurely. This is the same circumstance that caused the FRC runs to collide with the chair. The last run collided with the chair in the same manner as in the FRC runs.

Newell Simon Hall

In these validation sets, the individual was limited to a single lap. After the first two runs of the combined individual, it was decided to divert the robot into areas it had not seen during training, from which it would find its way back and complete the lap. As with the FRC validation runs, the robot started by retracing the route it used during training. In all runs, it treated the black stripe at the end of the hallway as obstacle, causing it to turn back. While this was happening, a large recycling basket was placed in its way, causing it to divert from the hallway and head toward a kitchenette. The linoleum floor of the kitchenette was labeled as obstacle, causing it to double back towards a copier room. Again, the white floor of the copier room was labeled as obstacle, causing it to head back toward the hallway. It would then find the black stripe once more and turn around again to find the recycling bin gone and the path clear to return to where the experiment began. This distance was approximately 40 m.

The individual trained on only the NSH data is discussed first. Again, while adjusting the iris, the individual worked well on a range of settings, so only a coarse adjustment was needed. The best iris setting was approximately the same as that used on the training data, between a half and three quarters of the way to fully closed. There were no pieces of metal or paper on the floor, so transient ground errors were less common than in the FRC.

On the 700 MHz Pentium III, when only the image processing and display were running, the NSH individual ran at 9 to 11 frames per second. Again, several speed gains suggest themselves, so faster performance is possible. During the actual runs, image saving and communication with the robot drops the frame rate to 1.71 to 2.65, averaging 2.15 fps, which could be greatly sped up.

In operation most of the individual's failure modes were transient, even when the robot was stationary, and did not cause the navigation to fail. When the black patch of carpet was very far off (too far to affect navigation), the individual marked it as obstacle, but before it came within range, it correctly labeled it as ground. However, once the robot was almost on top of it, it was labeled as obstacle, and the robot panic halted and then turned around. As with the FRC individual, the output of this individual danced around the bottoms of obstacles not seen in the training data, such as the recycling bin or people's feet, although it successfully avoided them. These errors classified obstacle as ground, i.e. they were not conservative.

Four of the five runs succeeded. In the exception, the second run, the robot hit a door that was not seen during training. This door was mislabeled as ground in most columns of all runs. In the first run, a few columns in a few images labeled it correctly as obstacle; this was enough to avoid it. Similarly, in the third run, the robot saw enough of it

to panic halt and start to turn; near the end of the turn, a person opened the door revealing the white floor beyond. The white floor this was consistently labeled as obstacle and the robot avoided it. In the fourth and fifth runs, people opened the door as the robot was approaching it, and the door stayed open long enough to be avoided.

When the combined individual was run, the iris setting was accidently left at the proper location for the FRC hallway, which produced images significantly brighter than the training images. Again most mislabelling from the evolved were transient and not conservative even when the robot was stationary, and did not affect navigation. Perhaps because of the brighter images, the combined individual performed better than the individual trained on the NSH data alone except in detecting the black stripe of carpet, on which it consistently labeled as obstacle. In particular, it was better able to identify the bottoms of object not seen during training, such as the recycling bin and the bottom of the new door. However, it often labeled the brightest parts of the carpet as obstacle, in a way consistent enough for obstacle avoidance to avoid it.

The first two runs did not divert the robot to the kitchenette and copying machine. The robot successfully navigated the route used during training, with the exception of the black carpet, which caused it to turn around and retrace its steps. It returned to where it started without incident. The other three runs used the recycling bin to divert the robot. Two of these runs were completed without incident. In the other run, just after being diverted, the misclassified the ground as obstacle consistently enough to panic halt and turn around. It then went back to the black stripe of carpet, turned around and was diverted again. Once again it misclassified ground as obstacle, and eventually turned around and headed for the black stripe. After turning around at the stripe, the recycling bin was removed and the robot headed back to its initial location without incident.

Observations

With the exception of the red carpet, the failures of the obstacle avoidance were in areas not seen during training, such as the darkened end of hallways or previously unseen doors. A more diverse training set, including a larger variety of obstacles, may prove effective. Co-evolution, discussed in the next chapter, could allow for a greater training set without increasing the run time of the offline learning.

Another possibility is the use of state. Objects could be identified and tracked from frame to frame. With a more powerful evolutionary system, this competency might arise naturally with the fitness function used in this dissertation, or could be bootstrapped by having the individual predict future locations of the ground/non-ground barrier from previous locations.

The error properties bear some resemblance to those of sonar. Transient errors that usually overestimate an obstacles distance are most common, although consistent errors also occur. This suggests using navigation algorithms that are typically used with sonar.

The navigation algorithm was not considered a contribution and was therefore created and debugged in only of a few days. It could clearly be improved. The process of creating it involved guessing at a reasonable solution, trying it out, seeing where it failed, then changing

the algorithm to perform better in that case. This is exactly the sort of process that this dissertation automates. In other words, once the vision algorithm is fixed, it could be the input to a navigation routine which is designed using evolution. This topic and more are discussed in the final chapter.

Part III

Reflections

9 Discussion

Experiments

The experiments in this dissertation fleshed out the ideas in the chapter "Technical Framework" and demonstrated the mechanics of the research program. At the end of each chapter, the section "Observations" described the points that deserve further reflection. Those points are summarized here and discussed further.

With very few exceptions, the initial population generally did as bad as or worse than the best constant approximation. The average time for a focused representation run—fifty generations, four thousand individuals—was just over a day on a dual 700 MHz Pentium III. The runs that converged on better solutions generally took longer, and spent a larger percentage of their time in the image operator terminals.

In the three focused representation experiments, the fitness of the best-of-run individuals was bimodal. With one exception, runs that ended in the better scoring group had best-of-generation individuals with fitness of over 50% by generation 11. The exceptional run took an inordinately long time to run and performed worse than any other individuals in the better scoring group. Fitness did not improve significantly once the median fitness came within one percent of the best fitness for a few generations. By contrast, there was not any obvious correlation between average fitness and future improvement.

With the exception of the non-seeded expanded representation experiment, all experiments produced an individual which achieved a fitness of greater than 85%. They did this despite burned out lights and other effects that caused the carpet's average intensity to vary from zero to at least 140s out of 255; despite large gradients caused by luma coring; despite moiré patterns of image noise; and despite the shadow of the robot.

All of the best-of-run individuals that were examined in this dissertation handled the bottom of the image differently than the rest. At the bottom, the task is to decide whether or not the window images carpet, whereas in the rest of the image, the goal is to find the boundary between carpet and obstacles. In the focused representation, the individual that was trained on the FRC data used the two branches for the two cases, the best of the NSH runs tested the value of `first-rect` to modify the values in many tests, and the best of the expanded FRC runs tested `first-rect` at the beginning of each iteration. There is nothing in the representation, fitness function or other details to suggest such a division. In fact, this aspect of the problem was not appreciated by the author until the best individuals were simplified, and was not a part of any of the relevant related work. This is an example of the genetic algorithm simultaneously exploiting regularities in both the problem domain and the representation.

All best-of-run individuals worth examining used the strategy of, in each window, deciding whether or not the window contained an object or boundary between floor and object, and setting the "a" register to the midpoint of the lowest such window. They all used multiple `if-le`

statements to create a high level structure similar to a decision tree. However, the conditions being tested were often non-linear relationships between two or more image operators. One individual's second branch detected objects that go to the bottom of the image, not by using a decision tree, but instead by iterating a continuous function of an image operator and previous register value.

With one exception, the registers were not used to maintain state from one window location to the next, only to set the return value, which was never read. The one exception was the best individual from the seeded, expanded representation runs, which calculated the change in average gradient magnitude over both one and two frames, and ensured that a condition must be true on two consecutive iterations.

Random constants played a smaller role than they did in [Koza 1992]. Instead, `image-max-y` was more common in that role, more common than both random constants and other constants. The horizontal location of iteration was always in the desired column, that is, the second parameter to `iterate-up` or `iterate-down` was always numerically equal to `desired-x`, so that the window was always centered in the column it would be judged against. The representation in no way implied such a correspondence, in fact, the correspondence was only implicit in the training data.

On the training data the best individuals had only transient errors that did not pose a problem for obstacle avoidance, except for the black carpet on the Newell Simon Hall runs, which was consistently interpreted as an obstacle. They generalized well to different columns and live video from the same hallway, requiring only minimal filtering. By examining their simplified forms, it is clear that they make specific assumptions about their environment, for example that the ground has little or no visual texture. However, the success of the experiment that combined data from both the Field Robotics Center and Newell Simon Hall suggests that we can create one algorithm that works in multiple environments by simply combining data sets.

The graphs of fitness vs. generation also show that most improvement happens early on. After the first 20 or 30 generations, only marginal improvement is observed, even in poorly scoring runs. This is confirmed by the seeded expanded runs, which did not record significant improvement when extended for another 50 generations.

Discussion of Experiments

For those looking to use the framework mostly as is, the bimodal distribution of the best-of-run individuals, along with the observation that runs that were going to get good did so by generation 11, suggests a strategy of terminating runs if they do not achieve 50% by, say generation 20. Also, the relatively minor improvements once the median fitness becomes close to the best in a given generation, again suggests a criterion for halting a run. Similarly, there was not much improvement after generation 30, and the seeded expanded runs, which continued to generation 100, did not see a lot of improvement. This suggests extending only the best runs if such incremental improvement is desired. It would be interesting to continue runs to a large number of generations, say 1,000 or 10,000, to see if any significant improvement develops. For those seeking more fundamental improvement, these results suggest that the population has become largely the same, differing largely in branches that are never executed or in other ways that do not affect per-

formance. Various ways of maintaining diversity are discussed in the section “Future Work” below.

Does the fitness measure used for the vision subsystem correlate with performance avoiding obstacles in the real world? Because only the best individual in each experiment was validated by developed it into a full obstacle avoidance system and evaluated in practice, this dissertation does not answer that question. However, several clues are apparent. As the online validation made clear, the individuals did almost as well on new data from the same environment as they did on the training data, implying that they did not overfit. The error rates achieved, approximately 10%, are low enough that errors can be rejected if they are distributed evenly throughout the training data. However, if the errors followed some pattern, even error rates this small could seriously hamper obstacle avoidance performance in practice. This was borne out in the online validation: in the Field Robotics Center, where all errors were transient, the robot navigated quite well, but in Newell Simon Hall, where red carpet is consistently misclassified, the robot never succeeds in traversing it. Thus, at least in these experiments, it can be said that (a) performance on the training data correlates well with performance on new data, and (b) that overall error rates in either case correlate well with performance in obstacle avoidance only when errors are not systematic.

While Liana Lorigo’s work used a 20 by 20 window in a smaller image, the best individuals here succeeded using much smaller windows. That may be because neighboring columns often had objects even when the desired column did not, meaning that a wide window would often detect the wrong object. Also, as long as objects can be reliably detected, a smaller window will localize their location with more accuracy.

Because the representation is relatively low level—the atoms are similar to those of a traditional programming language—they could be put together in many ways and supported many styles of programming. However, the examination of the best-of-experiment individuals was as interesting for what it did not find as for what it did find.

Most interestingly, only the seeded experiment used the registers for maintaining state from one iteration to the next. Without this, it would most likely be impossible to distinguish the black swath of carpet from the baseboards, especially since significant portions of each were set to the same value by the luma coring. In this particular environment, treating the black baseboards as ground would probably not affect obstacle avoidance. However, with the evolved individuals used solely to estimate the boundary between ground and non-ground, this would need the ground truth to be modified. However, if the evolved programs computed the direction to travel, or were simply scored on the navigation ability of the overall navigation algorithm, such a solution might easily be discovered.

Returning to the lack of use of registers, if EC had problems with two step processes, this could certainly explain it. In order for memory to be used, it needs to be written during one iteration and read during the next. The writing provides no visible benefit, and hence is subject to genetic drift. In practice, although setting the registers was common, they were often set to different values on different paths through the individual. Unless these different values corresponded to something useful to the next iteration, making decisions based on them would most

likely hurt an individual. Given that the majority of fitness cases could be handled without it, there simply may not have been enough of an incentive to use them. Co-evolution, discussed in the section "Future Work" below, could help with this.

The evolved program's use of the language certainly differed from the way a human programmer would have used it. First of all, many things were computed that were not needed. There were many "if" statements whose condition always evaluated true or always false, and values were often computed and then thrown away. This is not surprising given that there was no penalty for it. Also, the value of a single terminal was often written to a register, then the register used in an expression; in such a case, using the terminal directly would have been clearer.

Clearly there are certain constructions that human programmers come up with that would be difficult for the system in its current form. Coordination for example, where, say, a variable is set and used in many different locations. Knowing that the form must have a certain structure is another, for example knowing that image operators must influence the result. However, these could be incorporated into the framework. Crossover can cause the same subtree to occur in many different locations, and automatically defined functions can also help. Typing can enforce various structural constraints. It would be interesting to add more language constructs and genetic operators, based on ideas from procedural, functional and object oriented programming, and perhaps incorporating various software design patterns.

It should be clear that it was not restricted by the normal problems with human readability of code. Does it only create messy, spaghetti code that is difficult or impossible to modify in useful ways? While a random search might, simulated evolution selects for programs that are evolvable, that is, that when modified by genetic operators, are likely to produce better results. Co-evolution, described in the section "Future Work," should facilitate this even more.

We can also ask if it generated any solutions that people would not have generated. Certainly the implementation is different, but when simplified, we can identify elements that seem familiar, such as ignoring the gradient when the image brightness is low, which presumably ameliorates the effects of luma coring. Can every line of the programs be similarly explained?

At this point we simply can not know. This dissertation is a first step of an entire research program, so it is quite possible that the solutions discovered here are equivalent to those a human programmer would construct. However, structures such as the recurrent mathematical expression, the nonlinear conditions and the particular use of registers in the best expanded run all point to the possibility of a more subtle mechanism.

It should also be noted that the above discussion demonstrates that far from being opaque, both the evolved individuals and the evolutionary process can be understood well enough to suggest improvements and new directions for research. The focused representation was borne out of observations about what the best individual in the seeded experiment kept from the seed, and what was modified. The result producing branch was simply used to return the value of one register, which meant that only a single iteration branch was ever used. As well, within certain limits of complexity, any computation on a register in the result produc-

ing branch could be done in the iteration branch before the memory location is assigned, and since such complexity was not being used, no expressive power was lost. The fitness measure was redesigned using the reflections described above.

Discussion

There are a number of ways of looking at the approach of this dissertation. At the most practical, the approach in this thesis automates all the tweaking that researchers do when developing algorithms. For example, the evolved individuals would declare a boundary in regions where there were large intensity differences. However, the luma coring in the FRC focused representation experiment created just such large differences on the carpet. Therefore, the evolved program developed a branch to ignore image differences when the average image brightness was near the cutoff.

Such tweaking is both essential for creating a working system, yet not considered research in and of itself. Much of the time spent in getting a robot to work in a competition or museum involves such tweaking, and distracts from the research. The system described here has the potential to accelerate research in the same way that buying pre-existing computers and robot hardware does: it solves practical problems, allowing researchers to spend their time on research.

This same idea can be applied in many different areas. When approaching almost any problem, we can often think of many possible approaches. For example, if a camera is tracking a lecturer and trying to determine where they are looking, we could separate the lecturer from the background by tracking skin tone, performing background subtraction, using gait analysis, etc. Given pencil and paper, we could write out these algorithms in pseudo code. The framework advocated here can be seen as searching this space, looking for a combination of approaches that works well. In presentations of this work, this is the aspect which excites people the most.

From a machine learning perspective, it can be seen as using a novel representation, essentially that of a traditional programming language. Previous uses of machine learning to create programs that control mobile robots have used continuous functions or other representations which seem unnatural outside of machine learning. In other words, when a programmer sits down to write a program to control a robot, they will not typically create a single continuous function that maps an array of image values to a steering direction. Instead, they will most likely use loops and flow control in addition to continuous functions. Methodologically, they will construct a program by specifying symbols and their relation to each other. While the idea of searching the space of programs has been around for decades, it is only now feasible to apply it to practical robotics problems in unstructured worlds.

This representation has many advantages. It allows us to leverage most existing algorithms in vision and robotics, since they are already expressed in this form. We as programmers have a lot of insight into the properties of this representation, since we have used it ourselves. It also is amenable to special hardware. For example, in retrospect, the representation in this dissertation is similar to that provided by DataCube, a set of computing hardware elements that can be combined into image processing pipelines. It might be worthwhile to use the basic elements

provided by such hardware either as an inspiration, or copy them exactly and evolve programs for the hardware.

It also changes how the evolved programs generalize to new data. In supervised learning, it is the representation that is crucial in determining how the learned algorithm generalizes to new data. We program using traditional programming languages in part because we believe that the proper algorithm should have a brief description in such a language. This is also why we use mathematical expressions; again, we feel that mathematics is the "right" language for many things, that significant insight can be expressed briefly in mathematics. When discussing simplification, it was argued that if the program always took a given branch of an `if-le` on the training data, then it should probably be modified to always take that branch, on all data. This can be seen as a special case of the general argument that code is the right representation for such algorithms.

To give a concrete example, stereo vision uses a formula to compute the degree of similarity between two windows, one from each camera. This formula can take many forms, e.g. sum of squared differences, correlation, correlation after subtracting the mean intensity, correlation after subtracting the mean intensity but only if the mean intensities are similar, correlation that subtracts the image mean but adds a penalty proportional to the difference in image mean, etc. When creating this by hand, we are much more likely to use a simple mathematical expression than, say, adjusting the coefficients of a tenth degree polynomial. We feel that this is the "natural" representation for the problem. There is no reason machine learning can not use this representation too.

While the programs that evolve have a relatively short expression as code and/or mathematics, they may not be conceptually clean. For example, the best-of-experiment individual from the seeded runs subtracted the average gradient over one window from the average over the next. This effectively computed the average gradient over one horizontal line, minus the average over another line eight pixels away. Is this an approximation of a second derivative, or is the second derivative an approximation to the most useful operator? This applies to other endeavours as well. For example, should a certain network of neurons be viewed as computing an approximate Fourier transform, or is the Fourier transform merely an approximation to some more useful function that the network is computing? Is the wax and wane of caribou populations a discrete approximation to a differential equation, or vice versa?

While these questions do not need to be answered by those interested in applying or developing this technique, they do seem to be important for an understanding of its role. If conceptually clean algorithms and mathematics are truly fundamental, then the evolutionary approach will forever be doomed to approximating the best approach. If not, then evolutionary computing may search a larger, richer part of the space, allowing a better approximation than conceptually clean algorithms do. While the author sides with the latter, as with most fundamental questions, only time will tell. This seems to be a modern incarnation of the difference between "neat" and "scruffy" AI.

The shift from creating all details by hand to giving some of the work to machine learning subtly influences the research program. It allows researchers to focus on the representation and let the computer take care of applying it to a particular problem. Instead of having to

advocate all the details of a single technique, the researchers can specify a space of techniques to be searched. They must be careful not to bias the space toward a single class of solution, or to make even simple solutions too hard to find, as was apparently the case in the expanded representation experiments.

The discussion to this point emphasizes the search for solutions that researchers could find eventually anyway. However, there is every reason to believe that the solutions found could be more complex than those constructed by hand. For example, the people finder could use a combination of background subtraction, skin hue tracking and motion analysis, using each one to assuage the deficiencies of the others. In fact, this may have happened in this dissertation. The non-linear conditions, the iterative mathematical sequence and even the number of conditions could be elements that researchers would either not find or actively avoid. Certainly, the vision systems most similar to this work, namely those of Horswill, Lorigo and Ulrich & Norubakhsh, used fewer image statistics than the evolved programs in this dissertation.

This was the motivation for the technique, as described in the next chapter, "Philosophy & Manifesto." While many of the goals of that chapter, such as non-bottom-up perception and the integration of multiple depth cues, have yet to be fulfilled, the experiments here have demonstrated tantalizing steps in that direction. Even in this nascent form it creates useful programs for unstructured environments today, programs which superficially, and potentially in their essence are of greater complexity than those previously reported. Such an ability is needed before we can create systems with the abilities of the human mind.

[Nishara 1984, *Practical Real-Time Imaging Stereo Matcher*] lists four criteria for a successful computer vision algorithm: noise tolerance, practical speed, competent performance, and simplicity. The fact that the other three are not enough by themselves is telling, it implies that simplicity acts in opposition to the other three. The advocated framework relaxes that criteria.

The advocated framework can also be seen as a form of evolutionary computation that is friendly to the incorporation of existing knowledge and techniques. This is often discouraged out of an attempt to be *tabula rasa*, either for purely theoretical reasons or to emphasize that the user does not need to know much about the subject. It is also discouraged out of a belief that traditional approaches have emphasized the wrong things and constructed the wrong building blocks. However, such antagonism is unnecessary, and the results of this dissertation are testament to the power of incorporating the best of both worlds. This framework can therefore be seen as bridging the gap between the machine learning and hand construction communities, since they both use essentially the same representation (a programming language) but in different ways.

It has also been pointed out that the specific algorithms that were developed differed from previous work and the seed in ways not unlike how Lianna Lorigo's Master's thesis differed from Ian Horswill's previous work. From this point of view, the system can be seen as automating the Master's thesis.

Contributions

Robotics

The main technical innovation of this dissertation to robotics is realizing the value of using a programming language as a representation for learning and working the details of one way of applying it in practice. The previous subsection discusses the nuances of this in some detail.

The system developed in the experiments is another contribution. This system is a successful example of the application of genetic programming to a real world problem in robotics. A programmer could use it as is to evolve obstacle avoidance algorithms for a particular robot and environment that may work as well as hand coded ones, yet do not require the programmer to have detailed knowledge of computer vision and robotics. In addition, researchers can use it as a starting point for their own experiments, modifying it in various ways in order to improve it or adapt it to similar problems.

This dissertation also demonstrates a description of certain kinds of algorithms that the example system can learn, and a description of certain kinds it is unlikely to learn. This knowledge could help others to predict what sort of problems it could succeed on, as well as direct efforts on improving the system.

At a theoretical level, this dissertation demonstrates an inclusive framework, i.e. a framework that does not require a particular architecture of representation, but rather allows many to coexist and interact. It also shows how critiques of AI can be used to guide the search for alternative methods.

Other novel aspects of this work with respect to robotics include a list of approaches to constructing obstacle avoidance from sonar and their failure modes, and a the set of guidelines that grew out of that. In addition it demonstrates an extension of the Horswill/Lorigo/Ulrich algorithm to reactively identify objects that extend to the bottom of the image, a different task than finding the boundary between floor and object.

Evolutionary Computation

While the use of evolutionary computation to evolve algorithms to control mobile robots already existed under the name “evolutionary robotics,” it has never been applied to realistic unstructured environments, and therefore has not garnered much interest in the robotics community. This dissertation demonstrates that the approach is viable, that is, that evolutionary computation can produce vision algorithms for unstructured environments in robotics, that the algorithms do not suffer from overfitting, that they can successfully control a robot in real time using new data from the same environment. It demonstrates this by being the first to do it. In addition, this dissertation is the first work of evolutionary robotics to produce a vision algorithm that was not obvious from the problem definition. It is also the first to not force the image data through an information bottleneck.

The design of the system itself, and the exposition of its properties, is novel in evolutionary computation as well as robotics. In particular, the “iterated rectangle” node and associated image operators allow us to avoid the information bottleneck, by limiting the evolution to significant algorithms whose execution time is none the less reasonable on today’s computers.

This dissertation also demonstrates that evolved programs can be simplified and understood, and that understanding can be used to improve the system, as was shown by the changes to the representation and fitness function.

Other novel aspects in evolutionary computing include the demonstration of a different structure for individuals, namely multiple branches that communicate, not through return values, but through the final values of registers. This dissertation is also another case study of a system that contains memory registers and describes how they are used and not used. And finally, the dissertation is another example of using a seed to improve the performance of a genetic algorithm.

Critiques of Artificial Intelligence

This dissertation raises the idea that the sort of systems that AI has explored have been influenced by the affordances of the researcher's minds. In particular, when programs are created by hand, they must be understandable by people, and this is a strong constraint. It leads to modular, black box subsystems that cannot take into account each other's limitations, and ultimately brittle systems.

It also delineates a new distinction, between exclusive and inclusive frameworks. Exclusive frameworks make their contribution by proscribing large parts of an architecture or representation, thus limiting the sort of algorithms that can be created. In contrast, inclusive frameworks allow for the coordination of many different, existing approaches to work together. And finally, this dissertation contains another call for and an example of what Philip Agre calls a critical technical practice [1997, *Computation and Human Experience*].

Contributions to Robotics

1. Introduces the use of a scripting language as a representation for learning in robotics.
2. Provides an example working system that others can use as is to create obstacle avoidance algorithms on par with what people can create.
3. Researchers can use the example system as a starting point, modifying it in various ways to improve it or adapt it to similar problems.
4. Demonstrates what kinds of algorithms the example working system can learn, and what it is unlikely to learn.
5. Demonstrates an inclusive framework, i.e. a framework that does not require a particular architecture of representation, but rather allows many to coexist and interact.
6. Demonstrates how to use critiques of AI to guide the search for new approaches.
7. A list of approaches to constructing obstacle avoidance from sonar, and their failure modes.
8. Guidelines for constructing an obstacle avoidance algorithm from sonar.
9. Extension of the Horswill/Lorigo/Ulrich algorithm to distinguish objects from floor at the bottom of the image reactively.

Contributions to Evolutionary Computation

10. Demonstrates that evolutionary computation can tackle real world problems in robotics today.
11. Introduces the "iterated rectangle" node and associated image operator terminals as a way of achieving 2.
12. First evolutionary robotics system to tackle an unstructured environment.
13. First evolutionary robotics system to create a vision system that uses large parts of an image, rather than forcing it through a bottleneck.
14. Demonstrates that evolved programs can be understood, and that understanding used to improve the representation.
15. Demonstrates a different structure of individual: multiple branches that communicate not through return values but the values of registers
16. Another example of a system containing memory registers, and how they are used.
17. Another example of using a seed to improve the performance of genetic programming.

Contributions to Critiques of Artificial Intelligence

18. Human understandability is a constraint on algorithms, that leads to isolation of subsystems, and ultimately brittle robots
19. The distinction between exclusive frameworks, which specify architecture or representation, and inclusive ones, which allow many possible approaches to work together.
20. Another call for and example of a critical technical practice.

Future Work

If someone wanted to use this to create a visual obstacle avoidance system for their robot today, I would suggest using the "focused representation" setup with the following changes:

- Remove the second parameter to the iterate functions, the horizontal location, hard coding it as `desired-x`.
- Perform ten or so runs to see if your distribution is bimodal, and if so, whether the final performance can be predicted early on. If so, abandon runs that are performing poorly.
- If incrementally better performance is desired, extend the best run for another 50 generations.

While many extensions are exciting, it is easy to assume evolutionary computing is the black box that finds the best algorithm in your representation. Unfortunately, EC is not as powerful as all that. The experiments in this dissertation give some concrete examples, of both a setup that the GP found tractable, and one that it did not. As described above, there were many affordances of the representation, many possible possible algorithms that were not explored. Therefore, it is important to only

attempt extensions that do not make the search a great deal more difficult.

Near Term Projects

It would be interesting to use training data from a wider array of environments. This may lead to a more general free space finder, one that can handle a large variety of ground types.

The set of image operators could be augmented with other qualities and depth cues such as stereo, optical flow, spatial and temporal wavelet transforms, and colour. It should also be possible to evolve the image operators, to both create better ones, and see which ones are most useful. Sensor fusion might be easy in this framework; for example, sonar and vision could be an interesting starting point.

The navigation function, which takes the estimates of object locations and returns the direction to travel, could be learned. Since a continuous function seems appropriate, a neural net or similar representation may actually be best. If the evolved object estimator also provided confidences, the navigation function might be able to use them to become even more robust.

As suggested above, the framework in this dissertation might apply to more traditional vision problems, such as extracting a particular class of object from a still image or video stream.

While training on data from many environments simultaneously would not doubt prove interesting, the most general free space finder, one that can handle any kind of pattern on the carpet and transitions from one floor type to another, will probably need to look at more than just a column in the image. It may need to extract lines and edges, and use other such geometric techniques. As well, the depth of a textureless grey wall can not, in general, be estimated from the grey patch itself. Instead, any estimate would need to examine at least its boundary. A representation that allows such algorithms to be explored in a computationally efficient manner, analogous to the iterated rectangle from this dissertation, would be exciting.

These would most likely benefit from changing the representation assumed by the outputs, and therefore the fitness function. How would one evolve depth maps, 2D or 3D evidence grids? What about stereo or optical flow?

More conceptually driven elements could be encouraged by having the image operators take parameters, such as the direction along which to compute the gradient, or perhaps the mask to use with a convolution. This would allow the evolved programs to set these parameters depending on context.

Maintaining state from one image to the next could open up a new set of possibilities, including the technique of Ulrich and Nourbakhsh of using as reference a portion of a previous image that has since been traversed. Since the evolved programs are interested in the ground/non-ground transition, which is on the ground, a column in a previous image can be mapped to a line in the new image by assuming it is on the ground and that the ground is flat. This would allow approaches such as computing image differences over time, and seeing when large differences were a natural sign of obstacles.

A simulation may discover other approaches to navigation. In previous chapters it was argued that current computer graphics techniques probably were not up to the task of simulating images, but if the input to

new algorithms is the output of the ones developed here, we only need to simulate its errors. They should be relatively easy to quantify. It should be reasonable to assume they are conditional on the obstacle and independent of everything else. That is, there is a certain chance that ground will be labeled as obstacle, and that probability is independent of the column location, the frame number, or even the image as long as there is ground in that column of the image. If the ground is not mislabelled, then the boundary between ground and the lowest object is found. We decide at random whether there will be an error here. The probability depends on whether that object is a door, a wall, etc. When there is an error, it should not be hard to find an alternate value to return.

Such a simulation could be used in a traditional evolutionary robotics framework, of the types described in the "Related Work" chapter. It could then learn to do many tasks, such as wall following, navigating to a particular point, navigating to a user specified point, delivering mail, etc.

Understanding Evolved Programs

In general, the analysis of evolved programs is different than traditional methods of reading code. It is similar in spirit to the analysis of biological systems, although somewhat easier since perfect knowledge of a program's construction and components exists. In fact, in performing the analysis for this dissertation, the best analogy seemed to be analyzing the winners of the obfuscated C contest, a contest to make intentionally obfuscated computer programs.

However, this dissertation has raised the possibility that the evolved programs can be completely understood, that is, simplified until the structures and details are simple enough that we can understand how they work by applying existing knowledge of computer vision. The fact that the elements are familiar—existing operators and standard elements of mathematics and programming languages—is a great help. In either case, such analysis could ultimately help answer questions such as "Are there any obvious failure modes that we have not seen?" and even "Could a human programmer come up with this?"

The simplification process was partially automated, but automating it more fully would allow the best individual of every generation to be simplified, to see how its workings change over time, and discover how it uses the various affordances of the representation. The techniques used are similar to compiler optimizations, which suggests that our choice of representation once again allows the leveraging of decades of computer science knowledge, this time in compiler design.

After simplification, other methods of analysis could be used. Displaying which portions of an image a given branch is used could give insight into its function, as could a trace of the value of various memory locations. If certain branches are only used in one or two fitness cases, or are not used on a test set, that could be a clue that the branch is due to some form of overfitting. The same branch could also be tracked from generation to generation, to see the change in both where it is activated and what it does.

Ideas from Evolutionary Computing

Evolutionary Computing has been studied for decades and has developed a number of concepts and techniques that apply to any such system, including this one.

Co-evolution can be a powerful technique for maintaining diversity and creating solutions for problems that require slightly different solutions for different fitness cases, such as the black carpet. The central idea is to use only a subset of fitness cases to evaluate each generation, but to evolve that subset. That is, the fitness of an individual is how well it performs on a set of fitness cases, but each fitness case is assigned a fitness proportional to how *poorly* the individuals of the population do. Thus, as soon as the population starts to converge on a partial solution that does well on some images, the training set changes to include mostly other images. This technique could prove powerful for keeping the diversity up, and making sure that errors are distributed evenly, rather than concentrated in one area.

Another idea is demes. The demes approach considers each individual to be attached to a location. The locations usually form a two dimensional or three dimensional grid, although other topologies are possible. When creating an offspring for a given location, parents are chosen from nearby locations. This generally results in local convergence, with genetic material passed at the boundaries.

Another idea, thought up by the author and not known to be previously discovered or published, derives from the idea that crossover works best between similar individuals, and similar individuals are likely to have similar fitnesses. Specifically, the idea is to choose the first parent using traditional selection techniques, then choose the second to be preferentially close to it. If tournament selection is used, the second tournament could choose, not the individual with the best fitness, but the individual with the fitness closest to the first parent.

Following the ideas in the Ph.D. thesis of Tina Yu [2000], elements of functional programming could be introduced in addition to the existing procedure elements. Other approaches such as object oriented programming could also be examined for ideas.

A strategy to halt runs early could also be developed. It was suggested above to halt runs that were not performing well by generation 20. A generalization is to perhaps start a number of runs, and halt the lowest scoring one after so many generations, the second lowest one a few generations later, etc.

If a lot of computer power was available temporarily, much longer experiments could be run, for example 1,000 or 10,000 generations, to see if any improvement happens. Given the trace of best fitness vs. generation for such runs, it is not hard to calculate the best length to maximize the probability of finding a successful individual.

Other Ideas

With a combination of more powerful evolutionary techniques or simply more computer power, new approaches become possible. Instead of evolving the program directly, a description of a program could be evolved, and before evaluation, the description elaborated into a working program. This is similar to evolution in nature, where living systems go through a long developmental process.

In “Artificial Life and Real Robots” Brooks argued for simultaneously evolving the hardware and software of a robot. He mentioned, for example, that Ghengis had six copies of the same controller, one for each leg, and mutated insects that have extra sections, complete with legs, have the controllers for those legs as well. In the real world, computational elements are also physical elements, are created through

some developmental process, and are ready to interface with other nearby segments. Thinking about this may reveal a way to go beyond genetic programming, to provide another framework for evolving programs.

Such a developmental process could use an alternate representation for programs. A general directed acyclic graph could prove a more natural representation for computation and allow direct implementation in hardware, or the manipulation of logic gates as primitive elements.

Other representations needing only slightly more computer power could soon be explored. For example, at every pixel or small window, we decide what object is there, or at least whether or not it is ground. Windows can get information from their neighbors in semantic network/ neural network fashion, to help propagate constraints. The output could simply be the location of the lowest window that is not "ground." The windows could even overlap. The evolved program would then be working with a low resolution version of the image, but each pixel in the low resolution image contains info about average gradient, etc. over that window. One way of serializing this rather parallel setup is to try to use transformed info from the previous image, i.e. so that the information propagated from neighbors is one frame out of date. Another is to do a grassfire transform-style iteration, where updates happen in raster scan order from upper left to lower right, replacing existing values, then reverse raster-scan order from the lower right to upper left. The iterated rectangle can be seen as a limited version of this, where each window can only know about the window below it.

With more power, the evolution of learning could be explored. Robots could benefit from learning during their lifetime, but evolutionary computation is probably both too computing intensive and too general to be most useful. Instead, learning algorithms that embody some generalities of the environment but must learn its particulars should be used, and the details of those could be decided by an offline evolutionary computation. This has the potential to be a very powerful technique indeed.

Simulating images is a possibility, and could lead to the development of active vision algorithms, as described in previous chapters. The evolution of navigation, map building and the recognition and use of landmarks is could also prove powerful.

Finally, Peter Cariani has discussed the idea of evolving sensors in [2000, *Cybernetic Systems and the Semiotics of Translation*]. These ideas are ripe to be incorporated in the simulated evolution of a perceptually based autonomous mobile robot.

Reflections

As for when I would recommend this technique: Clearly, in the case of obstacle avoidance from vision for a mobile robot, this dissertation demonstrates results comparable to the best hand written systems. For those looking to build a practical vision based obstacle avoiding robot today, to work in a single environment, creating the vision system by hand would take less effort and lead to a greater understanding, while achieving comparable results. Also, for researchers looking for a single evolved program which can work in most environments, a completely new framework is needed. Two or three eight-pixel-wide columns is

simply not enough information for even the human visual system, without some prior knowledge of what ground or wall is likely to look like.

However, for researchers looking to build better robots tomorrow, this technique offers a lot of potential. Many of the possible improvements listed below have the potential for a dramatic increase in performance. The potential benefits over hand creating a system are many. A single setup can be adapted to many environments, by people without much knowledge of robotics. It should be easier to maintain a focus on the general properties of the representation, rather than be distracted by the details. Expanding the representation to allow many different ways of approaching the problem, rather than causing confusion, should allow the genetic algorithm to integrate a number of styles in a way that works well in practice.

In general, machine learning algorithms are best when a researcher doesn't know how to create a system as well as how to define the properties it should have. The next chapter argues the vision, if not all of intelligence, falls into this category.

To those looking to dabble in genetic algorithms, a few things became apparent through the course of this work. First of all, as has been stated already, genetic algorithms are not a magic search technique that can find intricate and subtle solutions with any arbitrary representation. Instead, it is important to make sure the type of algorithms you wish to find are easy to express in your representation. In fact, you would do well to create an individual or two by hand. This is how the seed for the expanded representation experiments started, as a program I sketched by hand to make sure my representation was reasonable and complete.

However, simply being able to express the desired algorithms isn't enough. There must be a path from individuals typical of the initial population, to the type of individuals desired. This may be why so little state was used in the non-seeded runs: before a memory location can affect fitness, it must be both written to and read from. Perhaps, without much evolutionary pressure on the red swath, a feature difficult to distinguish from baseboard without state, that this path was simply too difficult for the GA to find. In other words, genetic algorithms are good a finding an "obvious" solution, when there are so many solutions that seem plausible that we can't possibly search them all. That is, at least in the year 2001, genetic algorithms are good at finding a needle in a haystack, not at creating a better needle.

As for what this work has taught about vision: only that it is both easier and more difficult than had been expected. While images are notoriously noisier than people think, it was a surprise to find that the carpet outside my office, a constant shade of grey, could range in brightness in the image from almost black near the robot to over 2⁄3 of white many meters away. But it was also a surprise to find how easy vision can sometimes be. I had not expected a simple gradient to indicate the boundary of obstacles so well, or for the evolved programs to generalize so easily. Vision does seem to be ripe for learning: many algorithms we think of are simply too naïve, and yet there are simple algorithms that do work, if we can only find them…

10 Philosophy & Manifesto

With the details of the work and the immediate implications behind us, this chapter takes a broad look at Artificial Intelligence, using critiques of it constructively, that is as a list of "do"s and "don't"s for work in AI. The work in this dissertation is seen as the first step in a research program that answers these critiques. In fact, the considerations in this chapter were the motivation for the dissertation.

The plain and simple fact is that perceptual scientists are actually motivated to do things in the laboratory ... because they ... are trying to answer broader issues than the specific empirical questions being asked. ... This is all too often overlooked—these questions may not be explicitly stated, but they are ubiquitous nevertheless. The degree to which one is doing quality science that transcends the mundane collection of empirical measurements is closely associated with the degree to which one understands this fact.

— William Uttal, 1988, *On Seeing Forms*, p. 48

Critiques of Artificial Intelligence

Artificial Intelligence has had many successes, yet the goal of full human intelligence has proven elusive. This section examines the history of AI and critiques of it, in order avoid these mistakes. Many of these critiques are in the area of language understanding, but apply more generally, as explained below.

One of the few critics from within AI itself is Terry Winograd, the author of the early success SHRDLU. SHRDLU was a program for understanding natural language written at the M.I.T. Artificial Intelligence Laboratory in 1968-70. It also controlled a simulated robot arm that operated above the table top and could rearrange and stack the blocks. More impressively, it carried on a simple dialog with a user about a simulated world of children's blocks.

But after studying language understanding in more depth, Winograd came to believe that understanding human languages was impossible for a computer, and left the field. Among other things, he realized that the boundaries of categories in such programs depend very critically on context. For example, Winograd and Flores point out the context dependence of even something as simple as the literal meaning of the word "water," as demonstrated in the following dialog:

> A: Is there any water in the refrigerator?
> B: Yes.
> A: Where? I don't see it.
> B: In the cells of the eggplant.

[Winograd and Flores, 1986, *Understanding Computers and Cognition*, p. 55]

While B is literally correct, this does not help A. Winograd and Flores also point out that even the sense of the word "water" which means "water in its liquid phase in sufficient quantity to act as a fluid" could lead to problems. Consider:

B: Yes, condensed on the bottom of the cooling coils.

This response is just as unhelpful as the one about the eggplants, although the right context could make it appropriate. If A is there to repair the fridge, or to find sources of humidity that ruined some photographic plates, response 1 would be quite helpful.

This is not a purely linguistic distinction. The world is full of regularities, objects and processes that are similar to others in important ways. A computer that needs to find water needs to work very differently depending on whether it is thirsty and looking for a drink, or a refrigerator repair robot looking for symptoms. As above, these could be treated as two different definitions of water, but then we would quickly end up with a different definition for every context. And we would be ignoring the relationships between these different senses of "water." Detecting dew on a leaf has much in common with detecting condensed water on cooling coils.

In other words, the literal meaning of the question is not very useful when trying to get along in the world. The meaning of the query is not composed of the literal meanings of its constituent words in some context free way. And yet, the question is different from "What can I do to satisfy my thirst?" The original question expresses one approach to satisfying thirst, namely finding something to drink, in particular finding water in a refrigerator. So the question "Is there any water in the refrigerator?" expresses the subgoal of a plan, although the subgoal cannot be usefully understood without the original goal.

Many current frameworks in machine learning are intended for pattern classification, that is, given an input, determine which of several pregiven categories it is in. Therefore, such frameworks were not used here. Rather than attempt to spell out exactly how context interacts with regularities in the world to determine useful concepts, the work in this dissertation allows the genetic algorithm to determine this for itself. That is, rather than provide the robot with a set of concepts or categories, the robot is provided with raw sensor data. In interpreting it, the robot is free to decide exactly how it will distinguish between different aspects of the environment. For example, the evolved programs generally execute different branches based on the image and the values of memory registers. These decisions are under control of the genome. Since they are only evaluated in the context of obstacle avoidance, they will in general be task specific. A potentially fruitful extension of this work is to add additional state, so that the context of previous images can be used when interpreting the current image.

Critiques of AI have also come from outside. Philosophy has wrestled with the same issues and even discussed how they apply to AI At its inception, the goal of Artificial Intelligence was to create computer programs capable of the wide range of activities that, in people, are considered to require "thought" or "intelligence." It should therefore come as no surprise that many AI frameworks embody answers to old philosophical problems about thought, the nature of the outside world, and the relationship between the two: the balance between empirical and *a priori* knowledge, the interrelations between abstract and specific knowledge, and so on. Philosophers have debated these questions for a long time, in some cases for millennia, and while none of them have universally accepted answers, there is much discussion on the strengths

and weaknesses of competing ideas. Explicitly considering the relationships between these two fields can bear fruit for both.

Dreyfus and Dreyfus offer one attempt to trace the relationship between Artificial Intelligence and European philosophy. They are particularly interested in Good Old-Fashioned AI or GOFAI, a term they attribute to John Haugeland. For example, they write:

> GOFAI is based on the Cartesian idea that all understanding consists in forming and using appropriate symbolic representations. For Descartes, these representations were complex descriptions built up out of primitive ideas or elements. Kant added the important idea that all concepts are rules for relating such elements, and Frege showed that rules could be formalized so that they could be manipulated without intuition or interpretation. ... AI turned this rationalist vision into a research program and took up the search for primitives and formal rules that captured everyday knowledge.
>
> [From Dreyfus, 1992, *What Computers* Still *Can't Do,* pp. x - xi]

For example, in "On the Art of Combinations" (1666), Leibniz proposed that all reasoning can be reduced to an ordered combination of elements. If we could define such an algebra of thought, it would become possible for a machine to reason, like clockwork. Such a machine would be capable of resolving every philosophical controversy, as well as making discoveries by itself. Leibniz's thesis amounts to a theory of artificial intelligence for the seventeenth century. ... The GPS-style description of reasoning (in terms of simple algebraic symbols and operations that combine these symbols into expressions) directly follows from Leibniz's thoughts and "debugs" them. As far as I know, developers of AI systems have never emphasized just how much their work relies upon and develops related philosophical theories.

— Serge Sharoff, 1995, *Philosophy and Cognitive Science*

However, before AI even existed, philosophers such as Wittgenstein, Maurice Merleau-Ponty and Martin Heidegger reflected on this program and found problems with it. As researchers in AI later discovered, propositions are the wrong representation for common sense, everyday knowledge. There is little evidence that this representation underpins much of human intelligence, and it does not work well in practice.

One reason has already been discussed, namely that the concepts we use are context and task dependent. Another is the problem of how relevant knowledge can be brought to bear in particular situations, to avoid the exponential explosion of combinatorial search. In philosophy this has come to be called the *frame problem.* As Dreyfus explains about all activities that happen inside a room,

> We are skilled at not coping with the dust, unless we are janitors, and not paying attention to whether the windows are open or not, unless it is hot, in which case we know how to do what is appropriate. Our expertise in dealing with rooms determines from moment to moment both what we cope with by using and what we cope with by ignoring (while being ready to use it should an appropriate occasion arise.)
> [*Ibid*. pp. xxviii-xxix.]

Creating a collection of meta-rules to determine what is important in a given situation begs the question. We cannot try all meta-rules

without encountering combinatorial explosions, so how do we determine which meta-rules to try?

AI has traditionally avoided these problems by working in easily definable micro-worlds such as chess, where the set of things that the computer must know about the world is clearly definable. They liken this to the frictionless plane in physics, which simplifies the problem to expose the heart of the matter. The implicit argument is that real world problems are just as clearly definable but much larger. But as Varela *et al*. point out, the boundaries are not clear at all.

One can still single out in this "driving space" discrete items, such as wheels and windows, red lights, and other cars. But unlike the world of chessplaying, movement among objects is not a space that can be said to end neatly at some point. Should the robot pay attention to pedestrians or not? Should it take weather conditions into account? Or the country in which the city is located and its unique driving customs? Such a list of questions could go on forever.

— Varela *et al.*, 1991, *The Embodied Mind*, p. 147

They use the example of an automated car that is to drive within a city [1991, *The Embodied Mind*, p. 147]. Certain things clearly need to be described, such as stop lights and other cars. But do we need to teach it about pedestrians? To distinguish weather conditions and how to take them into account? Or the driving customs of the city in which it is located? The driving world does not have clear boundaries, but rather different factors have varying relevance that blend into the continuous use of common sense and background know-how.

In fact, AI has traditionally assumed the world is organized much like a text adventure game, with a small number of possible actions at each point, which are executed precisely, in a world that has a small number of well defined objects with clear relationships to each other. AI has spent its time creating programs that work in these adventure-game-esq worlds, and assuming that these techniques will work in the real world. But the real world is much "messier" than that. Brian Cantwell Smith makes the same point rather colourfully in the preface to his book *On the Origin of Objects*, quoted at left.

And for better or worse—but mostly, I believe, for worse—the conception of "object" that has been enshrined in present-day science and analytic philosophy, with its presumptive underlying precision and clarity, is more reminiscent of fastidiously cropped hedge rows, carefully weeded rose gardens, and individually labeled decorative trees, than it is of the endless and rough arctic plain, or of a million-ton iceberg midwifed with a deafening crack and splintering spray from a grimy 10,000-year-old ice flow. ...

That is what is profound about gardens: they have to be maintained. *It takes continuous, sometimes violent, and often grueling, work, work that is in general nowhere near as neat as its intended result, to maintain gardens as the kind of thing that they are. ... What more could one ask for, by way of ontological moral?*

— Smith, 1996, *On the Origin of Objects*, pp. viii-ix. Emphasis in the original.

These problems do not apply to AI alone, but to any programming that must model the vagaries and complexities of the real world. That includes such mundane things as employee databases, or even vector image manipulation such as Adobe Illustrator. In fact, as others have pointed out (Smith, Sharoff), Object Oriented programming models real world categories as classes with clearly delineated boundaries and precise relationships to one another. This means the line of enquiry discussed here could set the stage for a new framework for programming in general.

For that reason, this dissertation did not simplify or control the environment, but rather created algorithms for an everyday world that contained a variety of obstacles such as boxes and people. The robot had to decide for itself which visual differences were significant and which insignificant.

If our representational primitives and rules are not context free, then traditional deductive and symbol manipulation approaches do not apply. We need a new way to manipulate our primitives. The context dependent nature of concepts and the problem of relevance are crucial issues which must be addressed in any framework that aims at full, human level intelligence. The importance of these issues argues against physical symbol systems, frames, scripts, schema, propositions, deduction, and planning as central elements of intelligence.

After philosophers pointed this out to AI researchers, some within the field drew the same conclusions. Terry Winograd was discussed above. Philip Agre and David Chapman argue for a minimal representation, coining the phrase "the world is its own best representation." [Chapman, 1991, *Vision, Instruction, and Action*, p. 20] When representations are needed, they argue for "deictic representations," representing things in terms of their relationship to the agent. As Chapman says, "For example, *the-cup-I-am-drinking-from* is the name of an entity, and *the-cup-I-am-drinking-from-is-almost-empty* is the name of an aspect of it. ... It is defined functionally, in terms of the agent's purpose: drinking." [*Ibid*., p. 30]

For this reason, dense or literal representations are eschewed. The representation of programs allows only 5 floating point registers of state. While expanding this, and allowing state from previous images, would allow the use of context, the use of such state should be at the discretion of the genetic algorithm. Out of fear that the problem would be too difficult, the output of the vision system was given a fixed representation, namely the location in the image of traversable space. However, this form of representation is very weak (only a handful of numbers per image), and is much less tradition schemes such as stereo vision. It is also two orders or magnitude less than what was found in the proposal.

Bickhard and Terveen [1996, *Foundational Issues in Artificial Intelligence and Cognitive Science*] support these basic criticisms and add some of their own, faulting much of AI and Cognitive Science for what they term *encodingism*. They focus on how representations inside the computer come to actually refer to states of affairs in the world outside the computer. In brief, they argue that the computer must be *embodied*, and have its representation develop through *interaction* with the outside world.

Others have called for embodiment as well. Rodney Brooks, who also rejects the traditional role of representation, proposes that simulation can simplify away the real problems, and lead to solving irrelevant problems. For example, he points out:

> • We must incrementally build up the capabilities of intelligent systems at each step of the way and thus automatically ensure that the pieces and their interfaces are valid.
> • At each step we should build complete intelligent systems that we let loose in the real world with real sensing and real action. Anything less provides a candidate with which we can delude ourselves.
>
> We have been following the approach and have built a series of autonomous mobile robots. We have reached an unexpected conclusion (**C**) and have a rather radical hypothesis (**H**).

> **C**: When we examine very simple level intelligence we find that explicit representations and models of the world simply get in the way. It turns out to be better to use the world as its own model.
> **H**: Representation is the wrong unit of abstraction in building the bulkiest parts of intelligent systems.
> [Brooks, 1991, *Intelligence Without Representation*]

This suggests that to create intelligence, we need to create programs that confront the messy nature of categories head on, through gaining competency in the real world. In other words, the path to intelligence is through embodied perception. For this reason, this dissertation is concerned with computer vision for a mobile robot.

The Nature of Human Perception

Since the goal is perception, human perception provides an informative precedent. This section discusses what psychology teaches about human perception, in particular, about how people understand the spoken or written word.

Context And Perception

An initial, "obvious" theory is that we identify the letters and spaces (for written text) or phonemes and breaks (for spoken text) individually, then put them together to identify words, then figure out the syntax of the sentence, and finally the semantics. This is known as the "data driven" approach. This approach is so self evident that it may seem necessary — how can one identify a word before identifying the individual letters?

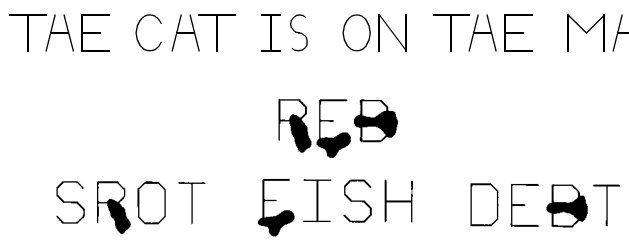


Figure 1: A naïve theory of how we read would have it that letters are identified first, then put together to identify words. But as these examples show, word identity can influence letter identity. Note that in the second line, none of the letters are unambiguous, as is demonstrated in the third line.

There is much evidence, however, that words and letters are identified together. Two famous examples are shown in Figure 1, the first due to Selfridge [1955, *Pattern recognition in modern computers*] and the second from Rumelhart, *et al.* [1986, *Parallel Distributed Processing*], based on Lindsay and Norman [1972, *Human Information Processing*]. In the second example none of the three letters are unambiguous on their own, as is demonstrated by the bottom line. As a third example, we often do not notice spelling mistakes, which evidences the idea that they are corrected early on.

So, lets assume the letter detectors return a set of possible interpretations, and our word identification algorithm tries out all combinations of letters, choosing only those that are real English words. However, written text is not always composed of correctly spelled and rendered words from a predefined list. A spelling mistake which substitutes a completely unrelated letter would foil our system, as would new words, let alone The Jabberwocky. Further, this assumes that we can, before any other processing, unambiguously find the division between letters.

In the realm of spoken language, things are even worse. The length of a pause within words is often longer than that between words [Ash-

craft 1989, *Human Memory and Cognition*, pp. 390-391]. Our perception of foreign languages is closer to the mark: they sound like a continuous stream of babble. But even if we did somehow start by segmenting the string of sounds into separate words, people do not recognize individual words in isolation. Pollack and Pickett [1964, *Intelligibility of excerpts from fluent speech*] recorded several spontaneous conversations, spliced out single words, then played them to subjects. Presented in this way, subjects identified the words correctly only 47% of the time. Success went up as the length of the segment increased.

So how do we recognize words? Consider the following sentences:

I like the joke.
I like to joke.
I like the drive.
I like to drive.

The 'to' or 'the' seems to determine whether the last word is interpreted as a noun or a verb. Indeed, this effect is very strong in English, and can force a word to be interpreted as verb or noun even when it is not usually interpreted that way:

I like to chair.

However, an experiment by Isenberg *et al.* [1980, *A top-down effect on the identification of function words*] showed that the most common category for the last word can influence whether we hear 'to' or 'the' before it. They presented sentences with sounds half-way between 'to' and 'the', and found that when the sentences ended in words like 'joke', which is more commonly a noun, the ambiguous word is interpreted as 'the', whereas words that are usually verbs (like drive) result in hearing 'to'. So, how we identify words does not depend only on physical stimulus, but on their syntactical relations.

In fact, the lowest level of perception—how we identify individual sounds in spoken speech—can depend critically on the highest level—what those words mean. Warren and Warren [1970, *Auditory illusions and confusions*] presented subjects with recordings of sentences. In the recordings, one speech sound was replaced by white noise (a cough works just as well). Here is one set:

It was found that the *eel was on the axle.
It was found that the *eel was on the shoe.
It was found that the *eel was on the orange.
It was found that the *eel was on the table.

In all cases, subjects perceived the "right" sound, the sound that best completed the meaning of the sentence. Apparently, none of the subjects even noticed anything unusual about what they heard.

Even when the utterance is unambiguous, people use semantics to speed up perception. Meyer and Schvaneveldt [1971, *Facilitation in recognizing pairs of words;* also Meyer *et al.* 1975] gave people two strings of letters and asked them to determine, as quickly as possible, whether or not both strings were English words. (Non-words looked like real words and could be pronounced, e.g. "manty", "cabe"). When both strings were words, reactions were faster and more accurate if the words were related (e.g. nurse-doctor) than if they were unrelated (e.g. bread-doctor).

So we do not identify letters or phonemes in isolation; we identify letters and words together. And we do not identify words (and therefore

letters) without also identifying the syntactic and semantic meaning of the sentence.

This argument should not be taken mean that context is somehow more important than the stimulus, or that unexpected stimuli are uninterpretable. Rather, human vision seems to find the best answer given the sensor data, context, domain knowledge, task knowledge, previous state, etc. Whereas traditional computer vision sees vision as "under constrained," it is perhaps better to think of it as "over suggested."

This view that context is as central to perception as the stimulus is the accepted view in psychology. "Context" here includes both "global" elements, such as expectations and domain knowledge, and "local" elements, such as other areas of the image or utterance. Consider this passage from a sophomore textbook:

> After perceiving the first 150 msecs worth of information in the spoken word, we seem to know what word we are hearing. This certainly is based not only on the phonological cues in the spoken word, but also on the context that the earlier part of the sentence has generated. As you hear "Some thieves stole most of the ...," your syntactic and semantic knowledge can specify fairly accurately what kind of word must come next — syntactically, it will probably be a noun or an adjective, semantically it must be some physical thing of enough value to be worth stealing. Thus context, both of syntactic and semantic nature, is having an influence on the process of word recognition. Indeed, such context is even influencing the phonological component, in that the entire phonological representation of a word is not necessary to identify and comprehend the word. [Ashcraft, 1989, p. 429]

[T]here has been much too great an emphasis on local-feature-oriented models of perception rather than theories accentuating the global, Gestalt, holistic attributes of stimulus-forms. In my judgement, all too many perceptual theories have currently fallen victim to the elementalist technological zeitgeist established by neurophysiology and computer science. Although disappointing, this is not surprising, because it is exactly comparable to the way in which theoreticians in this field have been influenced by the pneumatic, hydraulic, horological, and telephonic analogies that have sequentially characterized theories of perception and mind over the last two millennia.

In spite of this fallacious tendency toward features, a considerable body of evidence, only some of which has been surveyed in this volume, suggests that in fact we see holistically, that is, that there is a primary global precedence in human perception. We can, of course, direct our attention and scrutiny to the details of a picture, but we are as often ready to perceptually create details on the basis of some inference as to be influenced by the presence of the real physical details of the stimulus. First-order demonstrations [e.g. visual illusions], usually ignored but always compelling, urge us to consider the global aspects of a form, whereas one has to carefully construct an experimental situation to tease out some semblance of local precedence.

— William Uttal (1988) *On Seeing Forms*, pp. 282- 283

Lessons Learned from Neuroscience

William Uttal, in his review of the neuroscience literature on visual perception [1988, *On Seeing Forms*], concludes that visual illusions are persuasive evidence against models that emphasize local features, and for the use of perceptual context. He argues that it is now clear that classic Gestalt psychology was correct in its emphasis on global, holistic analysis.

He also points out that when we perceive hue, we are not simply recovering objective properties of the stimulus (the wavelength of light). What we are doing involves much more interpretation than that. In fact, virtually any hue can be associated with almost any wavelength by manipulating the context [Land, 1977, 1983; Land & McCann, 1971].

Integration of Depth Cues

A final point, this one about depth cues. People do not use a single cue to determine depth, but integrate a number of cues. These include binocular stereo, texture gradients, linear perspective, relative size, occlusion, focus, support and optical flow; some examples are show in Figure 2. While all of these are individually fallible, together they work pretty well.

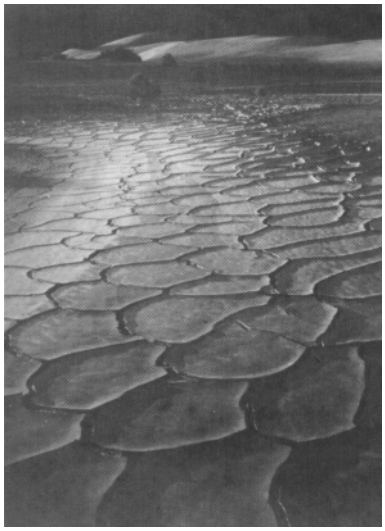
There is often a tendency, among roboticists, to consider vision synonymous with stereo, so let me point out that in human perception there are many times when stereo is not even used. For example, if we cover one eye we can still tell which objects are close to us and which far away. And when we watch movies, stereo is not giving us no information, it is giving us wrong information with complete certainty. Stereo tells us that everything is at the same depth (that of the screen), yet we can still perceive majestic landscapes stretching off into the distance, or that Tyrannosaurus Rex lunging at the Brontosaurus. Even when we are looking at a still image (which lacks any motion cues), we have a good idea of how far away the different parts of the scene are.

With this analysis in hand, I now turn to a critique of current methods in robot design.

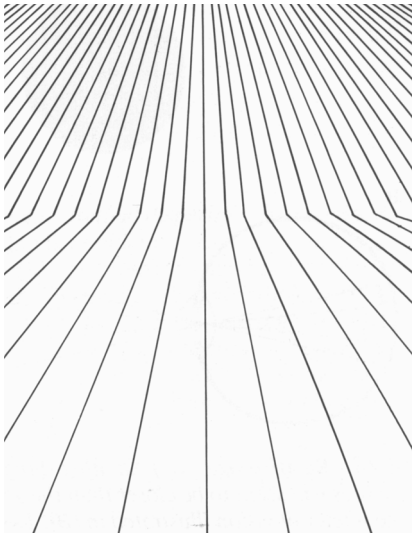
The Traditional Approach to Robot Perception

In contrast to human perception, most robot visual perception uses a single depth cue and is completely data driven. For example, the most common techniques for recovering depth information are stereo matching and optical flow. They both rely on finding the same object in two or more images, and they do this by template matching between small windows in each image. There are many situations where this does not work or is misled. The three most common are areas of low texture (e.g. a flat grey wall); fence post errors, where a feature repeats through the environment (e.g. a fence or row of file cabinets); and the halo effect where one object occludes another.

Perhaps the most explicit statement of this traditional view is still given by David Marr and H. Keith Nishihara [1978, *Visual Information Processing*]. They state “the problem commences with a large, gray-level intensity array, ... and it culminates in a *description* that depends on that array, and on the purpose that the viewer brings to it.” The role of expectations and domain knowledge are at best down played by this description, and at worst absent. Marr and Nishihara point to constraints such as continuity or rigidness as the basis of a theory of vision, because they are universal constraints, that is, independent of a particular domain or task. Yet, such algorithms work only in limited domains, because those constraints are *not* universal. For example, most scenes contain discontinuities at object boundaries, the ultimate discontinuity. There is little or nothing that is truly universal in perception.



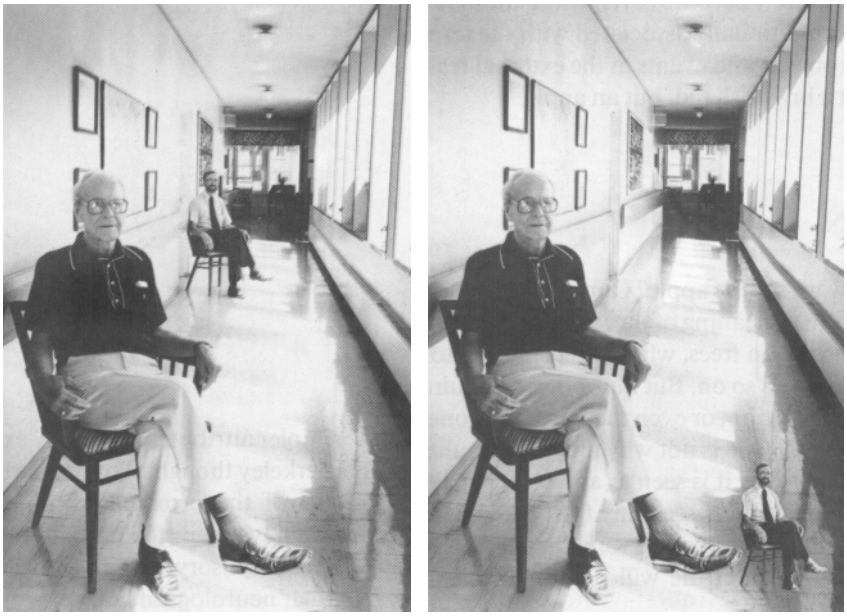
(A) Texture gradient Uniformly textured surfaces produce texture gradients that provide information about depth, for example, in the mud flats of Death Valley.



(B) Changes in texture gradients Such changes can be cues to corners, occlusions and other phenomena.

(C) Occlusion

(D) Linear perspective



(E) Perceived size and distance The 2D size of the men in the image—which corresponds to the size of their retinal image—is in the ratio of 3 to 1. But in the left image, they look roughly equal in size, with one about three times further off than the other. In the right image, however, the smaller man appears to be at the same distance as the other man, and one third the size.

Figure 2: An assortment of depth cues.

Marr and Nishihara think of vision as the inverse of computer graphics: given an image, and a model of the physical processes that went into producing it, what scene was rendered? However, to be fast and robust in the face of noise and other problems, the human visual system often "fills in the gaps" (think of the phoneme restoration effect above, or the subjective contours in Figure 3). Our edge detectors are not simply recovering objective properties of the image or the world, but using the image as one clue, together with much other information. Or consider that the colour of an object (as we perceive it) can change when we put a different colour next to it.

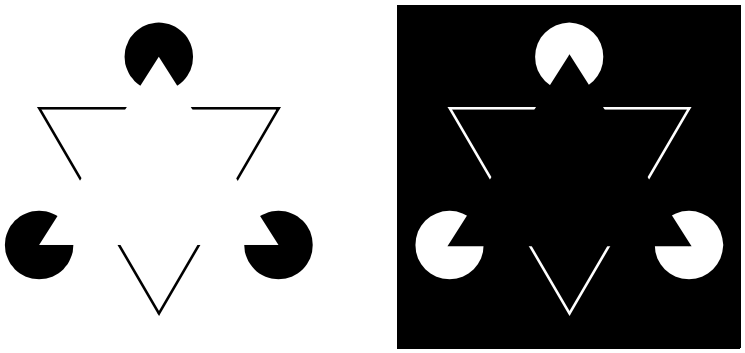


Figure 3: Subjective contours. On the left, we see a white triangle whose vertices lie on top of the three black circles. The three sides of this white triangle (which looks brighter than the white background) are clearly visible, even though they do not exist physically. On the right, black and white are reversed. Here, there is a black triangle (which looks blacker than the black background) with subjective black contours. [Kanizsa, 1976**]**

To say that people are bad at determining colour or intensity gradients misses the point. Much of the evidence that vision is bottom up comes from experiments where all context is eliminated. Whether it is anesthetized cats starring at line segments or people staring at triangles and circles, we use bottom up processing because it is all we have left. But in more normal circumstances there is a wealth of extra information we bring to bear. Whereas Marr and Nishihara think in terms of the vision problem being under constrained and what constraints to add to it, it seems more helpful to look at it as "over suggested," and ask how we can best take all our information into account.

As the Related Work chapter made clear, obstacle avoidance based on bottom up perception does not work reliably for long periods of time. Even those researchers searching for constraints recognize the potential benefits of including some contextual elements in perception, yet still choose not to investigate that path.

It is now known, for example, that there is not even any unique relationship between such "simple" and "directly" related parameters as perceived hue and stimulating wavelength (Land, 1977, 1983; Land & McCann, 1971). Rather, virtually any hue can be associated with almost any wavelength if one is given control over the spatial and temporal environment of the stimulus. In short, the causal relationships between stimuli and visual responses must be considered to be very loose; at the very least, multidimensionally determined; at the very worst, only fortuitously correlated by the very complex processes underlying what we glibly call illusion and perceptual construction.

— William Uttal (1988) On Seeing Forms, p. xv

Why not? The main reason is complexity; a system that takes into account its context would be considerably more complex than a data driven pipeline system. To manage complexity, researchers decompose intelligence into a number of smaller, independent problems (e.g. find edges given only the image), which are individually tractable. The particulars of how scientists manage complexity shed light on both the limitations of current robotics and the possibilities of circumventing them.

Black Box Systems

There are many concessions to complexity that researchers make, but two in particular concern us here. The first results from specialization,

in which each researcher focuses on a different subsystem of a robot. Examples of specialties include perception, planning and position estimation. Robots, after all, are complex systems and by specializing in how to model, analyze and build in one area, researchers can explore that area in greater depth. But at the same time, researchers want their approach to be usable by many different people, in many different areas. So they make them as independent of the other subsystems as possible, and usable for as many tasks as possible. Thus, robot software is composed of a number of black boxes.

In this dissertation, I intend the phrase "black box" to be a technical term to describe the components of such a design. The essence of a black box is that the task it performs and how it is used can be described succinctly, and that as long as one has this description, there is no need to know how it works.

A few clarifications are in order. To say that its function and use can be described succinctly, I mean that in order for someone outside that subfield to use it, they do not need to know as much as the implementers did, let alone the designers. For example, a canonical black box is the microprocessor. Although microprocessors are quite complex, those who design and build PCs do not need to understand this complexity; all they need to understand is the interface: the power pins, the address, data and control pins, and so on. And the interface to a black box is often described using new concepts or abstractions. Of fundamental importance to the microprocessor are the concepts of the *instruction*, the *operand*, various *addressing modes* and the like. Together these form the abstraction of the machine language.

When a black box does not conform to its interface, we declare it broken, as with the bugs in floating point divisions on early Pentiums. If we can describe under what conditions the error happens and what goes wrong, we can roll that into the original description and work around it. To keep the description simple, we usually give a superset of the error conditions (e.g. "during an `fdiv` statement", even though only certain numbers caused problems), and a bound on the error (e.g. produces accuracies as bad as four decimal digits). If this is not possible, we throw it out and get a new one. When we design and build systems using a black box, all we use is its description, possibly updated in the face of such errors. If an error is discovered in one subsystem, all bets are off and in general no one knows what the system as a whole will do. This is why people wanted new Pentiums; they had no idea what the ramifications of the fdiv bug were for the software they were using, and for software makers to prove that errors would be small would be a huge task. The simplification accomplished by black box decomposition is huge; even a small deviation from the spec leads to a great ballooning of complexity.

Phoebe Sengers makes a similar point in her Ph.D. Thesis [1998, *Anti-Boxology: Agent Design in Cultural Context*]. Her interest is in AI for characters in interactive narratives. As she points out, such characters are created from a set of independent behaviours. Unfortunately, this typically leads to a lack of global coherence, which Sengers traces to a strategy called atomization that AI shares with industrialization and psychiatric institutionalization. She too argues that while this strategy is essential for building understandable code, it is fatal for creating agents that have the overall coherence we have come to associate with living beings.

This dissertation agrees with Sengers about the nature and causes of "boxology," and the problems it can create. While the boxes in her area are externally distinguishable agent states—behaviors—in this dissertation they are subsystems of a robot. Thus both Sengers and this author arrive at the same high level conclusion, albeit in different domains.

The Interaction/Complexity Trade-off

Comparing this with the important of context in human perception above, the difference is apparent. Those who are interested in how words get their meaning are in a separate group from those who study the shape of letters and optics, so they create separate subsystems. For example, to make them usable by each other, the latter might create programs that use camera images as input and ASCII as output, and the former might accept ASCII as input. Once the first stage decides on a word's identity, that choice is never revisited.

But taken as a description of the field, that is oversimplified. The black box framework does not *require* that the intermediate representation be a single ASCII character for each letter seen. Instead, for example, it could return a list of letters with associated probabilities. For the character between A and H, it could return A and H, each with a probability of 0.5. In general, a black box in a perception program could return a "short list" of possibilities with associated probabilities.

But even so, this makes for slower perception. It cannot account for the use of semantic expectations to speed up recognition (see the "nurse-doctor" "bread-doctor" example above). More importantly, it means that errors cannot straddle black box boundaries. The system can never parse a situation where the physical stimulus is not all that close and neither is the syntax, but together there is only one consistent solution. In other words, if each subsystem only provides a weak constraint—and in general, enumerating all the possible interpretations that satisfy the constraint would be far too gigantic a task—the "list of possibilities" representation is impractical.

Hard Separation vs. Soft Separation

The next fall back position is for each black box to provide constraints in some constraint language. But the most natural language for such constraints use the concepts from the analysis at that stage—the very concepts that need to be contained within the black box, to manage complexity. For example, even if we cannot tell whether a given shape is an "R" or a "B," we *can* tell that the top is round, the left side is straight, and there is definitely something in the lower right. More importantly, any non-trivial analysis reduces information. And without knowledge of syntactic, semantic, task and domain constraints, we cannot be sure what is relevant and what we can safely throw out.

There have been attempts at frameworks for stronger interaction, such as blackboard architectures, recurrent neural nets, subsumption or the current interest in Bayesian frameworks. However, in order for these techniques to become widespread, they must work with a large number of techniques on many and varied subproblems. Therefore, the flavour of representation and interaction tends to be largely context and domain independent. In other words, the social environment in which theories are created and discussed provides additional constraints on those theories.

What characterizes the black box approach is *hard separation*. Since each box performs a qualitatively different task and has a simplified external description, there simply is not much room for subtle interaction between two boxes. Once we have decided on the structure of the system—how we will break it down into subsystems—the subsystems become fairly isolated. As a result, the structure we impose is rather rigid.

This is not to say that there should be no separation, or that there is no separation in the human brain. Certainly the human perceptual system is composed of a number of subsystems, physically located in different parts of the brain. However, these subsystems work fundamentally more closely than the subsystems in traditional robots. Using syntactic and semantic constraints in the word recognition subsystem is not possible in a system with hard separation; I refer to such interplay as *soft separation.*

This is also not to say that any old increase in communication between the various subsystems is good. Simply allowing one subsystem to read the internals of another subsystem is not enough; it needs to know what to do with that information.

It should be noted that the hard separation/black box approach applies not only robotics, but to most of Western science and engineering, if not other avenues of thought. We build our machines to have predictable and easily describable behavior. The connotations of the word "mechanical" come from this. Anyone who has worked with machines knows that naturally they shake and bend, that electronics is susceptible to noise, and that much effort is expended on finding rigid materials, precision machining and linear amplifiers. Viewing other endeavours in this light may lead to fruitful insights. However, in this work I am interested in its impact on robotics alone.

Software Engineering: Friend or Foe?

A last note on the role of complexity: It could be said that software engineering teaches the power of keeping code conceptually clean and understandable. I would argue that representations that make useful programs easier to express are directly applicable to the proposed framework. But other aspects are concessions to human understanding and maintainability of code. These aspects represent objects in the world using context free classes with simple, context free relationships to each other, and that make black boxes much easier to construct than interconnected systems. Software engineering makes the point that those help greatly when people construct programs. This work makes the point that we will need to overcome them to construct truly intelligent computers.

A Final Example: Depth Cues

Let's see how this plays out in the case of using multiple depth cues. Earlier we noted that people combine a number of depth cues, and it is unclear whether a single depth cue is sufficient for even simple tasks such as obstacle avoidance. Certainly researchers recognize the benefits of combining depth cues. In fact, it can be considered an example of the larger problem of sensor fusion. Why are not more people trying it?

By now, the answer should be clear: the problem is isolation and complexity. Traditionally, for each depth cue, we analyze an image and

return a depth map. It might make more sense to return a set of constraints, but what language would we use to describe them? We could also try combining the depth maps, but again how do we do that? Answers like “take the average” or “use the most common” are typical data driven answers, because they are optimal assuming that the errors of each method are independent of each other, independent from pixel to pixel, independent from one moment to the next, independent of the geometrical arrangement of objects in the environment, and in general independent of all other variables in the robot, its environment and the task.

A New Hope

So what would an alternate approach look like? As mentioned above, it could still have subsystems, but they would be much more interdependent than current subsystems. Such a system may not have a central organizing framework. Each group of subsystems may interact in some ad hoc manner that is most appropriate for them.

However, this does not mean there would be total chaos. Regularities would most likely abound, because many algorithms or data structures work well in a variety of situations. Once we back off from the idea that “one size fits all,” we are likely to find that one size fits many.

This is certainly true in biology. Animals have a large amount of structure, with only a small number of basic cell types for example. And brains definitely have structure, since different functions are located in different areas of the brain. Nevertheless, similar structures are used in many places, independently of each other, because they work well. D’Arcy Thompson [1917, *On Growth and Form*] is one of many to point this out.

Another interesting property of an alternate approach, perhaps the most important property, is that it could include the best of many other approaches. When a framework specifies that an interaction must happen a certain way, it *excludes* it from happening any other way. Subsumption, for example, specifies the highest level structure of an agent. It specifies discrete subsystems that interact through though virtual wires that carry simple values, often just real numbers. Subsystems at higher levels can only subsume the output of lower level subsystems, that is, disable the lower level subsystem’s output and replace it with its own.

This excludes other organizations, other ways for subsystems to interact. Such exclusions often mean the system cannot embody some ability that people have.

By contrast, what is argued for here is an *inclusive* framework, that allows subsystems to use whatever works best. It may use a technique or representation in many places, but allow exceptions where another technique would work better. It should allow Bayesian methods where they work best, subsumption where it works best, and any other method where it works best. This, perhaps, is the most important aspect of what is attempted in this work.

And in the end, this is what experienced engineers do. They know the strengths and weaknesses of a large number of techniques, and select the most appropriate one for every subproblem. While they often look for an overall design that will avoid the majority of problems, they still

need to occasionally go outside of it to “hack around” the occasional problem.

Summary

The problems and limitations of current robot perception algorithms come from using the traditional method of managing complexity: the *black box* methodological stance. This stance, which has been wildly successful in science and engineering, works by isolating subsystems so that the description of what they do and how to use them is much simpler than how they do it. Also, those who use them do not need to know about any of the issues of how they are designed. Although this is a great win from the complexity and division of labour standpoint, it makes for much isolation between subsystems. I call this type of isolation *hard separation.*

In contrast, to get the most information out of a sensor’s readings, the human perception system can use context*:* task and domain knowledge influence even the lowest levels of perception, and all levels influence each other. However, a perceptual system in which each subsystem can take into account the details of other components cannot have hard separation between its components. This is not to say it must be one undifferentiated mass; there should no doubt be identifiable subsystems, but these subsystems need to interact in subtle ways. I term this type of interaction *soft separation.*

Marr and Nishihara reflect the traditional view in all of robotics when they take hard separation as the fundamental tenet of computer vision. In their view, research in computer vision should not worry too much about the issues in other subfields of AI, but instead focus on finding universal constraints for this under constrained problem. They naturally pick only a single set of related issues to study, which they can study in great depth. And they search for techniques that are as generally applicable as possible. In this way they are lured into creating black boxes that are hard separated from other subfields’ boxes, and from task and domain constraints. In fact, it is hard to see how else the problem could be attacked.

The principles enunciated in this chapter, that the guide the development of an alternative framework in this dissertation, are summarized here:

1. **Context dependence of concepts:** The world cannot be usefully seen as composed of discrete elements and task. In practice, the vast majority of useful concepts are dependent on context.
2. **Relevance (the frame problem):** The system must decide for itself what part of the world around it, and what part of its knowledge, is relevant. Micro-worlds are poor testbeds because the researcher solves this problem. Many real world domains do not have clear boundaries, but rather, different factors have varying relevance that blend into the continuous use of common sense and background know-how.
3. **Use representation sparingly:** Representations can fail to represent what is relevant, can get out of sync easily, and often are not needed anyway. The world is often its own best representation.

4. **Embodiment:** Simulations necessarily embody assumptions about how the world works, in particular the nature of concepts and what is relevant. Historically, this has lead AI researchers to solve the wrong problems. Therefore, intelligence is more likely to be developed by dealing more and more competently with the vagaries and complexities of the real world, than by creating human level competency in ever more complex micro-worlds. Embodiment is also important in another sense: the body mediates how an agent views and affects the world, which greatly affects how it things. When all you have is a hammer, everything starts to look like a nail.

5. **Guide the agent:** The goal of perception is not to recover objective properties of the stimulus or even the object being perceived. It is to understand enough of the world to guide the agent in its task.

6. **Context dependence of analysis:** Any analysis or decision, such as identifying object boundaries, emphasizes some hypotheses over others. Such decisions must take into account constraints from other subsystems, as well as task and domain constraints. In other words, the various subsystems that compose a system need more interaction.

7. **Soft constraints:** There are many cues in an image that, while they are not hard and fast constraints, suggest various interpretations. The human perceptual system leverages those, and computers can stand to gain much by leveraging them as well. Perception is usefully seen as over-suggested, not under-constrained.

8. **Complexity:** Researchers largely agree on the above two points, but they do not incorporate them into their systems, instead creating systems out of black boxes. They use black boxes because otherwise the system would become too complex for them to work with.

9. **Regularities:** The most natural framework for a thinking device may not have a central organizing principle, but instead have elements or approaches that are used many times independently.

10. **Influences from research culture:** The previous four points are examples of a general point. The kinds of programs and frameworks that are created by people are influenced by the affordances of the human mind. These influences may come from universal, innate qualities of the mind, from social influences, or—most likely—both.

11. **Inclusiveness:** Frameworks should allow each design problem, including the interaction between subsystems, to be solved with whatever technique is most appropriate.

The first four (context dependence of concepts, relevance, use representation sparingly, embodiment) have been widely discussed in critiques of AI. The next three (guide the agent, context dependence of analysis, soft constraints) are generally understood in psychology and occasionally acknowledged in AI, where they are greeted with lip service, apathy or hand-wringing. The last four (complexity, regularities, influences from research culture, inclusiveness) have been discussed to various degrees in science studies, cultural studies and elsewhere, but have not been seriously discussed within psychology or artificial intelligence

anywhere to my knowledge. Unless practitioners in these fields are aware of and discuss these issues, they will not take these ideas to heart and change their practices.

The last seven points (and the discussion about them) are the contribution of this thesis in the area of the debate about AI.

The Proposed Research Program

A black box structure with strong separation appears necessary for managing complexity when people design systems. Therefore, we need to develop techniques for designing systems that are more interdependent than what people can design. In other words, we need to hand at least part of the design process over to software tools. Our problem then becomes the meta problem: how do we design tools that design robot software?

What could those tools be? How can they design fundamentally, qualitatively different systems, on par with nature? Since nature has been so successful, let's take a page from its approach to design. The fundamental difference is that nature starts with a number of designs, chooses the ones that work best, and combines them in hopes of getting the best of both or even something new. In other words, it uses learning.

Although learning is by no means new to robotics, it is not usually applied to designing architectures. Most applications of learning start with a decomposition into subsystems. This usually done by hand, in a black box way, and only the internals of the boxes are learned.

If evolutionary computation or some other learning technique is used to create the design, the subsystems can be tuned to the particular task and environment. Whereas a technique or theory of vision strives to be generally applicable, a technique or theory of vision *design* can be general, while the designs it creates are specialized. This also allows the individual subsystems to be more interdependent. Besides not having to be "plug and play" in other designs, they are designed together, so they can learn to compensate for each others weaknesses. The design does not need to have a conceptually clean organizational structure, or be simple enough to be understandable by a programmer. This is not to say that anything goes or that it can be arbitrarily complex. As pointed out above, animals have a wealth of structure. There are only a small number of basic cell types, for example, and different functions are located in different areas of the brain. But the separation between parts can be a soft separation.

Above I argued that hard constraints work most of the time, but fail every once in a while. This makes them perfect for a learning algorithm. Hopefully, a simple, general rule will be easy to find. Once found, it is likely to be kept because of its great boost to performance. Then, the cases where it fails will be noted and exceptions made for them. This kind of fitness landscape, where there is a nearby approximate solution that can be further refined, is the most conducive to learning.

This process, known as *iterative design*, is what researchers already do by hand when developing a new technique. After all, techniques and frameworks are the main contributions of AI, and in order to demonstrate and evaluate a new ideas they must be embodied in working programs. Systems never work the first time, and reflecting on what went wrong and why provides feedback. Sometimes the reflection shows

mistakes during implementation, but other times is shows that the technique or framework was not applied in just the right way.

The simulated evolution approach described here can be seen as automating this step[1], although the details are different. In particular, when constructing systems by hand, only a small number are implemented, and the researchers spend a fair amount of time analyzing and understanding how the program works and deciding what to try next. The above critique indicates the biases and limitations of that approach. By contrast, evolutionary techniques use a much simpler method of generating new designs, and try many more, typically hundreds of thousands to millions.

This gives the research a different flavour. The researcher focuses on the representation. It is seductive to treat evolutionary computation (EC) as a black box that automatically finds subtle and intricate uses of the building blocks provided. But after working with EC for a while, one realizes that this is not so. If the representation has a certain flavour, such as data driven or contextually dependent, it will be difficult if not impossible for EC to develop strategies in a different flavour.

This makes the representation even more important. As the machine learning community has discovered, representation is king. Still, EC will compose the building blocks in complex ways, taking care of many details, freeing us from the constraint of a conceptually clean structure. And the framework is naturally inclusive. Unless the representation explicitly forbids it, different parts of the program will use whatever structure happens to work. New approaches, for example active vision or conceptual driven elements, can be incorporated by expanding the representation without losing other approaches. It is similar to the programming language Lisp which, while known as a functional language, actually contains both functional and procedural elements, allowing the programmer to mix and match styles as appropriate.

The main research question is how to choose a representation that facilitates the EC search. Secondary questions include how to assign fitness to potential programs and how those programs are used. Those are at least partly perception and robotics issues; obviously there is room for exploring way to modify the EC that are independent of any domain. But it is also possible to leave those explorations to the EC community, and simply acquire a knowledge of the strengths and weaknesses of EC. That is the path I have travelled in this dissertation.

The methodology works as follows: The researcher picks a task, considers potential algorithms for that task, and comes up with a representation for those algorithms. They then come up with a particular instance of that task, select the specific inputs and outputs for potential programs, decide how to assign fitness, then run the EC.

By examining how the best programs perform, where they succeed and why, and by looking at how various primitives are used or not used, researchers reflect on the representation and the other details and make improvements. Improvements can also be inspired by the successes, failures and trends in traditional research. For example, Tina Yu's Ph.D. thesis found that functional programming elements helped Genetic Programming [2000, *An Analysis of the Impact of Functional Programming*

1. I am indebted to Ian Horswill for this observation.

Techniques on Genetic Programming]. Certain software engineering elements could prove useful as well.

To elaborate, when considering a particular task in robotics or computer vision, it is usually the case that we can write out a dozen or more potential programs in pseudo code. The goal of the initial representation is to express all of these programs, and all obvious generalizations and combinations of them, naturally and simply.

And although evolved programs generally will not be understood line by line, the way traditional programs are, many useful principles of their functioning can still be determined. We can determine under which situations they work and which they do not, we can see the values achieved by different parts in different situations, and we can explore small combinations of elements that commonly repeat. These types of analysis are more like those in neuroscience than traditional computer science.

As such, the level of understanding and analysis is somewhat lower than the norm in computer science. A sorting algorithm, for example, is generally proven to always get the right answer before anyone will use it. Simply saying "It was tried on a thousand lists and sorted them all properly" is not generally considered strong enough evidence that it will work.

However, this level of proof is simply not possible when interfacing with the real world, as is done in robotics. Edge detectors, for example, cannot be proven to find all the edges in an image. As such, robotics is typically full of optimal algorithms that do not work well in practice, since the conditions for optimality are rarely met. What can be done is to test it well and gather empirical data to establish statistical significance. And in practice this is more than good enough. If a vacuum cleaning robot works well in a laboratory, then in a home, then in many homes, that is more than enough of an argument that it works.

This leaves mathematical proofs with a secondary role in robotics, that of proving under what conditions an algorithm will and will not work. This can guide researchers in what to look for when the algorithm is failing. In this role it is simply one analysis technique among many. Much of machine learning trades off this ability in preference to other ways of understanding, or other properties altogether.

Therefore, the contribution in this approach is not that different than in traditional approaches. We still discover which representations and classes of algorithms work best in which situations, and that both guides us in engineering systems for particular problems, as well as teaches us about the nature of the problem domain.

The rest of this dissertation describes the first steps in this research program.

References

Activision, Inc. (2000). *Star Trek: Armada* [computer software]. www.planetSTArmada.com

Agre, P. E. (1997). *Computation and Human Experience.* Cambridge, UK: Cambridge University Press.

Anderson, S.L. (1990). Random Number Generators on Vector Supercomputers and Other Advanced Architectures. *SIAM Review* 32(2): pp. 221-251

Ashcraft, M. H. (1989). *Human Memory and Cognition.* Scott, Foresman and Company.

Baluja, S. (1996). Evolution of an Artificial Neural Network Based Autonomous Land Vehicle Controller. *IEEE Transactions on Systems, Man and Cybernetics Part B: Cybernetics.* 26, 3, pp 450-463.

Bickhard, M. H. and Terveen, Loren (1996). *Foundational Issues in Artificial Intelligence and Cognitive Science: Impasse and Solution.* Amsterdam: Elsevier.

Blackwell, Mike (1991). The Uranus mobile robot. *Carnegie Mellon University Robotics Institute Technical Report* CMU-RI-TR-91-06.

Brooks, Rodney (1991). Intelligence Without Representation. *Artificial Intelligence Journal*, 47, 1991, pp. 139-160.

Brooks, Rodney (1992). Artificial Life and Real Robots. *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life,* Varela and Bourgine (eds). Cambridge, Mass.: MIT Press. pp. 3-10.

Camus, T., Coombs, D., Herman, M. and Hong, H.T. (1999). Real-time Single-workstation Obstacle Avoidance Using Only Wide-field Flow Divergence. *Journal of Computer Vision Research,* Summer 1999, Vol. 1, No. 3, MIT Press

Cramer, N.L. (1985). A representation for the adaptive generation of simple sequential programs. *Proc. of an International Conference on Genetic Algorithms and their Applications.* Erlbaum.

Coombs, D., Herman, M., Hong, T.H. and Nashman, M. (1997) Real-time Obstacle Avoidance Using Central Flow Divergence and Peripheral Flow. *IEEE Transactions on Robotics and Automation.*

Chapman, David (1991). *Vision, Instruction, and Action.* Cambridge, Mass.: MIT Press.

Davies, E.R. (1997). *Machine Vision: Theory, Algorithms, Practicalities,* 2nd Edition. San Diego: Academic Press.

Dreyfus, Hubert (1992). *What Computers* Still *Can't Do: A Critique of Artificial Reason* Cambridge, Mass.: The MIT Press.

Floreano, D. and Mondada, F. (1994). Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural-Network Driven Robot. *Proceedings of the Third International Conference on Simulation of Adaptive Behavior: From Animals to Animats 3.*

Fogel, L. J., Owens, A. J., and Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution.* New York: John Wiley.

Friedberg, R. M. (l958). A learning machine: Part I. IBM Journal of Research and Development, 2(1) 2-13.

Friedberg, R. M., Dunham, B., and North, J. H. (l959). A learning machine: Part II. IBM Journal of Research and Development, 3(3) 282-287.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems.* Ann Arbor, MI: University of Michigan Press. 2nd ed: MIT Press, 1992.

Harvey, I., Husbands, P., and Cliff, D. (1992). Issues in Evolutionary Robotics. *Proceedings of SAB92, the Second International Conference on Simulation of Adaptive Behavior,* Meyers, J.-A., Roitblat, H., and Wilson, S. (eds). Cambridge, Mass.: The MIT Press

Harvey, I., Husbands, P., Cliff, D., Thompson, A. and Jakobi, N. (1997). Evolutionary Robotics: The Sussex approach. *Robotics and Autonomous Systems*, 20. pp. 205-224.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor.

Horswill, I. (1993). Polly: A Vision-Based Artificial Agent. *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, July 11-15, 1993, Washington DC, MIT Press.

Horswill, I. (1994). *Specialization of Perceptual Processes.* Ph.D. Thesis, Massachusetts Institute of Technology, May 1993.

Husbands, P. and Meyer, J.-A. (eds.) (1998). *Evolutionary Robotics, Proceedings, First Euopean Workshop, EvoRobot98.* Paris, France, April 1998.

Isenberg, D.; Walker, E. C. T.; Ryder, J. M. & Schweikert, J. (1980). *A top-down effect on the identification of function words.* Paper presented at the Acoustical Society of America, Los Angeles, November 1980. Quoted in Rumelhart *et al.* 1986, *Parallel Distributed Processing* Vol. 1.

Jakobi, N., Husbands, P. and Harvey, I. (1995). Noise and the Reality Gap: The Use of Simulation in Evolutionary Robotics. In *Advances in Artificial Life: Proceedings of the Third European Conference on Artificial Life.*

Kanizsa, G. (1976). Subjective Contours. *Scientific American* 234: pp. 48-52.

Katragadda, L., Murphy, J. Apostolopoulos, D., Bapna, D. & Whittaker, W. (1996). Technology Demonstrations for a Lunar Rover Expedition. *1996 SAE International Conference on Environmental Systems,* Monterey, CA.

Keith, M.J. and Martin, M.C. (1994). Genetic Programming in C++: Implementation Issues. *Advances in Genetic Programmming*. Cambridge, Mass.: The MIT Press.

Kinnear, K. E. (1994a). A Perspective on the Work in this Book. *in* [Kinnear 1994b]

Kinnear, K. E. (ed). (1994b). *Advances in Geneti cProgramming.* Cambridge, Mass.: The MIT Press.

Koza, J. R. (1991). Evolving Emergent Wall Following Robotics Behavior Using the Genetic Programming Paradigm. *ECAL,* Paris, Dec. 1991.

Koza, J. R. (1992). *Genetic Programming: On The Programming Of Computers By Means Of Natural Selection*. Cambridge: MIT Press.

Koza, J.R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge: MIT Press.

Krotkov, E., Hebert, M. & Simmons, R. (1995). Stereo perception and dead reckoning for a prototype lunar rover. *Autonomous Robots* 2(4) December 1995, pp. 313-331

Land, E. H. (1977). The retina theory of color vision. *Scientific American, 237,* pp. 108-128

Land, E. H. (1983). Recent advances in retinex theory and some implications for corical computations: Color vision and the natural image. *Proceedings of the National Academy of Science* (USA), *80,* pp. 5163-5169

Land, E. H. & McCann, J. J. (1971). Lightness and retinex theory, *Journal of the Optical Society of America, 61,* pp. 1-11.

Langton, C. G. (ed). (1987). *Proceedings of Artificial Life.* Addison-Wesley.

Langton, C. G., Taylor, C., Farmer, J. D. and Rassmuseen, S. (eds). *Proceedings of Artificial Life II.* Addison-Wesley.

Lenat, Douglas & Guha, R.V. (1990). *Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project.* Reading, Mass.: Addison Wesley.

Lorigo, L. M. (1996). *Visually-guided obstacle avoidance in unstructured environments.* MIT AI Laboratory Masters Thesis. February 1996.

Lorigo, L. M., Brooks, R. A. & Grimson, W. E. L. (1997). Visually-guided obstacle avoidance in unstructured environments. *IEEE Conference on Intelligent Robots and Systems* September 1997.

Lindsay, P. H., & Norman, C. A. (1972). *Human information processing: An introduction to psychology.* New York: Academic Press.

Maimone, M. (1997). *Lunar Rover Navigation 1996* (Web Page). `http://www.cs.cmu.edu/afs/cs/project/lri/www/nav96.shtml`

Matthies, L. H. (1992). Stereo vision for planetary rovers: stochastic modeling to near real-time implementation. *International Journal of Computer Vision,* 8(1):71-91, July 1992.

Matthies, L. H., Kelly, A., & Litwin, T. (1995). *Obstacle Detection for Unmanned Ground Vehicles: A Progress Report.*

Meeden, L.A. (1996). An Incremental Approach to Developing Intelligent Neural Network Controllers for Robots. *IEEE Transactions of Systems, Man and Cybernetics Part B: Cybernetics*. 26, 3, 474-485.

Meyer, D. E. & Schvaneveldt, R. W. (1971). Facilitation in recognizing pairs of words: Evidence of a dependence between retrieval operations. *Journal of Experimental Psychology, 90,* pp. 227-234

Meyer, D. E.; Schvaneveldt, R. W. & Ruddy, M. G. (1975). Loci of contextual effects on visual word-recognition. In P. M. A. Rabbitt, & S. Dornic (Eds.), *Attention and Performance* (Vol. 5, pp. 98-115). London: Academic Press.

Miglino, O., Lund, H. H. and Nolfi, S. (1995). Evolving Mobile Robots in Simulated and Real Environments. *Artificial Life*, 2, pp. 417-434

Horavec, H. P. (1996). *Robot Spatial Perception by Stereoscopic Vision and 3D Evidence Grids.* CMU Robotics Institute Technical Report CMU-RI-TR-96-34, September 1996. also Daimler Benz Research, Berlin, Technical Report, 1996

Moravec, H.P. (2000). *DARPA MARS program research progress*, http://www.frc.ri.cmu.edu/~hpm/project.archive/robot.papers/2000/ARPA.MARS.reports.00/Report.0001.html, Januray 2000.

Naito, T., Odagiri, R., Matsunaga, Y., Tanifuji, M. and Murase, K. (1997). Genetic Evolution of a Logic Circuit Which Controls an Autonomous Mobile Robot. *Evolvable Systems: From Biology to Hardware*.

Nelissen, M. (2000). *The liblayout library.* http://www.xs4all.nl/~marcone/be.html

Nishihara, H. (1984). Practical Real-Time Imaging Stereo Matcher. *Optical Engineering* 23(5):536-545, Sept./Oct. 1984.

Nolfi, S. and Floreano, D. (2000). *Evolutionary Robotics.* Cambridge, Mass.: MIT Press / Bradford Books.

Nordin, P., Banzhaf, W. and Brameier, M. (1998). Evolution of a World Model for a Miniature Robot using Genetic Programming. *Robotics and Autonomous Systems*, 25, pp. 105-116.

Nourbakhsh, I. (1996). A Sighted Robot: Can we ever build a robot that really doesn't hit (or fall into) obstacles? *The Robotics Practitioner,* Spring 1996, pp. 11-14.

Nourbakhsh, I. (2000). Property Mapping: A Simple Technique for Mobile Robot Programming. Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000).

Park, S.K. and Miller, K.W. (1988). Random Number Generators: Good Ones Are Hard To Find. *Communications of the ACM.* 31: pp. 1192-1201.

Pollack, I. & Pickett, J. M. (1964). Intelligibility of excerpts from fluent speech: Auditory vs. structural context. *Journal of Verbal Learning and Verbal Behavior, 3*, pp. 79-84.

Pomerleau, D. (1989). ALVINN: An Autonomous Land Vehicle In a Neural Network. *Advances in Neural Information Processing Systems 1*. Morgan Kaufmann.

Reynolds, C. W. (1994). Evolution of Obstacle Avoidance Behavior: Using Nosie to Promote Robust Solutions. In *Advances in Genetic Programming*, MIT Press, pp. 221-242.

Rice, J.A. (1988). *Mathematical Statistics and Data Analysis.* Belmont, CA: Wadsworth, Inc.

Rumelhart, D. E.; McClelland, J. L. & The PDP Research Group (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* Volume 1: Foundations. Cambridge, Mass.: The MIT Press.

Selfridge, O. G. (1955). Pattern recognition in modern computers. *Proceedings of the Western Joint Computer Conference.*

Sengers, P. (1998). *Anti-Boxology: Agent Design in Cultural Context.* Ph.D. Thesis, Carnegie Mellon University Department of Computer Science and Program in Literary and Cultural Theory.

Sharoff, Serge (1995). Philosophy and Cognitive Science. From *Stanford Humanities Review,* vol. 4, issue 2: *Constructions of the Mind.*

Smith, Brian Cantwell (1996). *On the Origin of Objects.* Cambridge, Mass.: The MIT Press

Smith, T. M. C. (1998). Blurred Vision: Simulation-Reality Transfer of a Visually Guided Robot. In *Evolutionary Robotics, Proceedings of the First Euopean Workshop, EvoRobot98,* Paris, France, April.

Teller, A. and Veloso, M. (1997). PADO: A new learning architecture for object recognition. In *Symbolic Visual Learning*, Oxford Press, pp. 77-112.

Thompson, D'Arcy. (1917). *On Growth And Form.*

Turing, A. M. (1950). Computing Machinery and Intelligence. *Mind*, LIX, 2236, pages 433-460.

Uttal, William R. (1988). *On Seeing Forms.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Varela, Francisco J.; Thompson, Evan & Rosch, Eleanor (1991). *The Embodied Mind.* Cambridge, Mass.: The MIT Press.

Warren, R. M., & Warren, R. P. (1970). Auditory illusions and confusions. *Scientific American* 223, pp. 30-36.

Winograd, Terry & Flores, Fernando (1986). *Understanding Computers and Cognition: A New Foundation for Design.* Reading, Mass.: Addison-Wesley.

Yamauchi, B. and Beer, R. (1994). Integrating Reactive, Sequential, and Learning Behavior Using Dynamical Neural Networks. *Proceedings of the Third International Conference on Simulation of Adaptive Behavior: From Animals to Animats 3.* The MIT Press/ Bradford Book.

Yam, P. (1998). Roaches at the Wheel. *Scientific American* 278 (1) January 1998, p. 45

Yu, Tina (2000). *An Analysis of the Impact of Functional Programming Techniques on Genetic Programming.* Ph.D. Thesis, http://www.addr.com/~tinayu/