1997

Zooming in on Artificial Intelligence

Zachary Eyler-Walker

Zooming in on Artificial Intelligence: Investigations in Genetic Programming

Zachary T. Eyler-Walker

College Scholars Project
University of Tennessee, Knoxville
zwalker@cs.utk.edu

December 8, 1997

1 Introduction

This project is the culmination of my studies at the University of Tennessee as a College Scholar in Cognitive Studies and Artificial Intelligence. The specifics of the work here are most directly relevant to the latter half of the concentration title, but the motivations are drawn from my program as a whole. Toward this end I plan to demonstrate proficiency in the area of artificial intelligence, in both some of the philosophy and in implementation and work with the model of genetic programming.

This paper is reflective of my time as a College Scholar through its form as well as its subject matter, in that it holds both a wide perspective and a more tightly focused one. To help make this sensible I've tried to clearly delineate the turning points in scope. The paper begins by cutting a wide swath through the history of artificial intelligence as I justify my interest in nontraditional methods (e.g., neural networks, genetic algorithms); there are a large number of possible techniques available to those interested in the field, but like others, I believe there are good reasons for examining these in particular. I will then describe genetic programming specifically in terms of regular genetic algorithms. Once I have established a working foundation of the state of the field I will briefly discuss some details of my implementation of a genetic programming module (the source code forms Appendix B) before moving on to a discussion of some specific applications of the genetic programming package to various problems.

1.1 A Brief History of Artificial Intelligence (and Shortfalls of the Turing Test).

The idea of intelligent machines fascinates many, and people have made various attempts to realize them since long before the advent of digital computers. Artificial intelligence as we currently understand it came into existence with Alan Turing's famous 1950 paper, "Computing Machinery and Intelligence," and the Turing Test. Here he defines AI operationally in terms of the "imitation game," essentially a blind test where a questioner (judge) attempts to guess which of two "speakers"[1] is a real human and which a computer. Implicit in this definition are a number of points; the most important of these is an acceptance that it is not possible to define intelligence in others in terms of some sort of internal experience, since clearly that is closed to us as outsiders. We are grappling with the "other minds" problem here, and while we can by our similar make-ups feel comfortable with the prospect of other human intelligences, it is much more difficult to do so with other entities. Computers are about as physically and structurally different from us as things can be, leaving us, it seems, little but external behavior to make our judgment on. Ultimately the Turing Test wound up tying us to a new set of problems even as it freed us from others that were pressing at the time.

1. Actually, in the Turing test, all communication is held through written text only; all the judge sees is a screen with the responses of the entities being questioned.

The idea of external behavior as a significator of internal awareness was subsumed into the common understanding of the problems of AI by researchers in the field. On its surface, this seems reasonable or maybe even necessary -- what other way can we possibly have of attributing consciousness to another thing -- and in fact, I believe the latter is at least partially true. Nevertheless, through a seemingly innocuous misapplication the Turing Test is also linked to one of the more detrimental paradigms the field of artificial intelligence has known.

There are some obvious philosophical objections one might raise to the Turing Test as a true test of intelligence. For example, it assumes that the intelligence of the agent is comparable to human intelligence, when it maybe be possible to be intelligent in nonhumanlike ways. A more concrete objection is Searle's famous Chinese Room argument (Searle, 1980), which attempts to show that the rules of digital computers can only ever simulate intellectual processes and behaviors, and that mere simulation is not enough to bring about consciousness. Would a simulation of a tornado tear down a real house? Questions like these have divided AI into two different definitions, Strong AI and Weak or Soft AI. The former holds that it is possible to make genuinely aware artificial agents, while the latter is concerned only with the prospect of making programs that can behave passably intelligently.

While this sort of philosophical debate continues to rage,[2] however, there are at least as many difficulties on the practical side of the matter. Most researchers would be happy to create a genuine example of either strong or weak AI -- if not machines who think, at least machines that act as though they do. Before coming back to the Turing Test and my conjecture that it is possible to view the failures of AI as stemming from it, let me give a brief history of the actual work that has been done.

Artificial intelligence as a field seems to have come into its own with the ever more accessible computing power available around the 1960s. By the end of that decade we had several seemingly encouraging examples of solid steps in the right direction. These include programs like Weizenbaum's ELIZA, a simulated Rogerian psychiatrist, and some of the first powerful game playing programs, like Arthur Samuel's expert-level checkers software.

At this point, there was something of a reality check; Weizenbaum himself characterized ELIZA as a cariacture of a real psychiatrist, and of course it would never hold up under any real scrutiny. In effect, it was a domain-specific trick. The game players were also seen to be a dead end -- there is no way they could ever solve any problems other than the playing the games they were designed for. Again, they were so inflexible as to be little more than toys.

---

2. See, for example, Shadows of the Mind (Penrose, 1994), where Penrose presents an argument against any artificial intelligence, strong or weak, from Godel's Theorem. Far from being a definitive refutation of AI, this has sparked numerous objections from all over the cognitive, mathematical, and computer sciences. A number of these, along with a reply from Penrose, are available at http://psyche.cs.monash.edu.au/psyche-index-v2.html#som

To answer these problems, Minsky and others began working with the idea of "micro-worlds." Computers of that time, just as those of today, could never hope to represent all the information needed to solve the problems of proficiency in the huge real world, but at the same time they needed to work in domains with some generality. Micro-worlds were an attempt to bridge this gap, by being pared-down but still rich sub-sets of the real world. The best known example of a micro-world and its associated program is Winograd's blocks world and SHRDLU. Here the micro-world consists of different colored blocks of varying shapes and sizes, say, small and large blue cubes, red cubes, green pyramids, and so on. SHRDLU is able to direct the construction of structures from these blocks given assignments in natural language text, and is also able to answer questions about the world and the blocks in it. Hubert Dreyfus, one of artificial intelligence's most tenacious gadflies, immediately pointed out that this is just another example of a program which can never get out of its first box (Dreyfus, 1979). There is no obvious way for such a program to bootstrap itself out of this limited domain into larger areas, and no reason to suspect that different micro-worlds should have in common any features at all. While Dreyfus' views were hotly contested by Minsky and others when first aired, gradually people came to accept that Dreyfus had identified a real problem.

All of these efforts in artificial intelligence could be grouped together as traditional, with the heading encompassing expert systems, production systems, typical tree-based natural language processors, discrimination nets, and so on.

One thing they hold in common is an overwhelming degree of problem specificity; that is, they can all solve just one problem, or maybe a variety of problems in one domain.

Now, let's come back to the Turing Test and the problems I believe its bastardization engendered in the field. Think again about the test itself and what it requires in the abstract. Essentially, to pass the Turing Test a computer program must be able to convince a human that the program is performing up to snuff (human snuff, that is). This test became the de facto standard as the trial which must be passed for a program to be considered intelligent. Still, writing a program to actually do this was obviously too difficult, and so researchers made the assumption that the Turing Test could be partitioned into subtests, each in a different domain. Thus, we have a Turing Test for chess, and one for dwellers in blocks worlds, and one for corny Rogerian psychiatrists.

At any rate, after these formative years and the failures that came with them there was a slow turning over to alternative and more general ways of solving problems, reaching full steam in the mid-80s with the publication of the <u>Parallel Distributed Processing</u> books (Rumelhart and McClelland, 1986). Artificial neural networks are an abstraction of biological neural systems, where an idealized neuron is the building block for a layered system of connections. In fact, neural networks, most notably the perceptron, had been around for some time; one of the earliest proponents of artificial intelligence, Warren McCulloch, along with

Walter Pitts, had actually developed a simple model of a biological neuron in 1943 and shown how collections of these could be used to perform any computation. However, there had existed no effective way to train neural networks of more than a single layer in depth, and they had fallen into disuse after the publication of Minsky and Papert's Perceptrons in 1969. This volume included a devastating attack on the model named in the title and the delta learning rule it made use of. Particularly, the authors were able to show that perceptrons could not learn even the simple relationship of the Boolean exclusive OR. The PDP books described artificial neural networks and their application in a compelling fashion, and also included a description of a the back-propagation or generalized delta rule for hidden weight tuning. The new learning rule allowed for fully trainable multilevel networks, and this both refuted the objection by Minsky and Papert and allowed the method to rise again from the depths of obscurity to become well known and often used, both in research and in industry.

The other well known non-traditional set of techniques in artificial intelligence is evolutionary algorithms, best known through genetic algorithms. As with neural networks, some work with them was done concurrent to the very beginnings of artificial intelligence, though not in that field. Like the name suggests, evolutionary algorithms are based on the theory of evolution and natural (unnatural?) selection; in effect, a random population of solutions is grown and then maintained and bettered through some kind of combination of the candidate solutions to form new members of the population.

In addition to being grouped together by exclusion from the traditional methods, these particular non-traditional techniques also have in common at least abstract biological foundations. This is certainly intellectually appealing, as it automatically and necessarily brings them closer to the phenomenon they are used to model. While not a direct congruency, it is far closer than the kind of rules-based approaches of which traditional AI makes use, which we know are at odds with natural organisms and systems. Another commonality is that these methods can be used to solve nearly any sort of problem, though in practice not necessarily all at once. Again, this makes the methods more appealing than traditional artificial intelligence, as it gives hope that they can be used to produce the kinds of generality clearly needed to make a dent in the problems facing the field.

1.2 Why Genetic Algorithms?

Of the two nontraditional methods mentioned, neural networks are by far the better known and more often used. So, why would I choose genetic algorithms as an area to focus on? Neural networks can do amazing things and can be nearly direct reflections of the systems we'd like to model. However, they are extremely slow to train for even simple problems when using the back propagation learning rule, and their structures are notoriously difficult to specify. Small changes in the numbers of neurons and their connections between layers can make or break the success in a given application. Additionally, because they are so widely used, there is quite a lot of literature about neural networks.

The Darwinian action of genetic algorithms is intrinsically beautiful, and the idea is quite elegant. The basic action of a generic genetic algorithm is as follows:

1. The problem is examined and an encoding is found that can be represented as a string of ones and zeroes, the building blocks of the "chromosomes" used.

2. A function is developed that can grade candidate solutions of the problem as it is encoded in step (1). A solution that completely solves a problem would be given a rating of 100%, while one that only partially solves it might be rated 43%; a total failure would score 0%. This is the mechanism for selecting the fitness of solutions.

3. An initial population of solutions is created, each a fixed length string of randomly placed ones and zeroes.

4. Each solution is tested with the function described in step (2), and they are all put in a rank-order.

5. With probability proportional to their fitnesses, solutions are selected from the population, and are "bred" with the operators of cloned reproduction, crossover, and mutation. See Figure 1 for details on these operations. The new solutions created form a new population of the same size as the initial population.

6. This cycle is repeated from step (4) above with the new population until a solution emerges which is correct enough for the problem.

The applicability of genetic algorithms is limited only by the ability of the user to generate an appropriate description of solutions in terms of ones and zeroes, then. While the solution ultimately arrived at may not be very comprehensible (coming in a binary string as it does), the process by which it is gotten is transparent and intuitively meaningful. Theory behind genetic algorithms has been harder to come by and often is subtle, relying heavily on probability theory to describe the way that a population will converge on optimal solutions. Much of the work has been done by Michael Vose and his collaborators, who have derived formalisms of finite population sized genetic algorithms as a Markov chain (Nix and Vose, 1991).[3] For the purposes of application, it is sufficient, of course, to know that GAs do in fact work.

This cleverness of modeling a computer learning process after natural selection; the satisfying and relatively fault-tolerant (at least as opposed to neural networks) way in which genetic algorithms converge on good solutions; and the fact that there are fewer applications of them to the problems that interest me were all deciding factors in my turn to this area.

1.3 Why Genetic Programming?

In 1992 John Koza published an expanded version of his Ph.D. dissertation entitled <u>Genetic Programming</u>. This described a new extension to the genetic algorithm model in which candidate solutions were represented as trees composed of functions and variables rather than fixed length strings. See figures 1 and 2 in the first appendix for descriptions of genetic crossover and mutation with function trees. Rather than breeding bit strings which need to be decoded,

3. A Markov chain is a stochastic process where the probability that the system will be in any specific state at time depends only on the state of the system at the time immediately before.

we are now breeding possible direct solutions to the problems at hand. There are some additional constraints, the most important of which is that all elements that make up a program tree -- the functions as internal nodes and the terminals as leaf nodes -- must have consistent return values and must not fail to work on any input possible. For example, a function for performing division should probably not be but in the position of having to account for an input like "The quick brown fox," even though a program designed to do text processing of some sort might accept that handily. Steps must be taken to ensure that only numbers reach arithmetical operators, or that if something else does there is some nondestructive way of dealing with it. Additionally, because it is not possible to divide a number by zero, a division function must catch any instance where the divisor is zero and deal with the problem in some arbitrary manner.

This is the primary advantage of genetic programming over simple genetic algorithms. Defining the problem becomes much more simple because it can be defined in terms of the real solution required. If one is trying to model a mathematical function, all one must do is provide a few basic arithmetical functions and some constants, and press go. If one wishes to develop a means of controlling some robot with certain possible behaviors (e.g., move forward, turn left, turn right, look, grasp), then those behaviors become the functions and terminals, and to test the problem one has only to execute the candidate programs. Defining problems in these terms is obviously much more natural than attempting to model problems in terms of finite state machines or with complex grammars derived from zeroes and ones, as is generally required by a standard genetic algorithm.

2. The Genetic Programming Package

Part of the project was simply creating a working example of a module that could be used to implement a genetic programming solution to various problems. This process certainly turned out to be the most difficult, and very time consuming for a couple of reasons. First, getting what I considered to be a reasonable amount of speed out of the package turned out to be fairly difficult. Initially I designed the system using tree-based data structures to specify the logical program trees. This turned out to be extremely slow because of the constant memory access; one day, frustrated by this implementation, I seized on the plan to scrap it and write a new one, this time using a much faster array-based data structure. While this method is much quicker -- nearly an order of magnitude, typically -- it was much more complicated to write and debug. Though only around 1500 lines of code, it took a very long time to get working correctly. In retrospect, the time taken rewriting the program might have been better spent implementing a better memory allocation/garbage collection routine for the original tree-based program. Nevertheless, rewriting the program did allow for some streamlining and addition of functionality, so I am ultimately not dissatisfied.

Any user of the package must know how to program in C to at least a moderate extent. While the module does vastly lighten the burden of finding a genetic programming solution to a problem, it cannot do everything.

3. Application of the Genetic Programming Package

In this paper I am including three sample problems solved by the method of genetic programming, using the module I developed. Only one of these applications is in a new domain, but each had interesting results. The problems were also chosen to demonstrate the large range of practical uses of genetic programming. The three are the Boolean 11 multiplexer, an autonomous lawnmower, and SKI-combinator trees.

3.1 The Boolean 11 Multiplexer[4]

This is a standard device used in the binary logic of digital circuits. There are three control lines, A0-A2, and eight data lines, D0-D7. Depending on which of the control lines are asserted, one of eight different values is generated; on zero, only the output of D0 is passed out, on one, only the value of D1, and so on, up to a control value of seven producing only output D7. The programs evolved will attempt to output the value of the correct data line. Koza points out that this is a Boolean function of $k + 2^k$ arguments, making it a particular one of 2^{2048} pos-

4. This problem is directly adapted from the 11 multiplexer in Koza (1992, pp 170-189)

sible Boolean functions of 11 arguments. This corresponds to a search space of about 10^{616} points; considering that the roughly 10^{120} possible games of chess have been shown to require a computation that would be halted by the heat death of the universe, it is clear that this search space is intractable by blind methods.[5]

In order to solve the problem, we need to pick terminals and the functions to operate on them. Here selection of terminals is simple; we designate the set of control and data lines as the terminals, and nothing else. The functions to be used are a little less clear, but following Koza for now we will use if-then-else, and the Boolean functions AND, OR, and NOT. These functions have three, two, two, and one arguments respectively.

T = {A0, A1, A2, D0, D1, D2, D3, D4, D5, D6, D7}
F = {AND, OR, NOT, IFTE}

Because there are 2^{11} possible combinations of inputs, we will test each member of the population that many, 2048, times. The fitness will be calculated as the sum of the correct outputs -- when a fitness of 2048 is reached, we will have created a 100% successful program. Like Koza, I chose a population size of 4000, and set all other major parameters to his specifications as well. Unlike Koza, I was completely unable to pick an initial value for the random number

5. Koza (1992, p. 172)

generator I used which would give me a population that would converge to a solution in nine generations. Because we are evaluating 2048 * 4000 individuals per generation, each of which averages around 200 instructions (something like 1.5 billion high level functions from the set {F + T}, per generation), this can mean that even with the fast GP I've implemented, quite a bit of time is required to find a solution.

A typical run requires between 25-35 generations to solution, although there has been one that took only 15. One of the parameters Koza specifies is the depth of the program trees in the initial population; 6 is used here. In my experience, an initial depth parameter of 3 tended to produce populations that converged significantly more quickly, typically between 10-20 generations.

Koza uses this particular problem as an illustrative example of some of the self-organizing properties of genetic programming. Specifically he suggests that the (attractive) structure of the solution arose due to the "relentless pressure exerted by the fitness measure."[6] Again, I would like to suggest that he was simply very lucky, and that while there is a relentless pressure towards the production of solutions, there is little in here that could possibly pressure nice-looking solutions.[7] For every solution like his:

6. Koza (1992, p. 179)
7. It is possible to include some notion of what the function ought to look like in the evaluation function, including, for example, a term that decreases as total program size increases, pressuring smaller programs. This tends to decrease convergence towards a problem solution, however, and must be carefully handled.

```
(IFTE A0 (IFTE A2 (IFTE A1 D7 (IFTE A0 D5 D0))
                      (IFTE A0 (IFTE A1 (IFTE A2 D7 D3) D1) D0))
          (IFTE A2 (IFTE A1 D6 D4)
                      (IFTE A2 D4 (IFTE A1 D2 (IFTE A2 D7 D0)))))
```

there are probably thousands like the following solution I reached:

```
(| | (&& (&& (IFTE (IFTE A1 (&& A1 A0) (! (&& A2 A0))) (! (&& A2 A0))
(&& (| | (| | D0 A1) (IFTE D0 D7 A0)) (| | (IFTE (| | (&& A1 A0) (IFTE
(! (IFTE D0 A0 A0)) (IFTE A2 D3 D2) (&& (! (&& A2 A0)) (! (| | A1
D1))))) (IFTE A2 D3 D6) (| | (IFTE (&& A0 D7) A0 A0) (&& D3 D2)))
(IFTE A2 D3 (IFTE (! (| | (| | D0 A1) (IFTE D7 A1 D4))) (| | (IFTE A1 D2
D4) (&& D1 A1)) (| | (| | D0 A1) (IFTE D7 A1 D4))))))) (IFTE (! D5) (&&
(IFTE (! A1) (IFTE (! (&& (| | (IFTE A2 D1 (IFTE A0 D6 D0)) D5) (&& D0
A0))) (IFTE A2 D1 D0) (IFTE D7 A1 D4)) (! (! (| | (| | D0 A1) (IFTE D7
A1 D4))))) (IFTE (! (&& A2 A0)) (IFTE A1 A1 D1) D5)) (| | D5 D0)))
(IFTE (IFTE A1 (&& (&& A0 D7) (| | D4 D7)) (| | (&& A1 A0) (! D5)))
(IFTE A2 D1 (IFTE A0 D6 D0)) (IFTE (IFTE A1 (&& A2 A0) (! (&& A2
A0))) (IFTE A2 D1 (IFTE A0 (IFTE A2 D1 (IFTE A1 D2 D4)) D0)) (&& (| |
(| | D0 A1) (IFTE D0 D7 A0)) (| | (IFTE (| | (&& A1 A0) (! D5)) (&& A2
A0) (| | (IFTE D0 A0 A0) (IFTE (! A1) (! D7) (! A1)))) (IFTE A2 D3
(IFTE (! (! A0)) (IFTE A0 D6 D0) (IFTE (! (! A0)) (IFTE A0 D6 D0) (| |
(! A1) (IFTE A2 D3 D2)))))))))) (&& (IFTE A0 (IFTE A1 D7 D4) D0) (IFTE
A1 (&& A2 A0) (! (&& A2 (IFTE A0 (IFTE A1 D7 D4) D0))))))
```

To say the least, this is not a very illuminating piece of code. There are many subexpressions above that could be simplified since they include redundant functions, but even then the result would be quite unwieldy.

A final point of interest regarding the 11 multiplexer can be found after examining the solution Koza reached; namely, the only function his solution makes use of is the IFTE operator. What if we reduced the total function set to include only that operator? This will obviously still give a sufficient degree of functionality to solve the problem, and one of the selling points of this method is that a sufficient set ought to be good enough to solve the problem.

When I reduced the function set accordingly, I got a surprising result, which was that none of the runs ever converged within 50 generations. Apparently it is not always enough just to include the bare minimum of a search space to the genetic programming module.

3.2 The Lawnmower Problem

This is a problem of my own devising, although it does draw from ideas in Koza's book regarding an artificial ant picking up food along a trail.

The function set is:

F = {IF GRASS AHEAD, IF MOWN LAWN AHEAD, PROG2, PROG3}

Both of the IF expressions take two arguments, the first of which is executed if the condition (grass ahead, mown lawn ahead) is met, and the second if it is not. PROG2 and PROG3 take two and three arguments respectively, and simply execute them one after the other unconditionally.

The terminal set is:

T = {TURN LEFT, TURN RIGHT, MOVE FORWARD}

These each do as you would expect. Additionally, each of these terminals increments a counter corresponding to the total number of (abstract) time steps the lawn mower uses. When the counter reaches a certain number, computation for that member of the population ends. Another condition of the mower's existence is that, of course, any square it passes over that contains grass becomes mown.

The evaluation function for this problem takes into account two things. First, it sums the successfully mown squares. Then there is a penalty to the mower for driving over non-grassy or mown surfaces. These might correspond to gravel, which we would like to avoid, of course. So, this penalty, the number of times a non-lawn square is driven over, is subtracted from the mown total. This number is then compared to the total number of grassy squares to give the total fitness. To achieve 100% success, then, no nongrassy squares may be covered.

Initially I tried to evolve programs with just the information and functionality above. The population size was set to 8000, and the initial depth of generated programs to 6. The mower performed fairly well, though not flawlessly, on simple problems like a square shaped virtual lawn. It failed miserably on more complex patterns such as spirals and concave polygons (those with internal angles of greater than 180 degrees). To remedy this I inserted the program evaluation into a hardcoded looping structure, which looped until the mower had used up all its time steps.

After this modification the mower performed amazingly well. It was able to eventually learn even quite difficult shapes, such as the joined spiral shown in ascii graphics on the following page.

The interesting thing about the mower's performance on such difficult shapes as this is that typically it would immediately hone in on a fairly good solution, in this case successfully traversing the basic spiral only, as seen in the first graph of figure 3. After this first success there would be a long refractory period of no change, sometimes lasting as long as 10 or 15 generations. Looking at the pattern it is easy to guess why (the $ at the upper left corner is the starting point of the mower. It is facing south):

```
$###############################################
                     #                         #
###########################################    #
#                            #            #    #
#    #################################    #    #
#    #       #                       #    #    #
######   #########################   #    #    #
#    #   #           #           #   #    #    #
#    #   #  ###############      #   #    #    #
#    #   ####    #        #      #   #    #    #
#    #   #  #   #######   #      #   #    #    #
#    #   #  #####  ####   #      #   #    #    #
#    #   #  #   #  ####   #      #   #    #    #
#    #####  #   #         #      #   #    #    #
#    #   #  #   ###########      #   #    #    #
#    #   #  #      #             #   #    #    #
#    #   #  ######################   #    #    #
#    #   #              #            #    #    #
#    #   #############################    #    #
#    #              ##                    #    #
#    #              ##                    #    #
#    ######################################    #
#                                              #
################################################
```

The spiral is very easy to complete with the new hardwired iterating structure. A simple (IFGRASSAHEAD FORWARD RIGHT) will do nicely, for example. Any modification mowing new squares will have the side effect of bypassing large sections of the lawn maze, and in the absence of sophisticated case rules the mower will more than likely fail to return to mow those sections later. Eventually, though, a slightly more effective searcher will tend to be produced, which quickly opens the flood gates for more and more effective solutions using the new operator. Once a single square not in the spiral was mown in addition to the spiral, it typically took no more than four to five more generations to attain 100% success. The rather devious path taken by one 100% successful solution is seen at the bottom of figure 3.

I was curious to see if modifying the function set to include some iterating structures would make it possible to evolve successful mowing programs without the hardwired iterator in the evaluation code. To this end I added two looping functions, a WHILE GRASS AHEAD MOVE FORWARD and a corresponding function testing for mown squares. The loop in the evaluation code was removed.

I didn't expect to get quite the success of the hardwired iterator, but was surprised to find that these new structures almost totally failed to increase the fitness of the population, to the point that the most fit programs typically didn't even include the new operators, or only included them in spots which could not logically ever be reached. This was true on all problems, including a bare square lawn, where it would be easy to handcode a solution using these iterators to achieve at least a reasonable level of fitness.

The lesson from this is again that genetic programming can be somewhat unpredictable, even with very plausible operation sets. However, it can also solve quite difficult problems; it would be hard to imagine writing a successful navigator of the maze-like lawn by hand given only the information the mower has (i.e., it can only sense the contents of the square directly in front of it).

3.3 SKI-Combinator Trees

In the early fall of this year, Bruce MacLennan released a technical report investigating some of the properties of a specific combinatory logic system, one based on the SKI calculus.[8] His aims were specifically to examine the possibility of using SKI trees to model chemical processes; in this preliminary paper he concentrated on an explanation of the SKI operators and set about trying to define limits on the behavior of SKI strings, or trees which branch only to the left, based on their statistical make-up.

The S, K, and I combinators are fairly simply defined as follows.

The I combinator is the identity function, defined by the rewrite rule Ix => x. This is simply to say that I replaces itself by its argument.

The K operator is a bit more complicated, written as Kxy => x. Like the I combi-

8. MacLennan (1997)

nator, K removes itself from the tree. This time, however, it also removes its second argument, which may be another tree in itself. K can have the effect of drastically reducing the total size of the tree, therefore.

Finally, the S combinator is the most complicated member of the group. It is written as follows: Sxyz => ((xz)(yz)). While tricky to explain precisely, this operator like the others deletes itself from the tree upon application. It is productive, however, and in addition to rearranging its immediate locale also completely duplicates one of its arguments, the subtree z. So, just as K deletes a subtree, S creates a duplication of one. The combinators are complementary in a way.[9]

A final thing to consider is the order in which these combinators are applied to the tree they reside in. Typically, they are applied at each node in reverse order (that is, I, then failing that, K, then failing that, S). Additionally, MacLennan only considered the SKI reduction of trees in the normal order. Normal order reduction consists of always applying the operators at the point nearest possible the root of the tree, that is, closest to the top of the tree. With this constraint, it can be sure that if a tree grows infinitely, then it would also grow infinitely under all other orders of reductions; likewise, if any order can complete, one can be sure that a normal order reduction will.

9. For a more detailed explanation, as well as some graphical aids, see MacLennan (1997, pp 2-3)

In hopes of laying out some groundwork that will be able to further this research, I implemented an SKI problem with my genetic programming package. There are two advantages here over the methods offered in the technical report. One, the GP package automatically creates fully branching SKI trees, as opposed to the left branching strings so far studied. Two, it is possible to breed the strings to various arbitrary measures of fitness.

The terminal set is:

T = {S, K, I}

The function set is:

F = {APPLICATION}

The APPLICATION operator is simply the glue forming the structure of a given SKI tree; it takes two arguments, each of which is either another APPLICATION operator or one of the SKI terminals, forming either another branching point or a leaf.

Because it was conjectured that the most interesting SKI reduction behaviors are likely to exist in trees that neither die immediately out nor immediately shoot to infinite size, I decided to breed SKI trees for the slope of their expansion in size versus number of applications of the reduction operation. The evaluation function did not exactly check this slope since it was likely to fluctuate, but only the end points, with the average of the size being used an abstraction of the middle height. I was then able to select for a specific pseudo-slope. Due to memory limitations and time considerations, I stopped reducing the SKI trees after 25 iterations.

This is a strange enough problem that it seems likely it would be extremely difficult for a human to come up with a good way of building trees that meet the above criteria -- I certainly don't see how to do it. Genetic programming had no trouble at all. At this point I've bred trees corresponding to several different slopes (1, 5, 10, 20, 40 per application of tree reduction), and followed the relative proportion of S, K and I operators. There is not yet enough data gathered to provide a statistically valid answer to the question of whether there are interesting, but the initial results seem to suggest that for this problem, at least, the structure of the SKI trees is more important than the relative proportions of the combinators in them. The only thing that held constant was that there were always approximately 10-20% more S operators than Ks and Is. It should again be stressed that all the results are not yet in.

3.4 Discussion of Genetic Programming

So, these examples and results obtained through them should show some of the wide variety of uses genetic programming can fairly easily be put to. Like other techniques, the method is not without its pitfalls and shortcomings. There are even more parameters to tweak than in a typical neural network, for example; though, to be fair, genetic algorithms and programming seem more tolerant of the settings of these parameters. It is not always clear how to apply GP to a given problem, despite it being much more simple than the corresponding process with neural networks and normal genetic algorithms, let alone traditional rules based systems. In the end I think more effort needs to be devoted to increasing the efficiency and productivity of the model before it will truly come into its own. Drawing from real biological systems may be the best way to do this; certainly nature has done all right with its own version.

4. Conclusions

I hope I've managed to demonstrate both a knowledge of the field of artificial intelligence and its history and philosophy, as well as some of the finer points of one method it has available to it. At this point, artificial intelligence seems to have been around for a very long time, though the modern field is no more than 50 or so years old. Old, but surprisingly young. I am left feeling that this quote from Wittgenstein is particularly relevant: A child has much to learn before it can even pretend.[10] This applies not only to the systems we endeavor to make, but to ourselves, as well. With the new nontraditional models, I begin to feel that we may soon at least be pretending convincingly.

5. Bibliography

10. Unfortunately, the source of this quote is forever lost to me.

1. Dreyfus, H. (1979) "From Micro-Worlds to Knowledge Representation: AI at an Impass." From What Computers Can't Do, Harper and Row, New York.

2. Koza, J. R. (1992) Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press, Cambridge, MA

3. MacLennan, B. J. (1997) "Preliminary Investigation of Random SKI-Combinator Trees" Technical Report, University of Tennessee.

4. Minsky, M., and Papert, S. (1969). Perceptrons. The MIT Press, Cambridge, MA

5. Mitchell, M. (1996) An Introduction to Genetic Algorithms. The MIT Press, Cambridge, MA

6. Nix, A. E., and Vose, M. D. (1991) "Modeling genetic algorithms with Markov chains." Annals of Mathematics and Artificial Intelligence 5: 79-88.

7. Penrose, R. (1994) Shadows of the Mind. Oxford university Press, New York.

8. Rumelhart, D. E. and McClelland, J. L. (1986) Parallel Distributed Processing: Explorations in the Microstructure of Cognition. The MIT Press, Cambridge, MA

9. Searle, J. R. (1980) "Minds, Brains, and Programs." In Mind Design: Philosophy, Psychology, and Artificial Intelligence, ed. J. Haugeland. The MIT Press, Cambridge, MA

10. Turing, A. (1950) "Computing Machinery and Intelligence." In Computers and Thought, E. A. Feigenbaum and J. Feldman, eds. McGraw-Hill, New York, c. 1963

Appendix A:
Graphs and Figures

Mutation in Genetic Programming

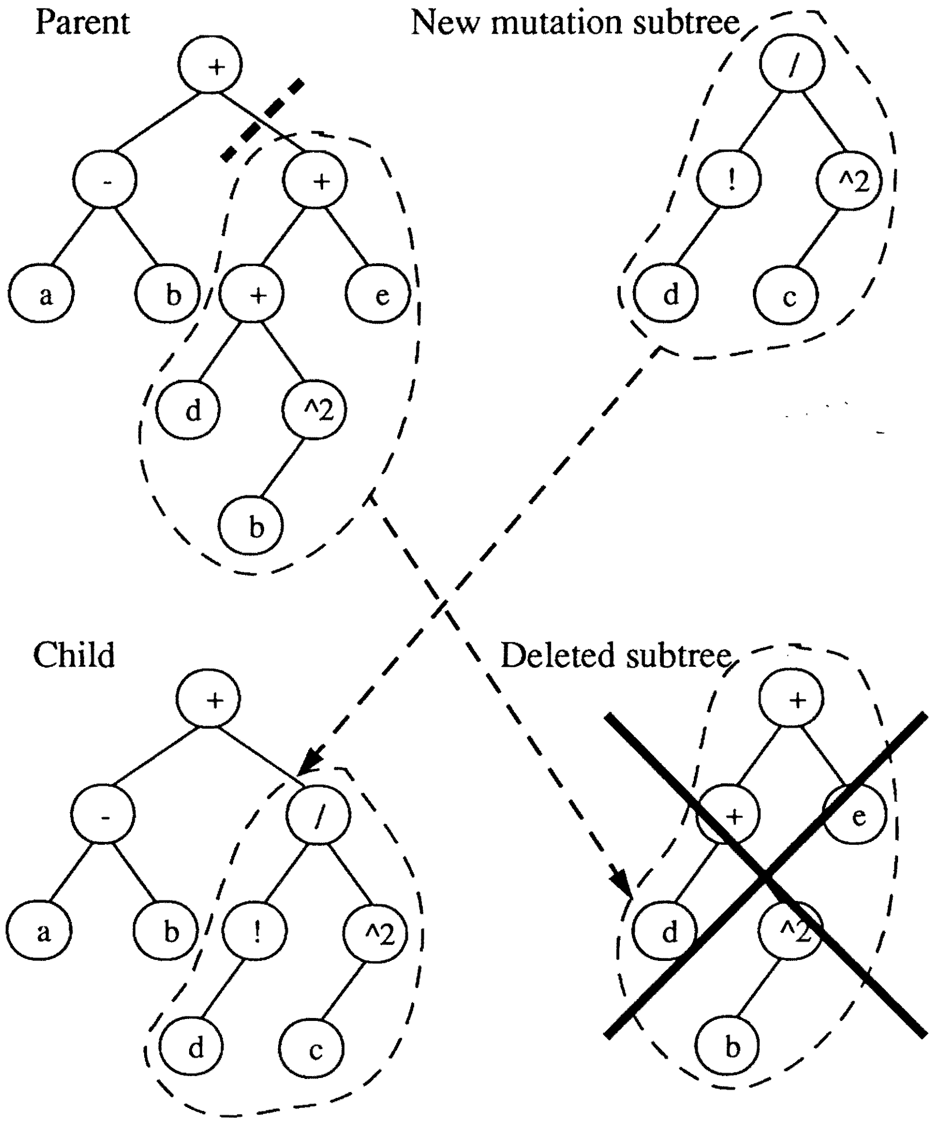

Figure 2.

Mutation in genetic programming is similar to mutation in standard genetic algorithms, with a few exceptions. A single parent is selected from the population, and then a link in the parent's program tree is chosen. At this point an entirely new random tree is generated; this is typically kept a moderate size to avoid too large a growth in program evaluation time. The subtree below the selected link in the parent tree is then deleted and immediately replaced with the newly generated subtree, to form a single new child program.

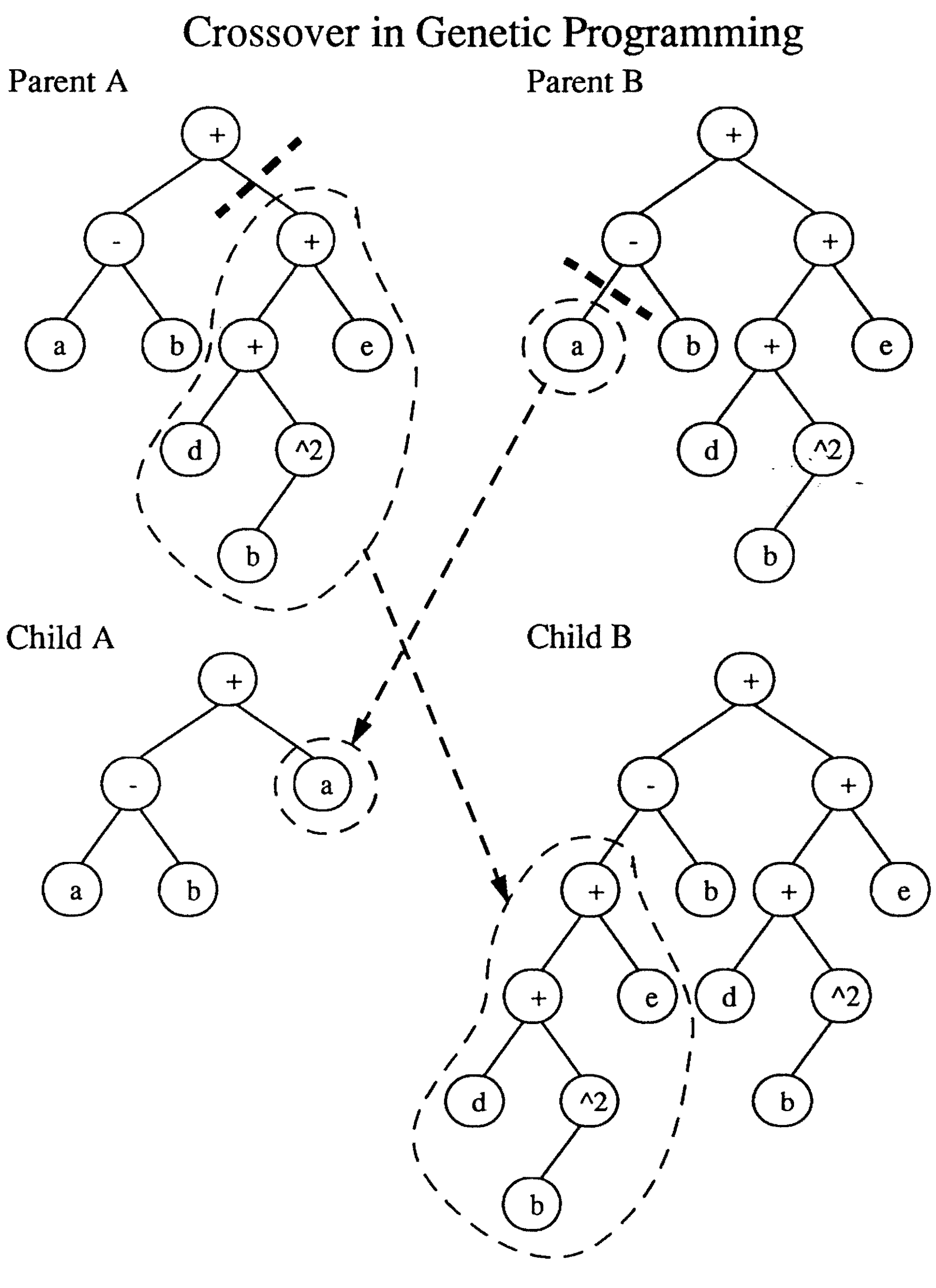

Figure 1.

Crossover in genetic programming is a relatively simple process. Two parents are selected (as here, they may both be the same), and a link in the program tree of each parent is chosen. To create the children the subtrees below those links are swapped, typically resulting in two new programs. Note that this process can both grow and shrink program trees.