Astro Teller

Most interesting problems do not have solutions that are simple mappings from the inputs to the correct outputs; some kind of internal state or memory is needed to operate well or optimally in these domains. Traditionally, genetic programming has concentrated on solving problems in the functional/reactive arena. This may be due in part to the absence of a natural way to incorporate memory into the paradigm. This chapter proposes a simple, Turing-complete addition to the genetic programming paradigm that seamlessly incorporates the evolution of the effective gathering, storage, and retrieval of arbitrarily complicated state information. A new environment is presented and used to evaluate this addition to the paradigm. Experimental results show that the effective production and use of complex memory structures can be evolved and that functions evolving the intelligent use of state quickly and permanently displace purely reactive and non-deterministic functions. These results may aid future research into the causes and constituents of mental models and are shown to open the field of genetic programming to include all learning strategies that are Turing-possible.

9.1 Introduction

In speaking about machine learning, we usually mean something more specific than the verb "to learn" in all its uses. Machine learning as a discipline often emphasizes problems of classification and the discovery of functions. These problems include tasks such as symbolic regression, broom balancing, and playing chess. All of these tasks are learning enterprises, but none of them requires the use of state. In general, an agent can neither act nor learn in an environment without some model of the environment. This model must come as a combination of those model features given to the agent and those it builds and maintains on its own. Learning, given an environmental model, is an important problem. Creating a model, however, is also a fundamental part of what it means to learn.

Imagine a small boy (or any small agent for that matter) riding a bicycle around his neighborhood. We might say that this agent has learned to ride the bicycle. This learning process is highly reactive. The ability to lean in different directions in order to remain upright requires no state information. We might also like to say that the agent has learned its way around the neighborhood. This learning is not only the developed ability to navigate but the construction of a mental map of the area. Genetic programming has been applied in the past to problems like the ability to balance on a bicycle but little work in evolutionary strategies has been done on creating and using a mental model of the environment.

The use of state to solve problems is not new to genetic algorithms or genetic programming. In both areas state has been used implicitly and explicitly. In no case, however, has it been a natural, integrated part of the process. In the field of genetic algorithms encodings have been done for both finite state automata and recurrent neural networks. [Jefferson *et al*. 1991] In both cases, binary encoding of lengths over 400 lead to 32 states. In an evolution of solutions to the Prisoner's Dilemma using state, the binary encoding used $2^{\mathbf{m}}$ bits for an individual who could remember back **m** binary moves. [Lindgren 1991] In genetic programming, one of the techniques has been to use *Progn* as a way to string together side-effecting terminals. [Koza 1992] *Progn* is a more implicit use of state. It is not clear whether this use of state in genetic programming would be generally applicable in memory intensive problems and it does not lend itself to any kind of introspection (i.e. decision making based on a function's current state).

The organization and use of memory is often roughly divided into three classes: state information, iconic models, and sentential models. However distinguishing between these three categories is a very subjective business. State information can be thought of as "memory" that divides all possible states of the world into subsets. State information becomes an iconic model when these subsets of the possible world states are divided along boundaries that correspond to " simple features" of the environment. State information can be interpreted as a sentential model when there is a semantic assignment to the state information that shows the memory to contain recursively combinable statements concerning what is true about the environment (e.g. "A implies B"). For a more complete discussion of these issues try *Mental Models: Towards a Cognitive Science of Language, Inference, and Consciousness*. [Johnson-Laird 1983]

These categories of mental models may be useful for speaking about certain mental activities. However, the subjectivity of these divisions suggests that mental processes are a continuum and cannot be divided along any well defined boundaries. Even if these mental categories are interesting ranges along this continuum, there is no reason to believe that these categories, taken from how we think, will be of any relevance to how evolved agents build and use state information. Rather than force the results that follow into some predefined mental pigeon hole, this chapter hopes to help call into question the meaningfulness of categorizing the organization and use of memory.

The purpose of this chapter is threefold. First, it is an attempt to evolve individuals that use state effectively and that even build mental models of an environment given only temporally and spatially immediate sensory input. These mental model builders use an addition to the genetic programming paradigm called **indexed memory**. The second purpose of this chapter is to evaluate indexed memory with respect to the evolution of mental models and as a general, Turing-complete tool for using memory in genetic programming. And finally, this chapter presents the environment in which these agents operate, **Tartarus,** as a benchmark for the effective use of state. This environment condemns the population individuals, like Sisyphus, to spend forever fighting the entropy of their system. At the beginning of each new generation, the world is returned to its disordered state and the individuals must start again. In this spirit the test environment is named Tartarus after the underworld where Sisyphus spends an eternity pushing his boulder out of a global minimum.

Indexed memory was not designed to foster the emergence of mental models *per se*, but to be a general method for the use of memory in the field of genetic programming. The evolution of the intelligent use of state was an appropriate first trial of this paradigm; indexed memory facilitates a variety of uses for state and is simple enough not to overly bias the evolution of state use. This chapter, through results and discussions, will suggest that this new scheme for incorporating the use of memory into genetic programming will be practically applicable to a wide range of problems that have already been investigated and will open the door to new problem areas that are memory-critical. In addition, this attempt to produce agents that build up and effectively use state may be a small step into using computational evolution to study the nature of mental models.

9.2 The Method and The Model

Before indexed memory is described, motivations and criteria for its creation will now be summarized. The ideal state storage model should allow genetically evolved functions some atomic abilities to examine and change their state. The *Progn* model of state usage is an example of a strategy that does not allow the function to examine its state using the same mechanism used to evaluate its inputs. Any new state storage model should be paradigmatically neutral and should operate at a low level. There should be little bias on how evolved functions make use of their state and no data structures or "concepts" should be impossible because of the high level at which the state storage scheme operates. In genetic programming there is a constant tradeoff between the benefits and losses of constraining the search space of algorithms. The "lower" (more basic) the memory scheme we choose, the more of the algorithm space we can/have to explore. As we tackle increasingly difficult problems with genetic programming, our intuition fails us and it becomes increasingly important not to warp the fitness landscape, causing good solutions to become hard or impossible to obtain. For this reason, paradigmatic neutrality and basic functionality were important criteria in settling upon a memory scheme. In addition, and most importantly, any new state storage model should facilitate any computation (i.e. be Turing-complete).

In the general case of indexed memory, each population individual has not only a functional tree, but an array (**Memory**) of elements indexed from 0 to (M-1). Each of these M elements is an integer in the same range. The idealization of indexed memory is a memory array indexed over the integers whose element values are also integers. For practical purposes however, some M will always be chosen to approximate this idealization. Two non-terminals are added to any existing function set: **Read** and **Write.** Read takes one argument (Y) and returns Memory[Y]. Write takes two arguments: X and Y. It returns the old value in Memory[Y] and then puts X into Memory[Y]. Write could return the new value in Memory[Y] instead and indexed memory would work just as well. Both parameters can be any integer between 0 and (M-1) and both functions return an integer between 0 and (M-1). For the Tartarus world M was chosen to be 20. This may seem small but consider that the agents now have 20^{20} states available to them (approximately $1.05 * 10^{26}$). This number is significantly larger than the few dozen states available to genetic algorithm and genetic programming experiments done in the past. The choice of M=20 was largely for simplicity, but indexed memory with dynamically expandable M is Turing-complete as will be discussed later in the chapter. Also later in the chapter the evolutionary effects of significantly larger M will be discussed along with the ramifications of having so many states available.

There will be only two kinds of terminals: constants (between 0 and (M-1)) and inputs. The number and meaning of these inputs will be set when the environment is described. All functions were constrained to return integers between 0 and (M-1) so that any computed value was a legal memory index. This restriction could have been relaxed by taking the index modulo M before accessing memory, but was not for simplicity. The non-terminals selected for this problem were:

F = { OR,NOT,LESS,ADD,IF-THEN-ELSE,SUB,EQ,READ,WRITE,ADF }

(**OR** X Y) --> This returns the MAX(X,Y)
(**NOT** X) --> This returns 1 if X=0, otherwise it returns 0.
(**LESS** X Y) --> This returns 1 if X<Y, otherwise it returns 0
(**ADD** X Y) --> This returns X+Y (ceiling of M-1)
(**IF-THEN-ELSE** X Y Z) --> This returns Y if X>0, otherwise it returns Z
(**SUB** X Y) --> This returns X-Y (floor of 0)
(**EQ** X Y) --> This returns 1 if X=Y, otherwise it returns 0
(**READ** Y) --> This returns Memory[Y]
(**WRITE** X Y) --> This returns Memory[Y] and THEN sets Memory[Y] to X

In addition to the main genetically programmed functional tree, the agent is given an automatically defined function (ADF) tree with two arguments. In that subexpression the terminals and non-terminals already stated are legal. The ADF has two additional

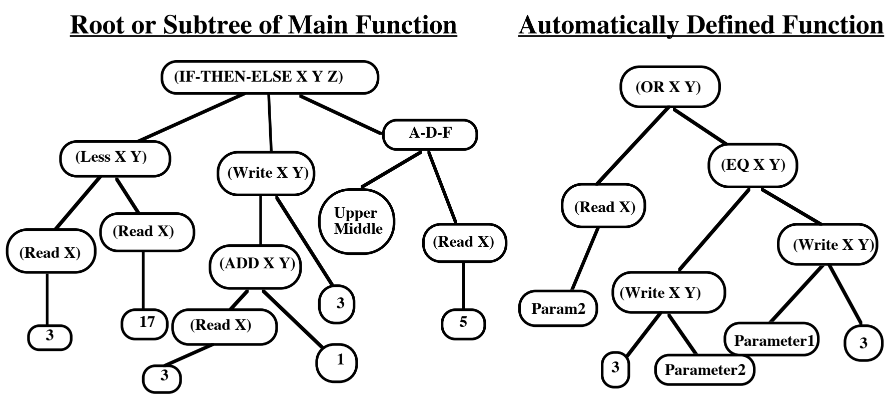

Figure 9.1

terminals: **Parameter1** and **Parameter2**. The main function of the agent has the additional non-terminal (**ADF** X Y) which sets Parameter1 to the evaluation of the subexpression X and Parameter2 to the evaluation of subexpression Y and then evaluates **ADF**. Like the other functions, **ADF** returns an integer in the range of 0 to (M-1).

The two trees shown above could be whole trees or subtrees of a main function and an ADF, respectively. In English the main branch above would translate into "If Memory[3] **<** Memory[17] then return Memory[3] and add 1 to it. Otherwise return the result of sending the value of the UpperMiddle sensor and the value of Memory[5] to the ADF." The returned value of Memory[3] is its value before the increment. Memory[3] is not always incremented because the tree is not fully evaluated. That means that the test branch of the IF-THEN-ELSE is evaluated and then, based on the result, either the THEN or ELSE subexpression is evaluated, but not both. The choice of whether to do full evaluation of an expression is an important one whenever there are side-effecting terminals or non-terminals. Because the simulator created for these experiments does not fully evaluate an agent's function, the agent is allowed some "choice" about what and when to write to memory. In full evaluation there are more total writes to memory but there is no way for a function to avoid writing to memory under certain conditions. Indexed memory works well, but differently, in the case of full evaluation.

9.3 The Environment

Tartarus is an NxN grid. Approximately $(N-2)^2/3$ boxes are randomly distributed on the inner (N-2) x (N-2) grid . In this arena the fitness of a particular agent is tested. The

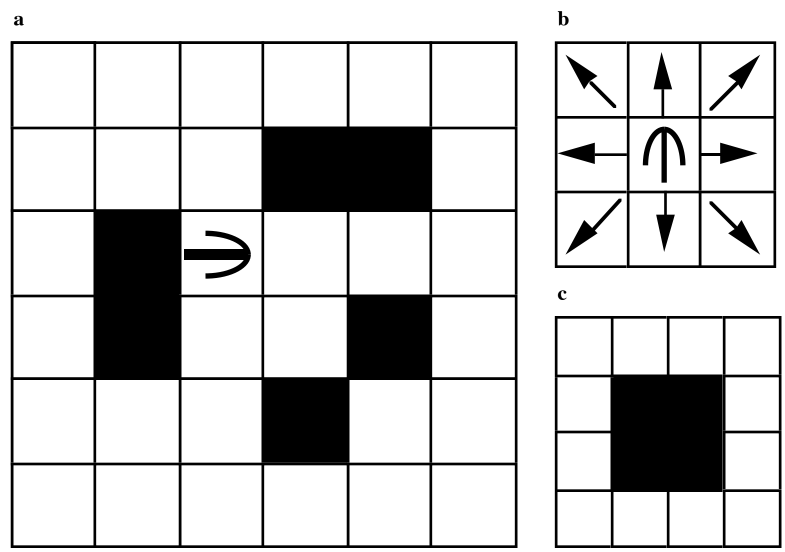

Figure 9.2

agent can be in any square that does not have a box and may face North, South, East, or West. N was chosen to be 6 in order to maintain computational tractability and to maximize certain criteria including the number of different possible initial configurations and low box density. This means that there are 6 randomly placed boxes on the inner 16 squares. At time step 0 of each fitness test, each agent finds itself on a random clear location facing in a random direction. In the discussion sections there will be mention of this choice and its ramifications. A typical trial might start as shown in figure 9.2.a.

The agent is given eight input sensor values for the eight nearest neighbors to the grid position it inhabits (figure 9.2.b). These eight inputs can take on one of three values: **Clear=0**, **Box=1**, or **Wall=2**. The **Wall** value (**2**) indicates that the position is outside the boundaries of the world. These eight input values can be used in the genetically programmed functions as eight terminals, called {UpperLeft, UpperMiddle, UpperRight, MiddleLeft, MiddleRight, LowerLeft, LowerMiddle, LowerRight}. Notice that these are relative to the heading of the agent; after an agent turns, while the environment will not change, one or more of the input values may have changed. This was chosen so that the agent could not figure out from the inputs which way it was facing. An agent has no knowledge of where it is or which way it is pointing unless it keeps track of that information using its indexed memory.

The agent's goal is to push all of the boxes out of the center onto the perimeter grid positions. At the end of a fitness test the agent is rewarded 2 points for every corner that contains a box and 1 point for every non-corner edge position that contains a box. Since no two boxes can occupy the same space there is a 10 point fitness test maximum.

The agent has only three possible actions: *MoveForward*, *TurnLeft*, and *TurnRight*. Because the agent is always on a clear square, turning left and right are always allowed. The agent is only allowed to move forward under two conditions: when the grid position in front is clear and inside the world boundary or when the position in front has a box but the space after that (in the same direction) is clear and inside the world boundary. In the first case the agent moves to the next square. In the second case the agent moves to the next square and the box is pushed one square forward. If the agent attempts to move forward into a wall or into a box which may not be pushed (either because it is against a wall or against another box) the action fails and the environment remains unchanged.

The main function for each agent does not return {*MoveForward*, *TurnLeft*, *TurnRight*}. It returns integers in the range 0 to (M-1). A filter is needed to map these numbers into one of the 3 possible actions. For M=20 the mapping was { Move (ReturnValue < 7), Left (6 < ReturnValue < 14), Right (14 < ReturnValue) }. Any partition with non-negligible ranges for each action would probably have worked.

The world is not always solvable by doing a random walk even in unbounded time, since there are many box configurations (e.g. figure 9.2.c) from which it is impossible to move any box to the wall. The simulator discards random initial environments with configurations like figure 9.2.c. Nothing, however, prevents agents from causing these configurations to occur during the fitness tests. Tartarus can, of course, be cleaned up more easily if the agent has a large amount of time. To keep the task challenging, turning left, right, and moving forward each take 1 time step (even when the move forward is unsuccessful). A complete tour of the board would take $N^2+2N-3 = 45$ and each agent is allowed only 80 time steps, somewhat less than two tours of the board.

Tartarus was designed to promote the intelligent use of state. The intention was to make a task with sufficient complexity in an environment with sufficiently random initial conditions so that agents who use memory would have a sizable advantage over agents who ignore their memory or who do not have memory. It cannot be overstressed how hard it is to achieve high fitness in the Tartarus environment without memory. Considerable effort has been put into writing a function that can reliably get more than 1 point every 80 time steps without using indexed memory. No human-generated or machine-generated program has yet materialized. As an example, imagine the set of conditions in which the agent should or should not move forward when there is a box in front of it and all its other sensors are clear. This could be the case when the box is against the wall, in which case moving forward would not change the environment and an agent without state information would spend the rest of the fitness test pushing the box against the wall. With the same input vector, the box could be against another box. Again, the agent that chooses to push in that situation runs the risk of spending the rest of the fitness test in futile pushing. Yet this input vector seems to be one in which pushing (moving forward into the box) is the best alternative.

It is, in fact, impossible to get better than minimal fitness without using state. Imagine an extreme case where the world is 1000x1000 and there is only 1 box somewhere on the inner 998x998 grid. Based only on the sensors it is, practically speaking, impossible to locate the box. The only other possibility is to do a pseudo-random walk and functional agents cannot do a pseudo-random walk unless they have access to a random walk action. This is a fundamental reason why Tartarus is a good environment in which to promote the use of state.

9.4 The Implementation

Table 9.1
Tableau for the Tartarus environment using Indexed Memory.

Objective:	Find an agent that performs significantly better than random or reactive agents competing in the Tartarus environment.
Fitness cases:	40 fitness cases in which the program has 80 time steps to move the 6 randomly distributed blocks to the edge and corners of the world.
Raw fitness:	Raw fitness is the sum of points earned in the 40 test cases. 1 point per box moved to the edge and an extra point per box moved into a corner.
Standardized fitness:	Standardized fitness: total possible points (400) minus the raw fitness.
Hits:	Same as raw fitness.
Wrapper:	{0 < MoveForward < 6} {7 < TurnLeft < 13} {14 < TurnRight < 19}
Parameters:	*Population* = 800. *Maximum Generation* = 100
Success predicate:	A program scores the maximum number of hits in all 40 test cases.
Overall program structure with automatic function definition:	One result-producing branch and one function definition called ADF which takes two arguments
Terminal set for the result-producing branch:	0...19 , UpperLeft , UpperMiddle , UpperRight , MiddleLeft , MiddleRight , LowerLeft , LowerMiddle, LowerRight
Function set for the result-producing branch:	NOT, OR, If-THEN-ELSE , LESS , ADD, SUB, EQ, READ, WRITE, ADF
Terminal set for the function definition ADF	0...19 , UpperLeft , UpperMiddle , UpperRight , MiddleLeft , MiddleRight , LowerLeft , LowerMiddle, LowerRight , Parameter1 , Parameter2
Function set for the function definition ADF	NOT, OR, If-THEN-ELSE , LESS , ADD, SUB, EQ , READ, WRITE

In most respects these experiments followed the standard genetic programming choices for various parameters. The population size used was 800. Empirically, any population size over 500 seemed to have sufficient diversity to keep the process going. The 800 initial individuals were created randomly. There was a 2 to 1 bias in the initial population randomizer in favor of the non-terminals READ, WRITE, and ADF and in favor of the input terminals. In the ADF there was a 2 to 1 bias in favor of using the terminals Parameter1 and Parameter2. Also, because 0 acts as False and all positive numbers as True, the constant 0 was as likely to come up as all the positive integers combined. The rationale for introducing this bias is that a solution to the Tartarus problem would probably contain more of these items (relative to others like the non-terminal AND or the terminal 6). This biasing does not warp the fitness space. At worst, it starts the population lower in the valleys. With effectively infinite diversity (i.e. population much larger than 800) this would have been unnecessary because there would have been no danger of "losing" a particularly useful terminal or non-terminal.

The maximum depth for the main function was set to 15 and the maximum depth for the ADF was set to 10. After the first 20 generations, the average number of nodes in a function stabilized at about 80 nodes for the main function and 30 for the ADF. The standard deviations were about 25 and 10 respectively. Fitness proportionate reproduction was used to create the mating pool. The crossover percent was 90. Forty-five percent were chosen in pairs and randomly selected subtrees were exchanged in their main functions. Another forty-five percent were similarly chosen and randomly selected subtrees were exchanged in their ADFs. One percent of each population was chosen, proportional to fitness, and mutated. If the mutated node needed a different number of subexpressions, the appropriate number of were added or subtracted. If subexpressions needed to be added, a subtree was created in the same manner that the original population was created. A non-elitist strategy was used, but the best of each generation was saved to a file for use in experimental results.

One fitness case is not sufficient to accurately determine the fitness of a particular individual. For each new generation, 40 environments were randomly generated and each member of the population tested on all 40 worlds. Given the large number of possible initial configurations, the chance that any agent was ever tried on the same environment twice is very close to zero. The number 40 was determined empirically as a reasonable trade-off between the need to get an accurate measure of the fitness for each agent and the need to keep down the run time for each generation. For each of the agent types described below, the runs each got about 10 hours of CPU time on a DEC5000/125. This usually achieved generation 80 for the mental agent with ADF and around 90 for the benchmark agents.

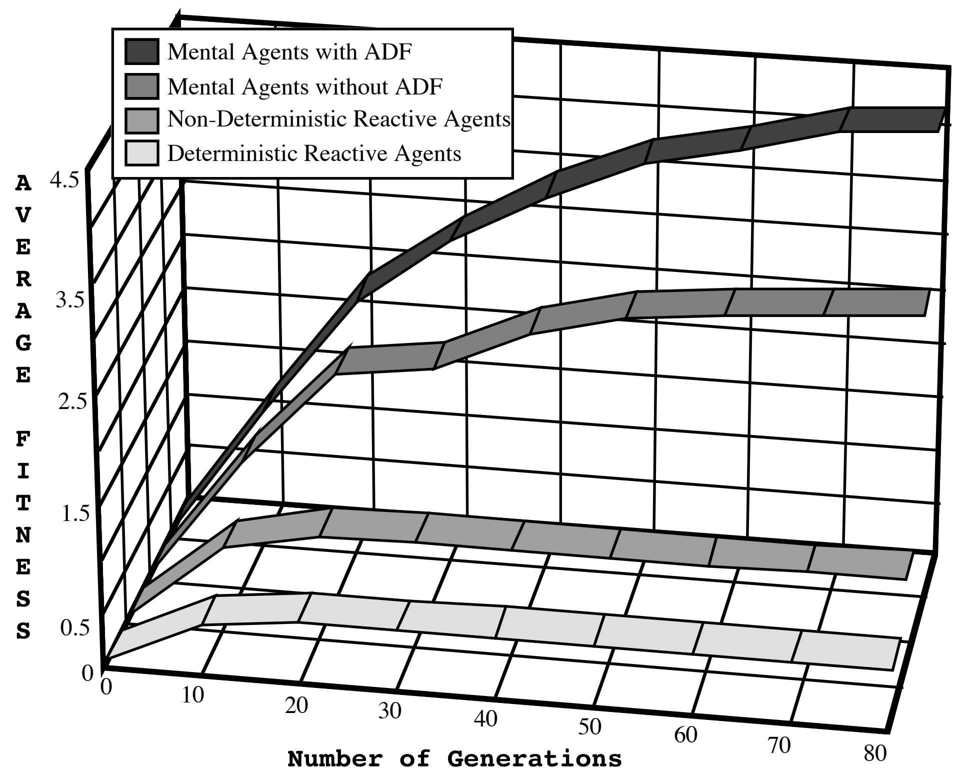

Figure 9.3

9.5 Experimental Results

So far this chapter has concentrated on a new method for incorporating memory into the genetic programming paradigm. A type of population individual was described and the following results focus on the success of this kind of agent. In order to fairly asses the merit of this type of agent, three benchmark agent types are introduced that are used for comparison in the experiments. The graph above shows the average best of generation for each of these four types. This average was taken across 10 runs, each of which was allowed to go to generation 80.

Agent Type1: Mental agent with ADF ==> This is the type already described.

Agent Type2: Mental agent with no ADF ==> This type is like the one already described in every way except that it has no ADF non-terminal symbol (and hence no ADF). This agent type is used to measure the relative effect of having an ADF.

Agent Type3: Non-Deterministic Reactive ==> This type has no state and no ADF. So READ, WRITE, and ADF are absent from its non-terminal set. Its filter divides the range of 20 up into four parts instead of three. The first three are still MoveForward, TurnLeft, and TurnRight. The fourth (ReturnValue > 15) causes one of the other 3 actions to be performed at random. This agent type is used to test whether the added ability to random walk is sufficient to solve the problem without state information.

Agent Type4: Deterministic Reactive ==> This type is the real benchmark. This agent type does not have state (READ and WRITE are no longer non-terminals) nor does it have ADF. And, unlike agent type 3, its actions are limited to Move, Left, and Right. It is over this agent in particular that an improvement is sought.

The most important observation to take away from the graph above is that the agent types with indexed memory do much better than the agent types with no access to state. The benchmark agent type 4 (Deterministic Reactive), as expected, is the least successful. Without state this environment is too complex to allow even marginal fitness improvements, and ,as a result, after 80 generations, its best of generation is getting an average of about half a point per fitness test. The random agent type 3 (Non-deterministic Reactive) does almost twice as well, but after 80 generations it still does not consistently get even one point per fitness case. While we can see that the ADF is very useful in this domain, it clearly is not the most dominant issue. The difference between the agent type 1 (Mental Agent with ADF) and agent type 4 is a factor of 8.5. The difference between agent type 2 (Mental Agent without ADF) and agent type 4 is about a factor of 6. But the difference between agent type 1 and agent type 2 is not even a factor of 2. The conclusion is that state information is critical to getting high fitness in this domain and that indexed memory is an effective evolutionary tool in achieving this use of state.

It is simple to determin if the agents that have evolved are using their memory; their memory can be switched off and their new fitnesses examined. Determining if these agents are using their memory in ways that correspond to features of the world is much more difficult, as we will see later in the chapter. In order to determine objective qualities of performance, it is necessary to perform experiments on a specific agent. The two following graphs show data taken for a typical agent type 1 Best of Generation 80. Its average fitness is 4.4 points per fitness test. This agent is shown directly after the graphs.

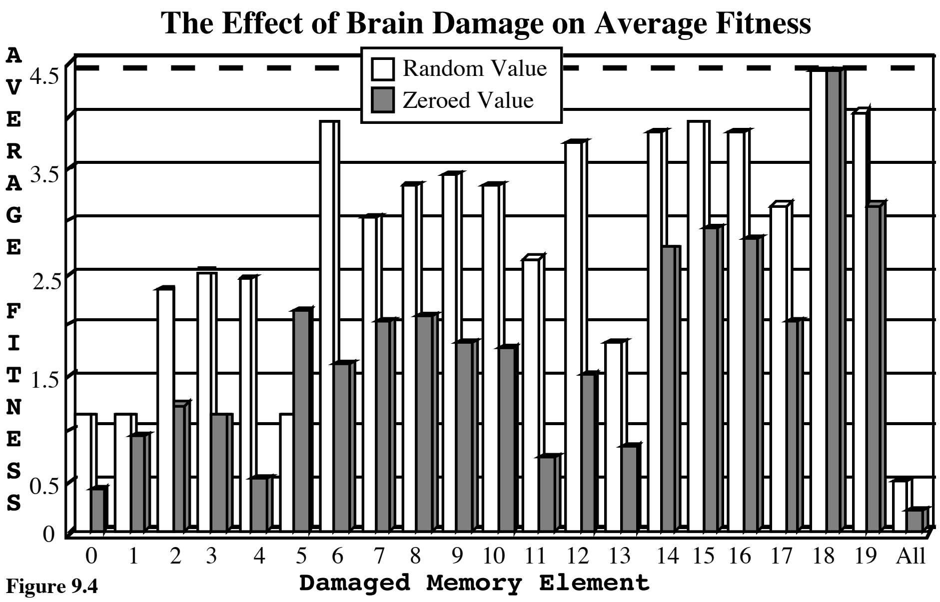

Figure 9.4

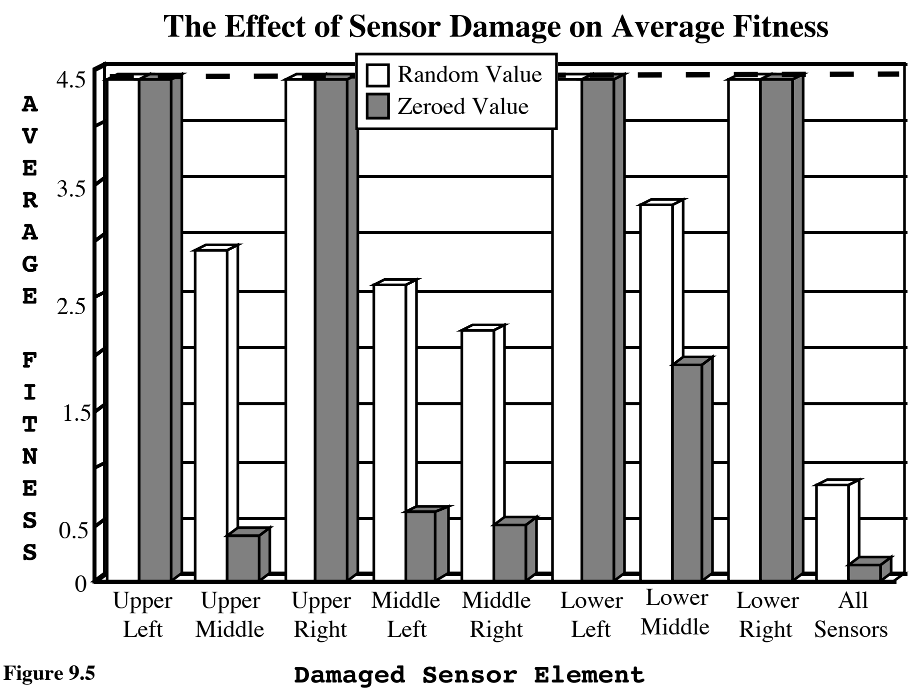

Figure 9.5

Automatically Defined Function

```
(WRITE (IF-THEN-ELSE (ADD (WRITE 0 Parameter1) (OR Parameter1 (READ
(WRITE (READ 0) (READ (ADD Parameter2 (ADD Parameter1 (READ 0))))))))
(ADD Parameter2 (ADD (ADD (READ (READ (READ 0))) (ADD Parameter1
Parameter2)) (ADD Parameter1 (READ MiddleRight)))) (WRITE (WRITE (WRITE
0 (WRITE (WRITE (OR (READ (WRITE Parameter1 Parameter2)) 0) Parameter1)
(WRITE Parameter1 Parameter2))) Parameter1) (WRITE (WRITE (READ 0) (READ
(WRITE Parameter1 Parameter2))) (WRITE Parameter1 0)))) (READ (READ
(READ (READ (EQ Parameter1 Parameter2))))))
```

Main Function

```
(ADD MiddleLeft (WRITE (SUB 3 (NOT (OR UpperMiddle (EQ 2 (IF-THEN-ELSE
(SUB (SUB (READ 17) UpperMiddle) 8) (*A-D-F* (EQ UpperMiddle (IF-THEN-
ELSE (READ 17) UpperMiddle 10)) (WRITE 10 UpperMiddle)) (LESS (WRITE
UpperMiddle UpperMiddle) (*A-D-F* LowerMiddle (WRITE UpperMiddle
UpperMiddle))))))))(WRITE 8 UpperMiddle)))
```

The first graph is an account, over many fitness tests, of how this particular agent fared with a form of brain damage. In each set of runs, some memory index **i** was selected and that memory index always returned a random number between 0 and 19 no matter what the actual value was. This was done for 100 random fitness tests, and an average fitness was found for the individual with that brain element damaged. This was done for each index **i** between 0 and 19. It is clear that this individual relies heavily on the values stored in certain memory positions. The same experiment was performed with the damaged memory index returning 0 independent of the true memory value. This caused greater reduction in the fitness of the individual. A possible explanation for why the constant misinformation was more deleterious to the performance than was random misinformation is that random numbers returned from memory may cause more random actions than the return of a constant. And in general, periods of random actions may help fitness, while periods of constant action usually lead to spinning in place or futile pushing against a box or the wall. When all of the memory indices were simultaneously subjected to the random or constant brain damage, the individual's fitness dropped to around 10% its normal fitness.

In the second graph we see the results of the complement to the brain damage study. This study subjects the same individual to sensory deprivation. Blinding each sensor input individually, 100 fitness tests were done and the histogram shows the resulting fitness averages. Each blinded sensor returned a random sensor value in the set {0,1,2}. The same tests were rerun and each blind sensor returned 0 regardless of the correct input. Here, as with the brain damage study, the sensor constant misinformation caused a greater loss of fitness than did the sensor random misinformation. A plausible explanation for the constant misinformation's greater effect on fitness may be that reacting to boxes is more important than reacting to clear spaces. When the "blind" sensor returns 0 all the time it is right an average of 63% of the time. When the "blind" sensor returns a random input it is right an average of 33% of the time. But the constant misinformation is never right in indicating there is a box (since 0 means Clear) and the random misinformation has a 33% chance of correctly indicating the appearance of a box at that sensor. When all the sensors were blinded simultaneously the individual's fitness dropped to 20% and 5% of the normal fitness (for the random and constant blinding respectively).

In this particular individual there are certain memory indices and certain sensors that are not used or, when damaged, do not adversely affect the individual's performance. For example, the particular individual shown above does not use any of its diagonal sensors {UpperRight, UpperLeft, LowerLeft, LowerRight}. This does not mean that these sensors are unnecessary. The UpperMiddle, MiddleLeft, and MiddleRight Sensors were the most popular among successful individuals from generation 80, but all of the other sensors were represented. Similarly, among successful generation 80 individuals positions 0,1,2,3, and 4 were the most heavily used memory positions, but all of the other memory indices were well represented.

9.6 Discussion of Indexed Memory

One of the advantages of the indexed memory strategy is its large number of possible states. Consider that for fewer than two dozen bytes of storage the agent gets over 10^{26} possible states (M=20) with constant access to 20 cross sections of the state space. Suppose we had a problem where there were 5 features to keep track of and each could be one of 4 values. With the indexed memory we could use 5 elements and store the values 0 through 3 in each one. Now this is wasteful in the sense that there are more compact representations, but the waste is a few states and this makes the use of state easier and less global in the following sense. Imagine a different memory strategy that has exactly 1024 different states (for example a recurrent neural network with 10 hidden units). There will be cases where a feature of the world changes and the agent switches from state 511 (0011111111) to state 512 (0100000000). This will require switching 9 of the 10 bits even though only one feature has changed. There are techniques, like the Grey code, which try to minimize the hamming distance between neighboring states. But if the order of state changes is not known ahead of time, it is hard to find a representation that will not sometimes have to change many or all of its elements just to effect a single change. Another way to say this is that redundancy brings with it a certain flexibility. This flexibility is the increased number of possible representations of a set of states. Since in using memory, the evolutionary process is trying to find some workable representation of the useful environmental features, more possible representations is better. Since we pay almost nothing for all the extra state in indexed memory, we effectively get the flexibility for free.

The efficacy of indexed memory is affected by the choice of M=20. Several other choices for M were investigated. For values of M less than 8 or 9 the performance noticeably decreased. For values of M larger than 40 or 50 there was also a slight decrease in the rate of increase of average fitness over generations. For low M the performance loss is probably attributable to insufficient memory space. When M is 5 there are only 3,125 possible states (5^5) and using all of them introduces the tight representation problem described above. A solution would be to have only 5 memory elements but to allow each element to store a larger range of integers. This solution would at least increase the number of available states. However, this solution would complicate indexing (Read X) by making the effective index be (X modulo M). Also, having a few large valued slots might still lead to the tight fit representation problem mentioned above. For large M the opposite problem occurs: there are too many memory elements. As a result, in early generations there is a lower likelihood of reading from the same memory element that useful information was written to. After a few generations certain memory elements become popular in sections of the population and this problem largely disappears. Part of the explanation for why certain memory elements become popular is that there is a bias in these experiments towards the numbers 0,1, and 2. All the boolean functions return 0 or 1 and the input values are {0,1,2}. For example, the typical agent shown above makes heavy use of memory positions 0,1, and 2. But they are not the only heavily used memory indexes for this individual; indices 3,4,10, and 13 are also particularly important. Though M was chosen to be 20, indexed memory is a general scheme which will work for any value of M, whether the element values have the same range as the memory index or not.

Along with the issue of the particular choice for M, the initial state of the memory elements at the beginning of each fitness is of some importance. For these experiments the memory elements of an agent were all set to 0 at the start of each new fitness test. Trials were done with other homogeneous constants and with fixed distribution of elements in the range 0 to M-1 (e.g. Memory[0] = 5, Memory[1] = 19, ...). As long as the configuration of the agent's memory was always the same on time step zero, the particular set of memory values made little difference. The future works section will mention experiments that go beyond these constraints.

The argument for the usefulness of an overabundance of available state is not the only important feature of the indexed memory paradigm. Another advantage is that for a non-trivial problem it is difficult to know how many states will be necessary. Picking some state size as a guess at the approximate amount of state that will be needed not only burdens the evolutionary process with the tight-fitting representation problem mentioned above, but runs the risk of having too few states and making high success at the task impossible. There will always be features of an environment whose values in a mental model are dubious, but by using indexed memory, part of the memory space can be used as a scratch pad for calculations and for those features whose values are only loosely tied to success in the domain.

Indirection is another critical aspect of indexed memory. While no additional constructs are necessary to facilitate (Read (Read X)), this adds a powerful ability to the memory system. Without indirection, data structures might evolve in the agents memory systems, but it would in general take linear time in the size of the memory to extract information from an element of a dynamic data structure. Think of a dynamic memory data structure as a variable sized area of memory containing closely related information which is larger than one memory element. With memory indirection, mental models that contain linked lists, queues, stacks, and other data organizations become not only possible, but relatively simple.

Indexed memory will work for a wide variety of problems requiring state or mental models. For example, the Prisoner's Dilemma problem has already been mentioned as a problem for which memory is useful. Indexed memory could easily accommodate keeping the last k moves as long as $k < M$ and could probably find encoding that would work for $k>M$. As an example the function could maintain a queue by moving Memory[2] into Memory[1], Memory[3] into Memory[2], ... until Memory[k] was moved. Then it could put its most recent input (the past move) into Memory[k]. Another strategy could be to use a circular queue and leave out Memory[0] and Memory[1] as pointers to the head and tail of the queue. These two examples show that indexed memory is a sufficiently powerful tool for the problem. Undoubtedly the evolutionary process would come up with an equally valid technique that would not map into any familiar algorithm.

Indexed memory is not only powerful enough to solve the Prisinors Dilemna, it can be shown that indexed memory can support **any** data structure and carry out **any** computation. This proof, that a genetically evolved function with indexed memory is Turing-complete, is too long to include here in full. The intuition behind the proof, however, can be given in a few lines.

The essence of the action of a Turing machine is the **delta** function which maps the machine's current state and current tape position symbol to a new state, a new tape head position, and possibly a new symbol at the old tape position. Imagine that M is allowed to be any integer and its memory is expanded dynamically when a memory position is read or written which has not before been accessed. Now set aside two memory elements (e.g. Memory[0] and Memory[1]) and define all other elements with a strict ordering. Let Memory[0] hold the "state" of the Turing machine and Memory[1] hold the position of the "tape head." Now it is clear that the rest of the indexed memory can be used as a tape and that the genetically evolved function is exactly that **delta** function. The function maps the state (Read 0) and the current tape position symbol (Read(Read 1)) to a new state (Write NewState# 0), possibly a new symbol at the old tape position (Write NewSymbol# (Read 1)), and a new tape position. To move the tape head right or left the **delta** function will need to skip from -1 to 2 and 2 to -1 respectively. Here is how the **delta** function might move the tape head left:

```
(IF (EQ (Read 1) 2) THEN (Write -1 1) ELSE (Write (Sub (Read 1) 1) 1))
```

So now it remains only to show that the **delta** function can always be evolved using GP with indexed memory. There are a finite number of states, and a finite number of elements in **sigma** (the tape alphabet). So the mapping from <Current state, Current tape symbol> to <New state, New tape symbol, New tape head position> can always be done with a finite number of IF-THEN-ELSE's. So as long as the number of Turing machine states is bounded in advance or there is no limit on the size of the evolved function, **delta** can always be produced. Just like **delta** from a Turing machine, the GP function will be repeatedly called until it returns a reserved integer T (e.g. 1045047) which signifies termination.

Since indexed memory is Turing-complete, genetically evolved functions that use indexed memory can simulate any machine and run any algorithm. Furthermore, because these programs are being evolved in the complete space of algorithms, this means that, in principle, we can evolve any program we need without ever expanding upon the simple set of terminals and non-terminals described in this chapter.

9.7 Discussion of Mental Models

In the experiments section an agent was presented that attributed signs of a mental model. This agent's performance dropped significantly when particular sensors failed to accurately report the correct feature value of the environment. The agent's performance also suffered when it could not reliably recover the same value it had stored in particular places in its memory. These conditions are neither necessary nor sufficient criteria for showing that the agent has an iconic or a sentential model of the world. There is little reason, however, to look for these human-centric mental attributes. No part of the fitness function rewards "thinking" that falls into line with how we speak about thinking. Before we ask about some highly evolved agent, "what is it doing?" we must consider what kind of answer we expect. It seems possible, given the results above, that the current language of mental activities does not allow us to speak fruitfully about what that individual is doing. This research is, therefore, about mental models and not about iconic or sentential models specifically. Given that these agents are not human and that the human notions of iconic and sentential models have no context-independent justifications, it may be unjustified to ascribe iconic or sentential models to these agents' mental behaviors.

Whenever we try to find meaning in the actions of agents other than ourselves, we have to doubt what we see. There is in fact no perfect way of determining whether an agent has or is using a mental model of the world. During the course of the research and quite by accident, the agent shown below was noticed, whose average fitness is about 5 points per fitness test. This agent evolved under slightly different conditions than those described in this chapter, but its existence is instructive. The environment this successful generation 80 individual evolved in differed from the environment that this chapter details in that agents always started in the upper left corner of the world facing east.

Automatically Defined Function

```
(NOT (READ (WRITE (WRITE Param1 (WRITE (WRITE (WRITE Param2 Param1)
(WRITE Param2 Param1)) (NOT (READ (WRITE (WRITE (WRITE Param2 Param1)
(WRITE 12 Param1)) 0))))) 0)))
```

Main Function

```
(READ (OR (READ (READ (READ 0))) (WRITE (*A-D-F* 10 (READ (READ (OR
(READ (READ (READ (READ (READ (READ (READ (READ 0)))))))) 0)))) (SUB 0
(READ (READ (ADD 0 (READ (SUB (READ 0) (READ 0))))))))))
```

Notice that this agent never examines **any** of its sensors. This agent **does** make use of state and the results of the brain damage study on this agent are similar to the results shown in the previous section. However, the sensory deprivation tests have no effect on this agent. It scores this high level of fitness by tracing out a complicated, variable path that it has evolved to use. Increasing the size of the world to 7x7 drops its fitness to an average of 3.1 points per fitness case. This shows a certain robustness in this variable pattern method. Starting this agent in a random clear location with a random heading in the 6x6 world dropped this agents fitness to an average of 1.6 per fitness case, which is lower, but still high considering its apparent brittleness. This agent, using state very effectively to pace out its pattern, has evolved a sort of "built-in" model of the world.

The existence of this agent does not mean that this research did not succeed in producing agents with a mental model of the world. But it does remind us that the problem of determining whether an agent is "thinking" is hard enough that behavioral evidence is not sufficient and that there may, in general, be no conclusive way of determining the level of attention an agent pays to the world in which it is acting.

From a certain point of view this sensorless agent has "learned" more than some of its fellow best-of-80th generation agents (though in a simplier domain). It has evolved a general strategy that works well no matter what configuration of boxes is presented. When the starting location and heading for the agent were randomized, no reoccurrence of this type of successful sensor-insensitive agent was discovered. But we can never rule out the possibility of the appearance of such an agent. This is not the kind of mental model that this research was trying to induce, but this agent serves as a reminder that despite the complexity of an environment, there are often simple solutions quite unlike any human approach to the problem. And if these simple solutions do exist, they will probably evolve.

9.8 Future Work

The results described in this chapter lead toward at least two different lines of the research. The first is the use of techniques, like the one presented here to create or evolve mental models and to design experiments to study these models. An interesting step would be to make some distinction between simple use of state and a mental model based on phenomena observed in the memory units. Another would be to eliminate the types of environmental features that allowed the agent with a "built-in" model to appear. As was pointed out, though, this type of agent may be impossible to completely prevent in general. The configuration of an agent's initial state of memory as it begins each new fitness test also plays a role in these questions. Work has already begun to study various kinds of learning using the persistence of memory across fitness tests as well as the Baldwin effect, Lamarkian evolution, and the evolution of the initial memory state itself.

The second line is to continue to improve upon indexed memory as a paradigm in genetic programming or to expand this work in some natural way into genetic algorithms. The introduction of natural uses of memory in these environments may eventually lead to a broader range of problems that can be undertaken using evolutionary strategies.

To substantiate the claim that indexed memory is truly a general, practical solution to the problem of incorporating memory into genetic programming, it must be tried on a variety of other memory-critical problems. Work has already been started using indexed memory and the same non-terminal set discussed here to find solutions to the problem of learning a maze, the Prisoner's Dilemma, and a modified version of Simon-Says. The inputs are different for each problem and the filters map the integers returned into one of the possible actions. It is important to note that while GP with indexed memory is Turing-complete, there is no guaranty that it is an efficient method for evolving Turing machines. The crucial future research question will be "Does the generality of indexed memory justify the low level at which it operates ?"

9.9 Conclusions

An important goal of this research effort was to evolve agents that use state effectively and build mental models of their environment. As was shown using the brain damage and sensor damage graphs, the loss of particular memory elements or sensor elements results in significantly lower fitness. This means that memory and sensory inputs play a pivotal role in that individual's success. It was argued that a fitness dependence on inputs that correspond to the real feature values from the environment and a fitness dependence on the consistent ability to retrieve from memory the same value that was placed there, implies some non-trivial mental model. The fact that this use of memory cannot be described in terms of the familiar forms of mental activity suggests a need not only for further investigation of the agents, but also for a new way to talk about mental activities themselves.

The abilities of another successful individual were then discussed. This one used state very effectively but completely ignored its sensor values. The suggestion is that while this agent does not invalidate the conclusions just summarized, it brings into question our ability to reliably tell when an agent is "making a mental model" and when it is using its state in ways that are successful but have little or no correlation to the current feature values of the world. It was also posited that this second individual had a different kind of "built-in" model of the world and that it had found a "general" solution to a simplier version of the same domain.

These results were produced using the new technique of **indexed memory**. Indexed memory succeeds in this domain and will succeed in others for three main reasons. First, the large amount of state made available to the evolving agents makes it unnecessary for evolution to find compact representations for stored information. Second, the ability to have arbitrarily deep levels of memory indirection (i.e. (Read (Read (....))) facilitates the efficient creation of pointers and more complex data structures. And, most importantly, indexed memory is Turing-complete; it is theoretically sufficient for the evolution of any algorithm.

Work in progress includes the use of indexed memory to solve several familiar memory-based problems and suggestions were given about how indexed memory might be used by evolving systems to solve problems in other domains. The intuitions were given to a proof that index memory is Turing-complete and ramifications of this proof were discussed.

It has been argued that this research saw the emergence of mental models in agents that evolved from functional agents with access to indexed memory. These behaviors could not have come about without some type of simple, malleable, state information. As a representative of this type of state information system and as an addition to the genetic programming paradigm, indexed memory stands in a position to advance work done on the genesis, types, and structures of mental models and to open up new types of problems to the field of genetic programming.

Acknowledgments

I benefited greatly from discussions I had on this subject with John Koza, Nils Nilsson, and many others. The original motivation for this research came from a problem John Koza and Nils Nilsson brought to my attention which they call "the 7-layer Cake" problem. It was in quest of progress on this domain, the evolution of intelligent use of state, that this chapter came about.

Bibliography

Jefferson, David, Collins, Robert,. "AntFarm: Toward Simulated Evolution." In Langton, Christopher, et al. (editors), *Artificial Life II*. Addison-Wesley. 1991.

Jefferson, David, Collins, Robert, Cooper, Claus, Dyer, Michael, Flowers, Margot, Korf, Richard, Taylor, Charles, and Wang, Alan. "Evolution as a theme in artificial life: The genesys/tracker system." In Langton, Christopher, et al. (editors), *Artificial Life II*. Addison-Wesley. 1991.

Johnson-Laird, P.N. *Mental Models: Towards a Cognitive Science of Language, Inference, and Consciousness*. Cambridge, Mass. Havard University Press. 1983.

Koza, John R. *Genetic Programming: On the Programming of computers by means of natural selection*. The MIT Press. 1992.

Lindren, Kristian. "Evolutionary Phenomena in Simple Dynamics." In Langton, Christopher, et al. (editors), *Artificial Life II*. Addison-Wesley. 1991.