EVOLVED RECURSIVE DISTRIBUTED REPRESENTATIONS

An Honors Thesis

Presented to

The Faculty of the Department of Computer Science

Washington and Lee University

In Partial Fulfillment of the Requirements for

Honors in Computer Science

---

by

Matthew Victor Grieco

May 2003

To the Numbers "1"and "0"

Contents

List of Tables

List of Figures

ACKNOWLEDGMENTS

I would like to thank Jeff Meriggi for proofreading this work. Also, I would like to thank Professor Tom Whaley for helping me get start on this thesis a year ago. Most importantly, I would like to thank my advisor Professor Simon Levy for his guidance and support throughout the year.

ABSTRACT

Computer Science has become increasingly biologically inspired, with neural networks and evolutionary computing gaining increased prominence in the field. Recently, several difficulties with modeling connectionist networks and using genetic algorithms to train neural networks were resolved (or at least the topic of several studies). These problems included how to create a connectionist model that could represent variable-sized recursive data structures in fixed-width patterns and how genetic algorithms were used to evolve neural networks. However, no coherent union of these problems has been made. The purpose of this study is to compare Back-Propagation, the classic way to train connectionist networks, to those Genetic Algorithms used for evolving weight dynamics, a novel way to train connectionist networks on a Recursive Auto-Associative Memory (RAAM) neural network architecture. The ability of a Genetic Algorithm to train a RAAM network is tested through two different problems. The first comparison is made through a simple auto-association problem where the fitness of networks trained by Genetic Algorithm's is compared with the fitness of networks trained by back-propagation. Since back-propagation has already been established to create RAAM networks that can solve novel tasks, it is used here as a benchmark with which to compare RAAM architectures created with evolutionary techniques. Secondly, a Pursuer/Evader Game is played by RAAM networks trained with a genetic algorithm. The results show that Genetic Algorithms are an effective method to examine the complexity found in a RAAM network.

EVOLVED RECURSIVE DISTRIBUTED REPRESENTATIONS

Chapter 1

Introduction

Computer Science has become increasingly biologically inspired, with neural networks and evolutionary computing gaining increased prominence in the field. Recently, several difficulties with modeling connectionist networks and using evolutionary algorithms to train neural networks were resolved (or at least the topic of several studies). The first of these problems was how to create a connectionist model that could represent variable-sized recursive data structures in fixed-width patterns [5]. The other problem lay in how evolutionary algorithms were used to evolve neural networks [1]. This thesis uses an evolutionary algorithm to explore the origin of complexity with respect to communication.

The algorithm used is a Genetic Algorithm (GA). GAs have been proved as an effective method for solving computational problems [4]. Following [2], three populations of agents are evolved in a Pursuer/Evader game. Fitness is awarded to an agent based on its ability to compete in the Pursuer/Evader game.

An initial experiment tests the validity of a GA to train the agent architecture by comparing it with a known method (Back-Propagation). The agent architecture is a recurrent

neural network capable of encoding and generating binary trees and strings of an arbitrary size.

Through this thesis the ability of GAs to evolve connectionist networks that represent variable-sized recursive data structures in fixed-width patterns is tested.

Chapter 2

RAAM Architecture

RAAM or Recurrent Auto-Associative Memory is a method for representing variable-sized symbolic sequences in a fixed-width form [5]. The RAAM network architecture, trained with back-propagation, has been shown to work for high-level cognitive tasks, such as Natural Language Processing, because it is able to capture the dynamically-allocated variable-sized symbolic data structures traditionally used in AI [5]. A RAAM Network is divided into two similar parts, the Encoder and the Decoder (fig. 2.1). The Encoder essentially computes the inverse of the Decoder.

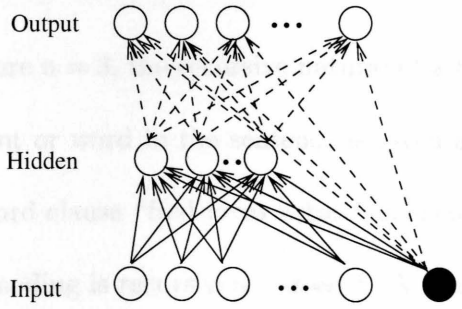

Figure 2.1: RAAM Network.

A RAAM network has a unique relationship between the input, hidden and output layers. There are $n*$k input/output nodes and k hidden units, where $n \in N$. Because of the $n:1:n$ ratio of input to hidden to output units the encoding and decoding procedure can repeated recursively. [1] The k unit encoding of nk input units is recursively feed back to the encoder. This is seen in the encoding of the sentence, "barney knows fred loves wilma". (fig. 2.2).

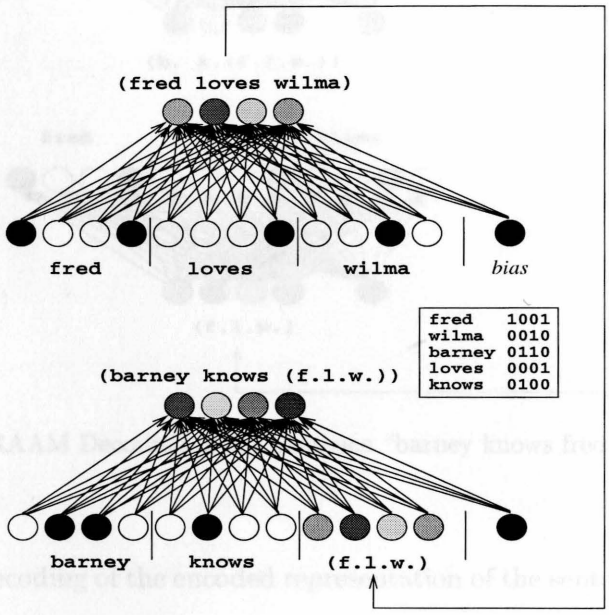

Figure 2.2: The RAAM Encoding of the Sentence "barney knows fred loves wilma."

In this example, where $n = 3$, the recursive natural of a RAAM encoding can clearly be seen. First, each element or word in the sentence is given a four bit symbolic representation. Then the three word clause "fred loves wilma" is encoded into a four-element vector representation. This encoding is recursively passed back into the RAAM network with the

[1] The figure shows an example with $k = 3$. Since a binary tree can represent the same information as a k-ary tree, we used $k = 2$ for the experiments described in the following chapters.

rest of the sentence to be encoded.

The Decoder works in the same recursive fashion (fig. 2.3). k hidden units can be decoded into $n*$k output units. Each portion of the output can then be decoded recursively.

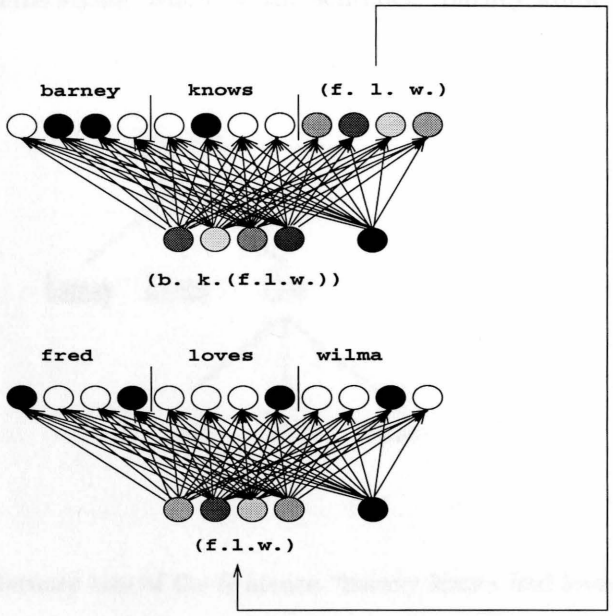

Figure 2.3: The RAAM Decoding of the Sentence "barney knows fred loves wilma."

This is seen in the decoding of the encoded representation of the sentence "barney knows fred loves wilma" First the full encoding is decoded to the symbolic representation of the words "barney" and "knows" along with the encoding of the clause "fred loves wilma." This clause is then recursively passed back into to the decoded to be resolved into the binary symbolic representations of its atomic elements. The difficulty with the process is how to determine when to stop decoding. The recursive decoding theoretically stops when the output no longer contains floating point numbers but only binary strings, which are the original symbolic sequences [5]. This result is the ideal, but in practice, it is unlikely that the decoding process while ever go completely to a binary string. A threshold is used to

determine when the floating point numbers have become sufficiently close to binary strings.

The Decoder can be considered to produce either trees or strings from the frontiers of trees. In the example above the decoder would produce a string from the frontier of a ternary tree (fig. 2.4). This string would be the sentence "barney knows fred loves wilma."

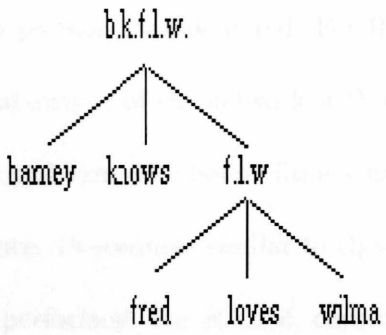

Figure 2.4: Ternary tree of the Sentence "barney knows fred loves wilma."

Neural Networks, such as RAAM, need to be trained in order to perform their tasks correctly. The training involves adjusting the set of weights for the network's connections. In this study, training is accomplished by one of two methods, a classical hill-climber method (Back-Propagation), or a novel evolutionary method (Genetic Algorithms).

Back-Propagation is the classical way to train neural networks. At each time step in the training of the network, the weights of the network are adjusted. The adjustment occurs by comparing the experimental output of the network with the known output for the training set. The weights leading into each unit of the output are adjusted by a specific ratio based on this comparison. The adjustments then propagate backward through the network [6].

Genetic Algorithms (GAs) are used for solving practical problems along with compu-

tational models of natural evolutionary systems [4]. They are part of the broader field of evolutionary computation. GA researchers view problems and problem solving in much of the same fashion that modern biologists view evolution. A set of possible solutions is viewed as a population. When dealing with a neural network a member of the population is the set of weights in the network. At each generation the fitness, or ability, of each member of the population to solve the problem, is calculated. For RAAM, the fitness is calculated by comparing the experimental output of the network with the known output for the training set. The members of the population with better fitness survive while the rest are discarded. Those that survive reproduce. Procedures similar to those of cross-over and mutation that occur in chromosomes are performed. For RAAM, chromosomes are weights. GAs provide a method of searching among an enormous number of possibilities for a highly fit solution or for designing innovative solutions to complex problems [4].

The first experiment was designed to address the question of whether RAAM can be trained through an evolutionary method. An experiment is performed to compare Back-Propagation with a Genetic Algorithm based on their ability to train a RAAM network.

Chapter 3

Experiment: Genetic Algorithm versus Back-Propagation

This experiment compared the training of RAAM networks using BP against GA. The success of each method was simply the root mean squared (RMS) error between the encoder inputs and the decoder outputs, using the tree training set:

$\{(ab), (a(bc)), ((ab)(cd))\}$.

BP takes two parameters *Eta* and *Mu* that determine how the solution space is searched. The *Eta* parameter is the learning rate of the BP algorithm and determines how much a weight changes. The *Eta* parameter is a constant proportion between the change in weight and the error of the network. Since the error of the network and the constant of proportionality are known, the change in weight can be found. The *Mu* parameter is the momentum of the learning function. The momentum parameter determines the effect of the previous weight change and effectively filters out high-frequency variations of the error-

surface in the solution space [6]. GAs take two parameters, *PC* and *PM*. The *PC* parameter is the probability that cross-over will occur between the weights of two parent networks to create the weights for their two offspring networks in the subsequent generation. If no cross-over occurs, the parents are simply copied into the next generation. The *PM* parameter is the probability that mutation will occur to a given weight of a network. If mutation does occur, a random Gaussian is added to the weight [4].

3.1 Results

BP and GAs were used to train/evolve a population of RAAM networks. Nine different test conditions of the parameters that govern how BP and GA work were tested. For each set of test conditions five runs were performed. Each run of either BP or GA consisted of allowing each method to train a population of networks for twenty minutes, after which time the success of the algorithm was reported.

The results from using BP to train RAAM networks are in Table 3.1 and the results from using GAs to train RAAM networks are in Table 3.2. The success of BP is one minus the error of the network. It is written in this manner so that the comparison with the GA is clearer.

Statistical analysis of these results confirmed an intuitive analysis of the data. For BP, neither the learning rate nor momentum is significant (in terms of final success). For the GA, mutation is significant ($p<.05$), but crossover is not. GA with PM = 0.1 was the most successful (fig, 3.1). Comparing BP to GA, the difference is significant at $p<.0001$ (the F ratio was extremely high, around 7700). BP had a higher success, see fig. 3.2.

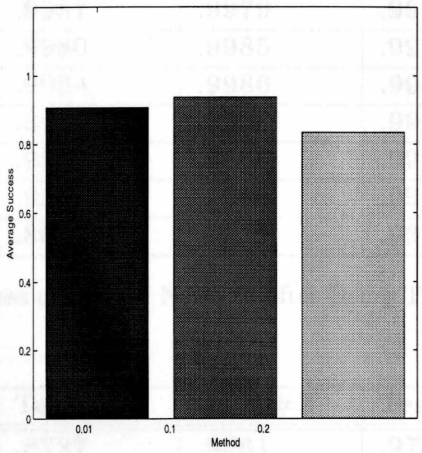

Figure 3.1: Average Success of GA based on PM.

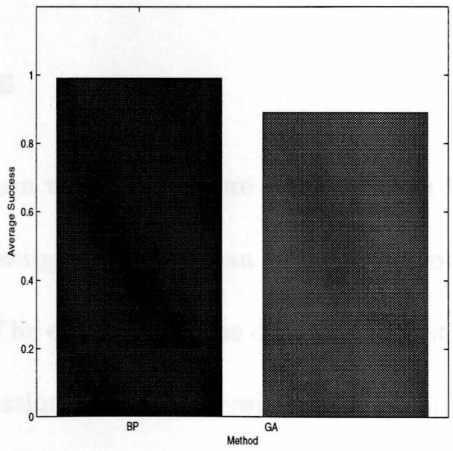

Figure 3.2: Average Success of BP compared to GA.

Test Conditions	Test #1	Test #2	Test #3	Test #4	Test #5
Eta = .1 Mu = .1	.9967	.9964	.9132	.9967	.9934
Eta = .1 Mu = .5	.9982	.9984	.9980	.9980	.9980
Eta = .1 Mu = .9	.9987	.9979	.9983	.9989	.9983
Eta = .5 Mu = .1	.9980	.9985	.9983	.9971	.9984
Eta = .5 Mu = .5	.9984	.9986	.9984	.9975	.9511
Eta = .5 Mu = .9	.9984	.9986	.9990	.9459	.9967
Eta = .9 Mu = .1	.9983	.9986	.9988	.9987	.9988
Eta = .9 Mu = .5	.9985	.9986	.9991	.9822	.9990
Eta = .9 Mu = .9	.8654	.9982	.9976	.9992	.9989

Table 3.1: Mean Success of RAAM Network after Being Trained by the BP Algorithm

Test Conditions	Test #1	Test #2	Test #3	Test #4	Test #5
PC = .7 PM = .001	.8727	.9061	.9741	.8783	.8941
PC = .6 PM = .001	.9323	.9192	.8333	.9396	.9851
PC = .8 PM = .001	.8961	.9079	.8031	.9121	.9549
PC = .7 PM = .1	.8557	.9949	.7871	.9088	.7959
PC = .6 PM = .1	.9943	.9951	.9144	.9988	.9824
PC = .8 PM = .1	1.000	.9765	.8528	1.000	.9990
PC = .7 PM = .2	.7118	.9978	.7133	.6614	.8429
PC = .6 PM = .2	.9697	.8527	.7484	.8472	.9908
PC = .8 PM = .2	.8176	.9994	.7233	.7959	.8519

Table 3.2: Mean Fitness of RAAM Network after Being Trained by a Genetic Algorithm

3.2 Conclusions

Here it can clearly be seen that BP is more effective than GAs to train RAAM networks. However, the goal of showing that a GA can be used to evolve a RAAM network to solve a problem was achieved. The experiment was designed to test the effectiveness of GAs versus BP in a real-world situation where processing time is limited. However, BP worked on a population of one RAAM network while the GA worked on a population of 100 RAAM networks. Thus, for each generation of the GA more computations were required. Also, as a result of using roulette selection in the GA to determine which members of the population

will reproduce in the next generation the GA frequently got caught in local maximums. Perhaps, if another selection method were used, such as tournament selection, the GA would have been able to escape these local maximums and find the ideal solution.

Although the difference in success between BP and a GA is highly significant ($p<.0001$) the GA was effective at evolving a RAAM network. Thus we can use a GA for a RAAM application in which BP is not possible, such as in the pursuer/evader game discussed in the next chapter.

Chapter 4

Co-Evolution Experiment Design

Since GAs were shown to be reasonable successful in evolving RAAM networks, other interesting population based problems could be investigated using RAAM and GA to get a true measure of success. This chapter describes one such experiment.

The experiment involves a Pursuer/Evader Communication Game. The basic Pursuer/Evader (PE) Game consists of two agents, an evader and a pursuer, that interact in discrete time and space [2]. The evader generates a set of binary strings that the pursuer must match at each time step.

In each match or bout between the pursuer and evader, each agent receives a score. The pursuer's score is the number of correctly matched strings, its ability to "pursue" the evader. The evader's score is the number of strings that are not matched, its ability to "evade" the pursuer.

In the Two-Player PE Game there are two agent populations, the evaders and the pursuers. In the Three-Player PE Game, used in this experiment, there are three agent populations, the evaders and two sets of pursuers. One pursuer is friendly and with whom

the evader wants to match. The second pursuer is hostile and acts in the same manner that the pursuer in the Two-Player Game acts. There are two intrinsic flaws with the Two-Player PE Game that are solved by moving to a Three-Player Game. First, it is not biologically realistic. Imagine the evaders to be individuals that literally were under constant predation who must, therefore, mate while on the run. But, the mate must be able to keep up with the evader. The friendly pursuer must match the evader in order for mating to occur. Secondly prediction is intrinsically more difficult than generation. The evader may start generating a set of strings that are too complex for the pursuer to predict making the results uninteresting. If the evader is required to be predictable to someone, it will stay bounded, allowing for interesting results to appear [2]. Populations of 100 were created of each of these PE Game agents. In a bout between the three agent,s each agent would generate a variable number of strings of "a", "b", and "c", of variable length and complexity. A pursuer would receive a numerical score of the number of correctly matched strings. The evader received a score based on the score of the friendly pursuer it was playing minus the score of the hostile pursuer it was playing.

The communication between agents uses sequences of strings, not trees, which a RAAM decoder can generate. If we treat the decoder as an Iterated Function System (IFS) [3] shows that high (infinite) generative capacity is possible.

The fractal attractor set is created using the iterative nature of IFS's. A 2-D space is created to which the left and right transforms (decoder weights and sigmoidal squashing function) are applied. The left and right transform create two new points from the original by stretching it and then shrinking it. Ideally, this space simple would have infinite precision. However, because of the finite memory and precision of computers, this cannot

be accomplished. When the left and right transforms are applied to the entire 2-D space, a subset of the space is created. The new set is a subset and not a superset of the original as a result of the IFS. Each transform first stretches the original point in two directions. Then a modified sigmoid squashing function is applied so that the resulting point is within the original 2-D space. The union of the left and right transforms being applied to the 2-D space is a rough approximation to the fractal attractor. The image of the attractor is clarified by iteratively applying the IFS to the result of the previous application of the IFS. The IFS stops running when a subsequent run of the IFS will not more clearly define the attractor. As discussed above, the fractal attractor is the result of executing the IFS for an infinite number of time steps. This, however, is unrealistic. Instead, the computer has only finite memory and precision. The attractor is defined to finite precision. The process is illustrated in fig. 4.1, using a set of transform weights that produces a pleasant looking attractor.

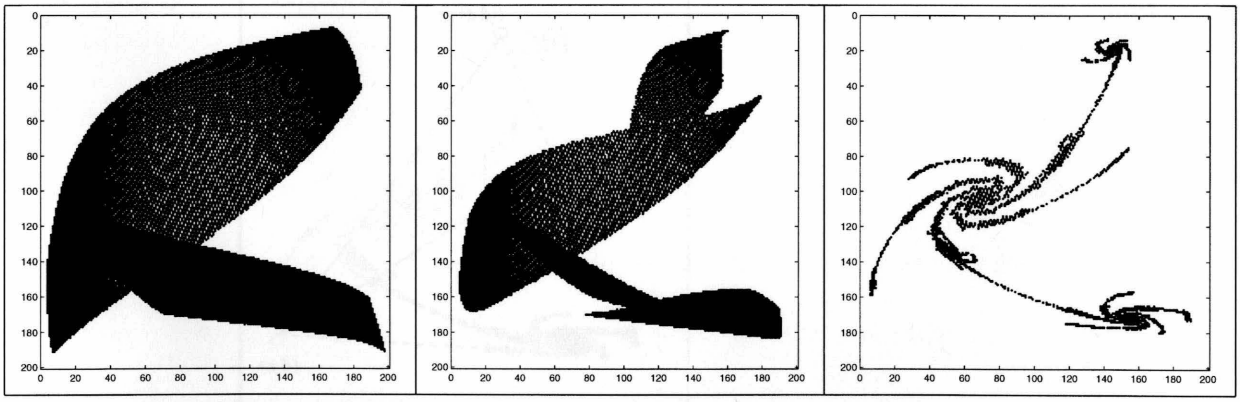

Figure 4.1: Three stages in the generation of an IFS attractor. Left: Two transformed copies of the unit square are combined to form a single image. Middle: Two copies of the image at left are combined to form an image that better approximates the attractor. Right: When the transform and combine operations no longer change the image the final attractor approximation is obtained.

Once the fractal attractor has been created ,the strings are generated as follows:

The 2-D space is divided into a finite number of discrete points. The IFS is applied to each point. A tree for the IFS decoding of each point is created. The root of the tree is the initial point. The left and right branch of the tree is the point in space reached from the left and right transform respectively. When a branch of the tree reaches the attractor, the IFS is no longer applied. Once all branches from the root of the tree have reached the attractor, the frontier of the tree is collected. The frontier of the tree is the set of all of the leaves of the tree or all the points were the attractor has been reached. Individual points on the attractor can be labeled based on whether they are reachable from the attractor itself on the left transform (label "a"), on the right transform (label "b"), or on both (label "c"). In this way, strings over a ternery alphabet can obtained at the frontier of the decoded trees. Once the set of strings has been gathered, all duplicates are discarded. This process is illustrated in fig. 4.2 for the tree $(a(ab))$, or string aab.

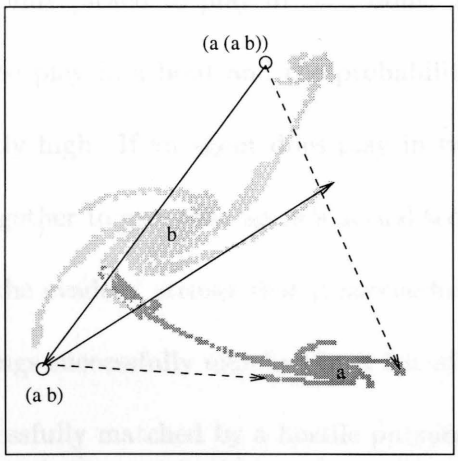

Figure 4.2: The Galaxy attractor, showing derivation of the tree $(a\ (a\ b))$ and its daughter tree $(a\ b)$. Attractor points with addresss a, reachable from the attractor on the left transform, are colored dark gray; points with address b, reachable on the right transform, are light gray. The left transients to the attractor are shown as dashed lines, and the right transients as solid lines. From [3].

Chapter 5

PE Communication Game

Because the execution time to play each agent of each population against each agent of each other population would take too long, a reduced number of bouts between agents was used. 100 bouts out of a possible 1000000 were played. Three agents, one from each population, were pseudo-randomly picked to play in each bout. In this situation, each agent has the same probability to play in a bout and the probability that each agent will play only one bout is significantly high. If an agent does play in two or more bouts, the scores of each bout are added together to give that agent's actual score. In a given bout a pursuer gains a point for each of the evaders' strings that it successfully matches. An evader gains a point for each of its strings successfully matched by a friendly pursuer, and loses a point for each of its strings successfully matched by a hostile pursuer.

The agent's score is used as the fitness function for the genetic algorithm. There is a possibility that an agent's score may be artificially inflated or deflated (depending on the result of adding the two scores together) as a result of playing in two or more bouts. However, because the probability that this will occur is small, the effects will be minimal.

The only effect that an agent's score being artificially altered will have is that that agent will be more or less likely to reproduce in the next generation of the GA.

The GA used is based on a fitness proportional roulette wheel. The higher a specific agent's fitness the more likely it is to be a member of the population for the next generation. Both the mutation operation and cross-over operations were implemented in the experiment. There was .01 probability that mutation would occur on each element of an agent. If mutation did occur, a pseudo-random Gaussian number was added to the element of the agent. There was .7 probability that cross-over would occur between two agents. If cross-over did occur a new (child) agent is created from two (parent) agents in the previous generation. The child agent is created by iterating through each node in the network. The values for the weights that are inputs for this node are pseudo-randomly chosen from one of the two parent agents. This process is repeated for each node in the network.

The GA was run for 1000 generations in order for long term behavior to be established. Each experiment was run with 5 different sets of initial conditions (random agent weights).

5.1 Results

The results show that GAs are capable of evolving RAAM networks to play a PE Game implementing an IFS decoder. In the three-player PE Game, the evader was continually able to evolve to be more predictable to the friendly pursuer then the hostile pursuer.

Fig. 5.1 is a time-series plot of the fitness of each agent for one run of the three-player PE Game. This figure agrees with [2] in that the average fitnesses of all three populations vary noticeably over time, without settling down to a stable value. As the evader gains

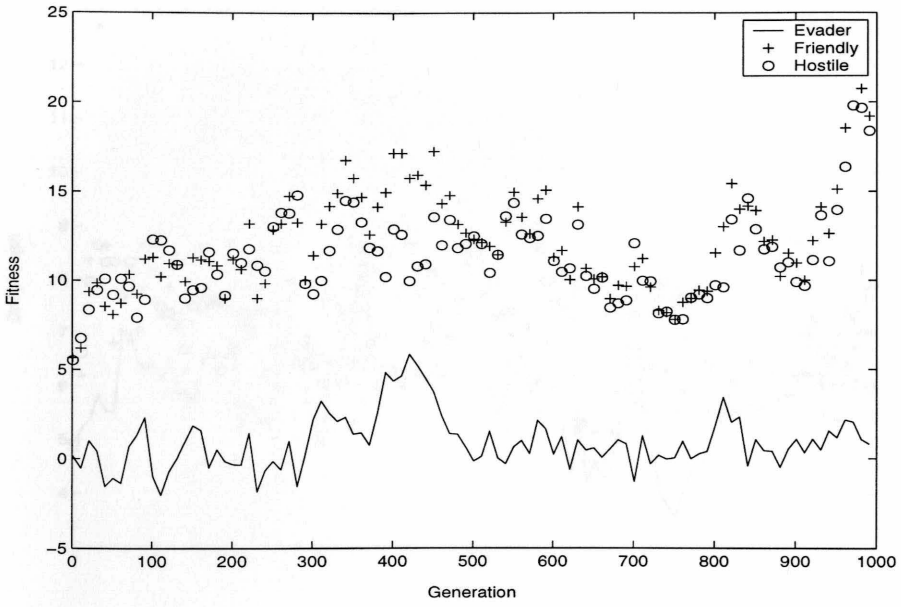

Figure 5.1: Average population fitnesses over generations.

fitness by becoming more predictable to the friendly pursuer, it eventually opens itself up to prediction by the hostile pursuer resulting in prominent peaks and valleys in the fitnesses of all three populations.

Another method of examining the dynamics involved in the PE Game is by looking at the average string length generated by each agent class, as in fig. 5.2.

This graph illustrates how the pursuers are staying ahead of the evader. Both classes of pursuers are generating longer strings than the evader. And the hostile pursuer is generating longer strings then the friendly pursuer. Every string that an agent produces has a substring. The agent must be able to produce every substring in a string to produce the string. Longer strings will have more substrings. Pursuers will then generate longer strings so that more total strings are produced, thus increasing their odds to have a match with the evader.

The attractors for an evader in a three-player PE Game, and the way strings decode

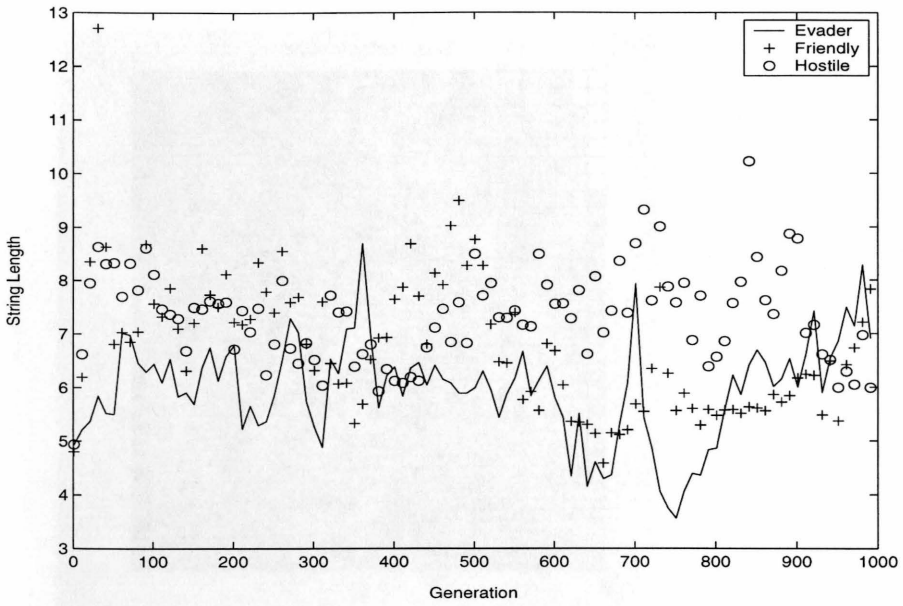

Figure 5.2: Average string lengths over 1000 generations for the PE game with an evader and both a friendly and hostile pursuer.

to these attractors, are shown at the first and last generations in figs. 5.3 and 5.4. These figures show that the complexity of the evaders' "language" generally increased over time.

This interaction can be seen more clearly in the Two-Player PE Game. Fig. 5.5 is a graph of the mean string lengths generated by an evader and a hostile pursuer. Fig. 5.6 is a graph of the mean string lengths generated by an evader and a friendly pursuer.

In fig. 5.5 the evader has been learned to produce short strings. Although these strings have not been examined in detail, we suspect that their unpredictability results from higher complexity. The pursuer produces a large amount of strings in the hope that it will find a match with the evader. At the end of this run, the mean evader fitness was -2.15 and the mean pursuer fitness was 2.15. The evader was attempting to increase its fitness toward zero, while the pursuer was attempting to increase its fitness away form zero. In fig. 5.6

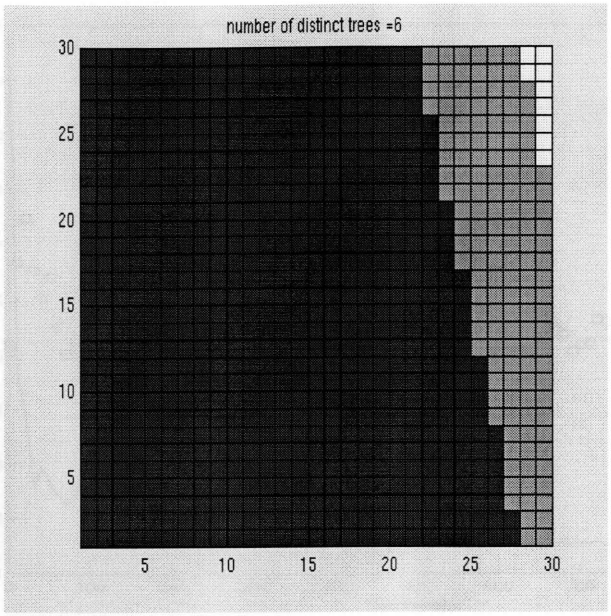

Figure 5.3: Evader Strings at Generation 0. Each pixel represents an (x,y) coordinate in the unit square. Pixels with the same color decode to the same string.

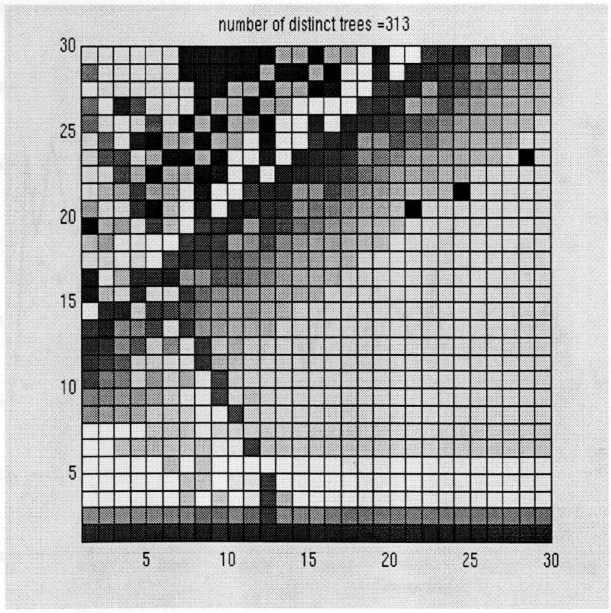

Figure 5.4: Evader Strings at Generation 1000.

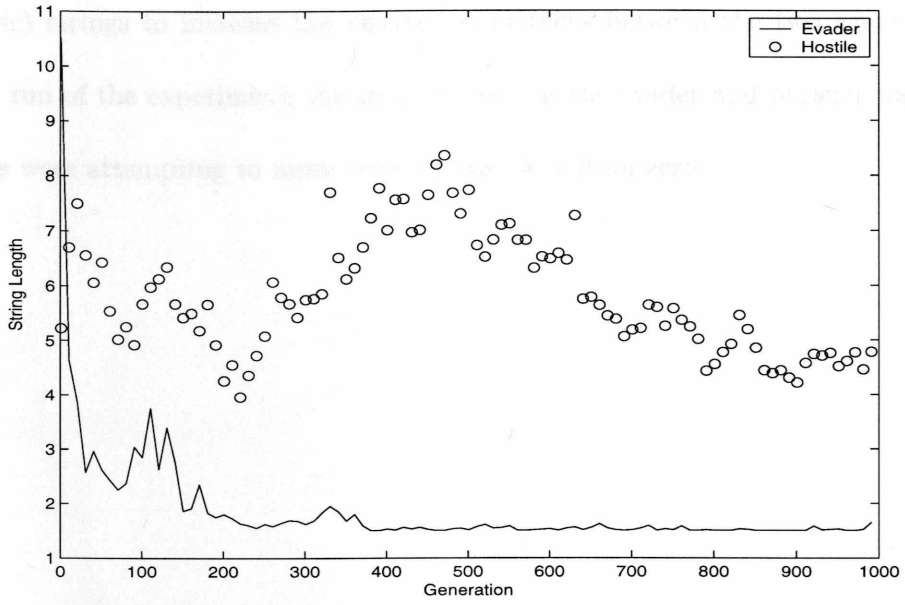

Figure 5.5: Average string lengths over 1000 generations for the evader/hostile pursuer game.

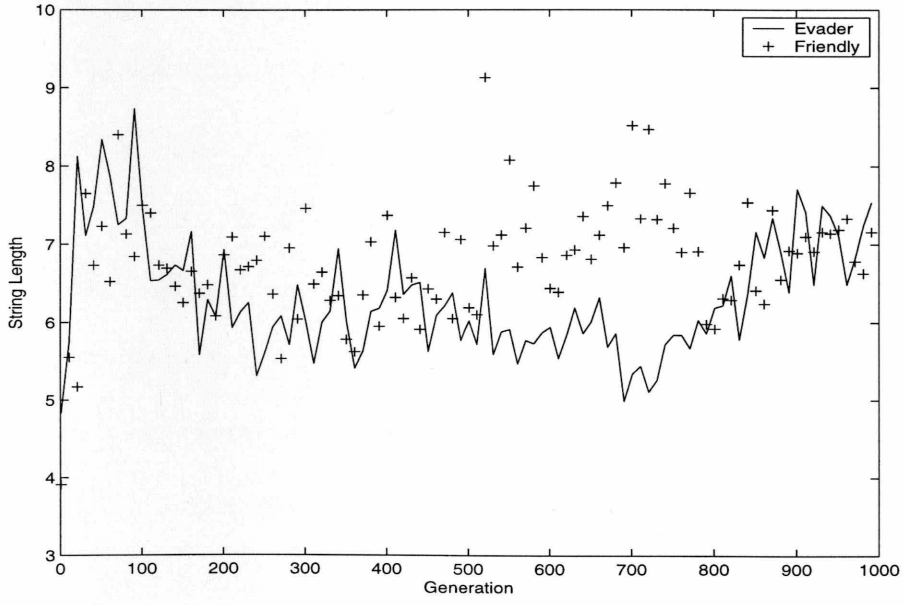

Figure 5.6: Average string lengths over 1000 generations for the evader/friendly pursuer game.

(friendly pursuer) both the agents are producing a moderate number of medium length (presumably simple) strings to increase the number of matches between the two agents. At the end of this run of the experiment, the mean fitness for the evader and pursuer was 23.59. Both agents were attempting to move their fitness away from zero.

Chapter 6

Conclusions and Future Work

GAs are able to evolve connectionist networks that represent variable-sized recursive data structures in fixed-width patterns. Moreover, the results of this thesis show that Genetic Algorithms are an effective method to explore the complexity produced in such networks.

We were able to show this through a series of steps. First, we showed that GAs were a viable method to evolve RAAM networks by comparing the GA with BP on a tree-encoding problem. Then we moved to a problem where there was no known desired output for the network. This prevented us from using BP to train the network and forced us to use a GA to evolve the network. The emergent complexity was a result of the problem, the PE game, and not part of the fitness function used in the GA.

An obvious next step is to examine in detail the complexity of the strings produced by the evader and not just the string lengths. We might speculate that the evaders' strategy may correlate with the grammatical complexity (regular, context-free, context-sensitive) of the strings they produce.

Bibliography

[1] P.J. Angeline, G.M. Saunders, and J.B. Pollack. Complete induction of recurrent neural networks. In *Proceedings of the Third International Conference on Evolutionary Programming*, 1994.

[2] Sevan G. Ficici and Jordan B. Pollack. Coevolving communicative behavior in a linear pursuer-evader game. In *Proceedings of the Fifth International Conference of the Society for Adaptive Behavior*, Kobayashi Pfeifer, Blumberg, editor. MIT Press, 1998.

[3] Simon Levy. *Infinite RAAM: Initial Investigations into a Fractal Basis for Cognition.* PhD thesis, Brandeis University, July 2002.

[4] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 2001.

[5] J.B. Pollack. Recursive distributed representations. *Artificial Intelligence*, 46(1), 1990.

[6] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representation by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, D.E. Rumelhart and J.L. McClelland, editors, volume 1. MIT Press, 1986.