Technical Report CS-2001-22

Evolution of Recurrent Neural Networks to Control Autonomous Life Agents

Tony Abou-Assaleh

Department of Computer Science, University of Waterloo
Waterloo, Ontario, N2L 3G1, Canada
taa@acm.org

ABSTRACT

Studies of artificial life (alife) attempt to simulate simple living beings. On the other hand, autonomous agents researchers are interested in building agents that are able to complete a particular task without supervision. In this research, these two areas of artificial intelligence are combined together into what we call "Autonomous Life Agent" (ALA). ALA is an artificial agent that is sent to some environment to live in without any supervision or any predefined behaviour rules. The primary goal of the agent is to learn how to survive in the artificial environment it lives in. In this research, we utilize a recurrent neural network to determine the agent's actions. A novel ALA Training System was developed that evolves recurrent neural networks using genetic algorithms. The resulting agents are capable of living in multiple similar worlds starting from random initial positions as well as in worlds that are unseen during the training.

ACKNOWLEDGEMENTS

Special thanks and appreciation to Dr. Jianna Zhang, Dr. Nick Cercone, and Dr. Brian Ross for their valuable comments

The financial support for this project was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) under the USRA award, NSERC Grant 228142-2000, and by Communications and Information Technology Ontario (CITO).

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

This ongoing research has many different points of interest. There are three main aspects that are of interest to us: (1) development of the ALA architecture, (2) development of the ALA Training System, and (3) observation of the ALAs behaviour. This project focuses on the second point: building a better genetic engine to evolve better ALAs. “Better” in this context means that the agent should be able to live in multiple similar environments starting from random initial positions as well as in new environment that are unseen during training. Thus, a better genetic engine is one that will be capable of evolving such agents.

1.1. GENERAL DESCRIPTION

ALA has a number of internal variables such as energy and maintenance levels. Each variable is reduced as the agent acts in the environment by a user-controlled rate. When one of the variables becomes negative the agent dies. Therefore, to keep the agent alive, it has to periodically increment the value of the internal variables by visiting specific cells in the world that correspond to each variable. This is analogous to acquiring energy from energy cells and receiving maintenance in home cells.

The artificial environment where the agent has to live is a grid of cells. Each cell can be an empty cell, a wall cell, or a user defined active cell that modifies an internal variable. Typically, there are at least two types of active cells: energy cells and home cells. Empty cells are passive cells. The agent can freely move over empty cells without any reward or penalty. On the other hand, moving into wall cells is disallowed. Any attempt by the agent to do so will cause the agent's death. Staying over an active cell will result in incrementing the corresponding ALAs internal variable by the amount provided by the cell up to a maximum limit of 1.0. After that, there will be a time delay before the agent can re-consume the active cell.

One of the features of the new system is that the agent’s world is surrounded by wall cells. This serves two purposes: (1) the agent has to learn to stay within the boundaries of its world and (2) the boundary cases do not require special treatment in the implementation. Wall cells can also be placed at various locations to act as mines that the agent has to avoid.

This project involves hidden system states. Not all the information about the world is available to the agent as input. For instance, at any time, the agent can sense only the immediately surrounding cells and the current cell. Thus, the agent has to explore the world and encode its information in the agent’s brain. Furthermore, some data is not available to the agent at all such as the time delay of the active cells. The agent must automatically discover this information by observing the visible system states.

1.2. PREVIOUS WORK

The combination of genetic algorithms (GA) with feedforward neural networks has showed many successes during the past decade. However, little research has been done to explore the combination of genetic algorithms with recurrent neural networks (RNN).

In our previous work [1], we developed ALA architecture and a basic training system. The system supported ALA architecture with two internal variables: energy level and maintenance level. The world consisted of only three types of cells: empty cells, energy cells, and home cells. The basic training system evolved only the link weights for a three-layered recurrent neural network with predefined topology.

1.3. RESEARCH OBJECTIVE AND SPECIFICATIONS

The objective of this research is to further explore the power of using recurrent neural networks to control ALAs. In this research, a new ALA Training System is implemented with the following specifications:

- RNN evolution: the training system evolves the link weights as well as the topology of the RNN. The topology evolution includes evolution of the number of nodes in the hidden layer and the number of recurrent nodes (backward links) in the hidden layer.
- RNN efficiency: RNN efficiency is measured by the total number of links in the network. Thus, the user may favour the network with smaller number of links when the performance of two networks is identical.
- Internal variables: the user may define any number of internal variables along with the rate of change per time unit for each variable.
- World definition: the world is surrounded by wall cells. The user may define any number of active cells that correspond to the internal variables. In addition, the user may position additional wall cells anywhere in the world.
- Initial position: the user may select to train the agent from a single starting position or from a number of randomly chosen starting points.
- Training environment: the user may specify one or more world definition to train the agent. A successful agent should be able to live in all the worlds in which it is trained.
- Testing: the evolved ALA may be tested using parameters different from the training parameters. For instance, the agent may be sent to an unseen world to observe its behaviour and ability to survive.
- Genetic engine: the user has full control over the parameters of evolution such as population size, mutation rate, and number of evaluations.

CHAPTER 2: BACKGROUND

2.1. LITERATURE REVIEW

2.1.1. General Review

The first formal survey of the methods used when combining GAs with NNs was done by Schaffer et al in 1992 [11]. This survey discussed two major areas: using GAs to support NNs and using NNs to support GAs. This paper extends the survey by focusing specifically on using GAs to evolve RNNs.

Many algorithms such as backpropagation through time (BPTT) and real-time recurrent learning (RTRL) are used to train RNNs [8]. These algorithms are proven to be effective and have strong mathematical foundations. However, their disadvantage is the complexity of the computation and the amount of processing power they require. Furthermore, they are difficult to apply to problems where the gradient function is not available. In other words, when the correct output that the network is expected to produce is unknown. One of such problems is reinforcement learning. This leads to a search for an algorithm that, while may not necessarily find the optimal solution, is capable of finding a good solution in reasonable time. Thus, GAs are excellent candidates of such algorithms since they are useful in searching large and complex search spaces and are relatively fast compared with the traditional search techniques.

There are several ways in which GAs can be used to evolve RNNs, which can be classified in three major categories: (1) evolving RNN parameters, (2) evolving RNN weights, and (3) evolving RNN topology. Each of these categories is presented in this paper. In addition, different strategies for combining these techniques are also discussed.

2.1.2. Evolving RNN Parameters

GAs can be used to fine tune many of the parameters required for training a neural network. Initial weight values [2] and the learning rate [6] are some of the parameters that have been prepared by GA in feedforward networks. This technique is rarely used. Although it can be extended to RNNs as well, the literature lacks research in this area. The reason behind this shortage is that researchers who are interested in employing GAs to train RNNs find that it is more practical to use GAs as a training algorithm rather than find optimal values for conventional training algorithms.

2.1.3. Evolving RNN Weights

In this method, GAs are used as a training algorithm to determine the link weights. This method is the most popular in the combination of GAs and RNNs. Wieland [12] did some of the early research on this topic where he evolved the weights of a fully recurrent network. The success of this technique is due to the simplicity of its implementation, its relative speed, and its effectiveness [1]. The use of real numbers to encode the weights instead of binary encoding has increased because it greatly reduces the amount of memory and computation time required, and is proven effective [1], [11].

2.1.4. Evolving RNN Topology

One of the main questions that puzzle any person constructing an RNN is its topology. There is no common algorithm to determine the number of node, which nodes should be connected, or which nodes should have recurrent links for a particular problem. Traditionally, the researcher would experiment with a number of possibilities and select one that is expected to work best. While this approach may be acceptable for small problems, finding good topology for larger problems is a very complex task. For this reason, GAs are found to be the best approach to determine the network topology.

It turns out that using GAs to determine the topology of a network is a non-trivial task. Its difficulty arises from the fact that GAs need appropriate data representation and genetic operators that will allow good features to be preserved during mating [6]. Both direct representations, where a connection matrix is used, and indirect representations, where there is a mapping between the network topology and the chromosome encoding, are used. Since it is almost impossible to understand the meaning of the weights and the interaction of the nodes in complex problems, the task of selecting a good representation is further complicated. As a result, GAs are often used to optimize some parameters that define the topology. For instance, GAs can be used to evolve the number of nodes in the hidden layer. On the other hand, if the number of nodes is fixed, GAs can be used to determine which nodes should be connected. A combination of these two techniques has been implemented in a system called ENZO [10]. An indirect representation that uses graph context free grammars has been successfully implemented [7]. To the best of the author's knowledge, neither of these approaches has been implemented on RNNs because of the difficulty to encode recurrent links.

2.1.5. Evolving Complete RNN

GAs can be used to evolve the network topology and the link weights at the same time. There are several possible implementations for this approach. Two separate GA engines can be used where one of them evolves the topology of the network and the other is used as a training algorithm to evolve the weights. Recently, Polani and Miikkulainen [9] proposed a new approach where the hidden-layer neurons with their incoming and outgoing links are evolved in one population, and the combinations of these neurons to form feedforward networks are evolved in another population. The benefit of this implementation is that neurons and their links are treated as building units and are preserved during crossover. This method is yet to be extended to RNNs.

2.2. TECHNICAL REVIEW

2.2.1. Neural Networks

Artificial neural networks consist of a number of artificial neurons that simulate the characteristics of biological neurons in terms of the relationship between the input and the output [3]. As Figure 1 illustrates, every neuron has a set of input connections p_i, a bias b, and an output *a*. Every input link has an input value *p* and a weight *w*. The total input, *n*, is calculated by multiplying every input value by the corresponding link weight and summing the products. The

output of the neuron can be viewed as function: $a = f(\Sigma wp+b)$. Various activation functions can be used to determine when and how the neuron fires (produces output).

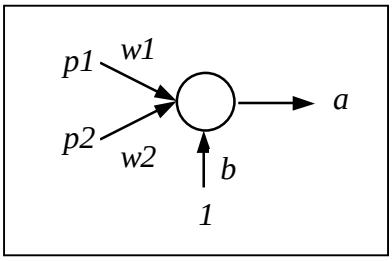

Figure 1: A Single Artificial Neuron

Neurons are grouped into layers with links from one layer to another, which forms a neural network (see Figure 2). If the links are always in one direction then the network is called a "feedforward neural network." On the other hand, if the network has backward connections then it is called a "recurrent neural network."

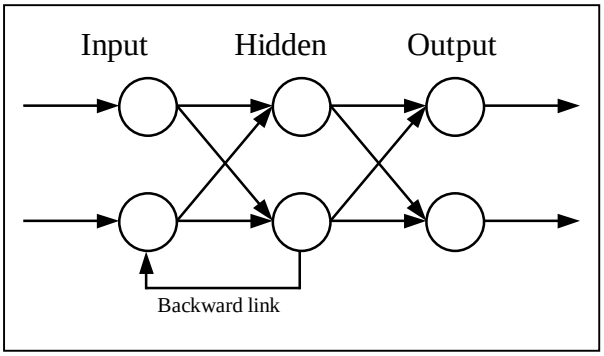

Figure 2: Three-Layer Recurrent Neural Network

2.2.2. Genetic Algorithms

Genetic algorithms are analogous to natural evolution [5]. They evolve populations consisting of individuals. Every individual is defined by his genes. After creating an initial population, individuals with higher fitness are more likely to survive and produce offspring. When an individual is probabilistically selected based on his fitness, one of the following three operations is used: (1) crossover, (2) mutation, or (3) replication. Crossover requires two parents. A part of the genetic data is exchanged between the two parents to produce two children. The advantage of this is to combine good features from both parents to produce a better individual. Mutation changes some of the genes to random values, which allows introducing new genetic information not currently present in the population. Some individuals may be replicated (copied) to preserve good combinations of genetic data.

CHAPTER 3: APPROACH

The ALA Training System consists of five components: (1) agent, (2) genome, (3) population, (4) genetic engine, and (5) world. The agent component implements an ALA using an RNN. The topology of the RNN and the weights of the links are defined by the genome component. A collection of genomes is managed by the population component. The genetic engine component uses genomes from the population to conduct evolution by applying genetic operators such as crossover and mutation to the genomes. Figure 3 represents the data flow between the different components of the system. Each of these components is discussed in details in the following subsections.

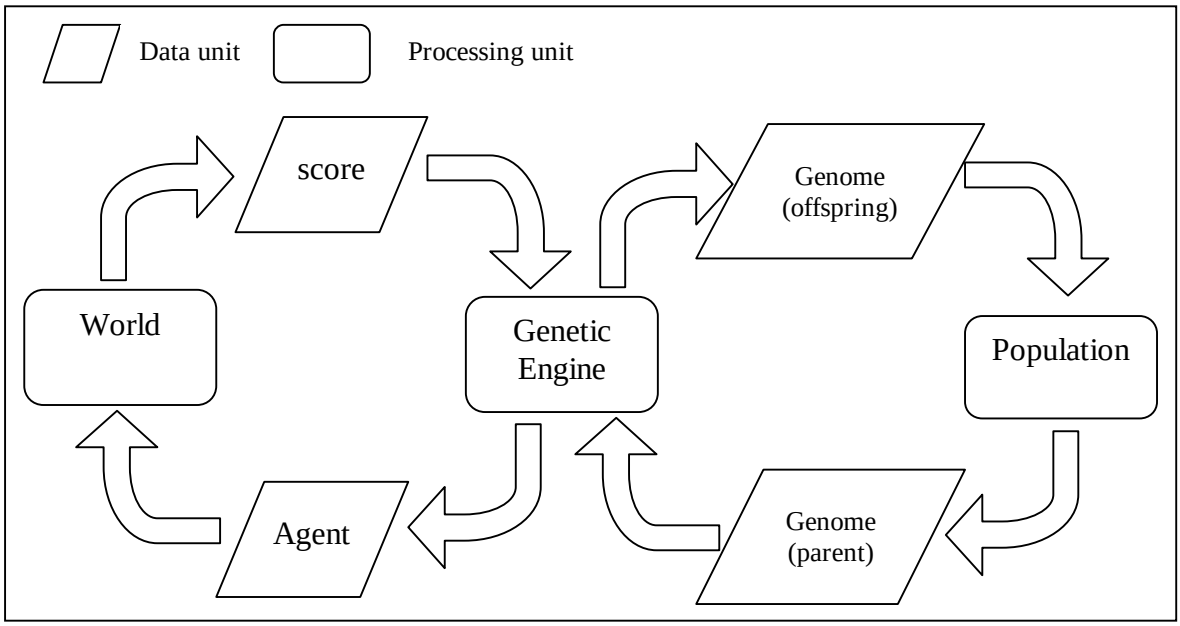

Figure 3: Data Flow Chart between Components of ALA Training System

It was originally proposed that the system would be implemented in Java. However, after analyzing the complexity of the system, we found that although Java would be a very convenient language from the programming point of view, the execution time during the testing phase might become impractical. As a result, C is the language that is used in implementing the training system due to its efficiency in intensive calculations.

3.1. AGENT STRUCTURE

The agent has a number of internal variables that must remain positive (greater than zero) to keep the agent alive. When an input vector is passed to the agent, the agent processes it and decides which action is to be performed next.

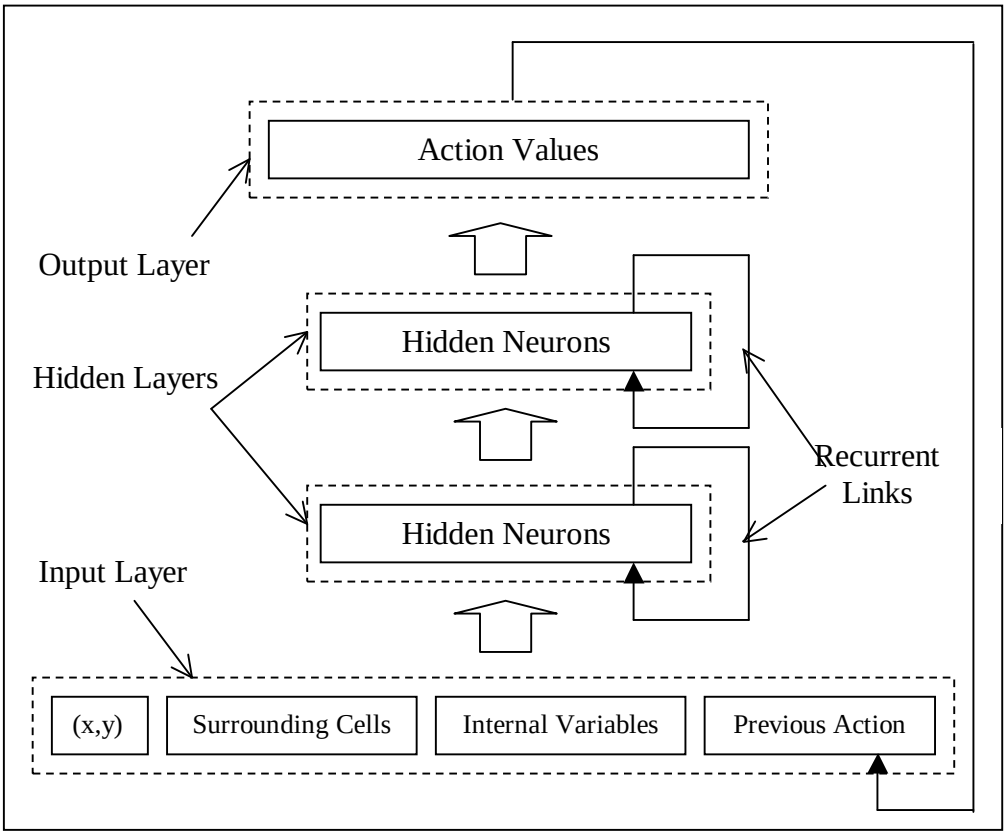

Figure 4: Agent Structure

The internal structure of the agent is a multi-layer RNN (see Figure 4). Every layer is fully connected with the next layer. Some of the nodes in the hidden layers have recurrent links. The input layer accepts the following information as input:

- x and y coordinates of the current cell
- the type of the eight surrounding cells
- the values of the internal variables
- the previous action

The input is then passed through a number of hidden layers with recurrent links. Finally, the output layer contains nine outputs that represent a movement to one of the eight surrounding cells or staying at the current cell. For example, if at time *t* an agent has two internal variables of value: (1) energy = 0.1 and (2) maintenance = 0.9, who is located in an energy cell (5, 5) of a

10x10 world that is surrounded by empty cells, and assuming that the previous action was 9, the input would like as follow:

[0.5 0.5] [0.0 0.0 0.0 0.0 0.5 0.0 0.0 0.0 0.0] [0.1 0.9] [1.0]

Where the first set of values correspond to the coordinates of the agent divided by the world dimensions. The second set of nine values corresponds to the types of the surrounding cells. A value of 0.0 indicates that the corresponding cell is an empty cell and a value of 0.5 corresponds to an energy cell (variable index divided by total number of variables). The third set of values consists of the current values of the internal variables. The last value is the previous action of the agent divided by 9, the number of possible actions. The output vector of the RNN would have nine values where the greatest value corresponds to the action to be performed. In this example, the action should be 5, which corresponds to remaining in the current cell and consuming the energy it provides. Thus, the output vector might contain the following values:

[-0.12 0.25 0.04 -0.78 0.75 -0.44 0.32 0.19 0.11]

3.2. GENOME STRUCTURE

The genome completely defines an agent. It consists of two parts: (1) topology definition and (2) link weights. The topology definition consists of:

- *H*: the number of hidden layers
- *N1*: the number of nodes in the first hidden layer
- *R1*: the number of recurrent links in the first hidden layer

:

- N_H: the number of nodes in the *H*th hidden layer
- R_H: the number of recurrent links in the first hidden layer.

The link weights are sets of two-dimensional arrays where each array *Ai* represents the weights of links from layer *i* to layer *i+1*. In addition, the initial outputs of the recurrent neurons are also represented in the genome.

3.3. POPULATION

The role of the population is to manage genomes and make them available for the genetic engine for reproduction. The selection of individuals is done using the tournament selection with a user defined size *tsize*. First, *tsize* random genomes are selected from the population. Then, the genome with the highest fitness (score) is sent to the genetic engine. If there is a tie in the score then the size of the genome is used to favour the genome with the smaller size. The genetic engine adds the newly created children using the reproduction operators to a new population.

3.4. WORLD STRUCTURE

This component defines the environment in which the agent has to live. Its main function is to evaluate agents by measuring how long they are able to survive in the given environment. The internal representation of the environment is a 2-dimentional grid where each cell consists of the following:

- x and y coordinates
- cell type (e.g. home, energy, empty)
- increment: the amount of change that will affect the corresponding internal variable when the agent selects to 'stay' in this cell
- delay: the number of time intervals that must pass after a discharge in order for this cell to become active again

Examples of world configurations are provided in Tables 2, 4, and 5.

3.5. GENETIC ENGINE

The genetic engine conducts the evolution process to evolve agents. It uses populations as collections of genomes and uses the world component to determine the fitness of individuals. The evolution terminates when the following termination criteria are met:

- The agent can live up to the maximum age whenever it is evaluated.
- The size of the genome is less than or equal to the desired genome size as specified by the user.

The training algorithm for the system is outlined below:

1. Initialize a random population
   1.1. Randomly generate a topology
   1.2. Generate random weights
2. Evaluate population
   2.1. Create an agent using a genome
   2.2. Use the world component to evaluate the agent
   2.3. Calculate the fitness based on the age of the agent and its size
   2.4. If termination criteria are met then stop
3. Replicate best *E* (elite) genomes to the new population
4. Use tournament selection to select individuals
5. Select a reproduction operator probabilistically (based on the genome crossover/mutation probability)
   5.1. Mutation: select probabilistically the type of mutation (based on weight/topology mutation probability)
      5.1.1. Topology mutation:
         5.1.1.1. Randomly change the number of nodes in each hidden layer
         5.1.1.2. Randomly change the number of recurrent links in each hidden layer
         5.1.1.3. Recreate the genome with new topology using previous weights when possible and random weights when new links are introduced
      5.1.2. Weight mutation: probabilistically select weights and assign them a random value
      5.1.3. Crossover: use standard n-point crossover
6. Go to step 2.

The modified topology mutation operator that we developed in this project changes the number of the neurons in the hidden layers as well as the number of the recurrent links in each hidden

layer to a random number. Whether a particular topology defining value is mutated or not depends on the layer mutation probability variable that is controlled by the user.

Once a new topology is defined, the link weights and the initial outputs of the recurrent nodes of each hidden layer are determined. The values of these genes are copied from the original genome. Whenever the new topology definition requires additional value, these values are filled with random numbers between -1.0 and +1.0. Thus, in the topology mutation operation, effort is made to preserve the original weight values.

One of the advantages of this approach is that it eliminates the need for topology crossover. If a separate topology crossover operation were to be implemented, its function would be to create a new genome that uses the same link weight values as one of its parents but has the topology of its other parent. However, since the evolved networks have less than five nodes in the hidden layer, the search space of network topologies is fairly small. As a result, producing the topology of a particular genome using the topology mutation operator has a good probability of happening.

CHAPTER 4: EXPERIMENTAL RESULTS

Originally, one of the objectives of this research was to compare the relative efficiency of a one-hidden-layer RNN and a two-hidden-layers RNN. This objective was based on the results obtained from our previous research [1] where more than 10 nodes where used in the hidden layer of a single-hidden-layer RNN. However, during the initial experiments with the new training system it was found that even somewhat complex world definitions with multiple internal variables could be controlled with less than five neurons in the hidden layer. Experiments with world definitions of a high complexity were not performed but are on our future research list. This research did not aim to correlate the size and complexity of the environment with the required minimum number of nodes in the hidden layer. Since genetic algorithms produce a good solution and not necessarily the optimal solution, the experiments were more directed towards evolving an agent who can primarily survive the environment with preference to solutions with a smaller number of weights, rather than determining the absolute minimum number of nodes that is required in the hidden layer.

The difficulty level of an environment is characterized by the following factors:

1. The minimum number of distinct active cells the agent has to visit. The agent normally learns how to reach one active cell at a time. The more active cells the agent has to visit the longer it will take to train the agent.
2. Distance between the active cells. As the distance increases, it becomes more difficult for the agent to find another cell once it found one. Moreover, longer distances require the agent to remember more information about the world.
3. The change of rate of the internal variables. With small rate of change, the agent can afford to waste some time while wondering around looking for active cells. Larger change of rate means that the agent has to find a near optimal path between the cells. Any wasted move could mean the death of the agent.
4. The presence of mines can further complicate the environment for the agent depending on their position.

There are four testing phases as outlined below:

1. Phase 1W S (one world, static initial position): the training is conducted on a single world definition with the agent starting its life span from a static initial position.
2. Phase 1W R (one world, random initial position): the training is conducted on a single world definition with the agent starting its life span from a random initial position each time it is evaluated.
3. Phase *n*W R (numerous worlds, random initial position): the training is conducted on *n* world definitions with the agent starting its life span from a random initial position each time when it is evaluated.
4. Phase *n*W R X (numerous worlds, random initial position, unseen testing world): the training is conducted on *n* world definitions with the agent starting its life span always from a random initial position each time when it is evaluated. After that the agent is tested on a new world definitions that were unseen during the training

The parameters listed in Table 1 were kept constant for all runs. Each run was initialized with a unique random number seed. The parameters that were customized for each phase are described in the following subsections along with the experimental results.

Table 1: GA evolution parameters that were constant for all runs

Parameter	**Value**
Tournament size	4
Number of elite genomes	3
Genome crossover probability	0.7
Genome mutation probability	0.3
Crossover probability of a weight	0.25
Weight mutation probability	0.9
Topology mutation probability	0.1
Mutation probability of a weight	0.05
Mutation probability of a layer topology	0.7
Number of simultaneous runs	10

4.1. PHASE 1W S

The objective of this phase was to test the functionality of the new ALA Training System and evolve agents similar to the agents evolved in our previous research [1] but with more complex world definition. The complexity of the world used in this case emerges from a moderately large number of internal variables. The agent is required to learn how to satisfy four different internal variables. Moreover, although the world size is still small (10x10), it is larger than the world that was use in our previous research (4x4). The parameters used in the training process are listed in Table 2.

The age of the best individual and the average age of the population were averaged over ten runs. The results are displayed in Figure 5. The jumps in the fitness curve shown in Figure 5 indicate that one of the runs had reached the maximum age at that generation. The age of the best individual and the population average of the run that produced the best result are shown in Figure 6.

Table 2: GA Evolution Parameters Used in Phase 1W S

Parameter	**Value**
Population size	5000
Number of generations	2000
Maximum age	501
Internal variables (variable, rate of change)	(E, -0.02), (H, -0.018), (M, -0.016), (B, -0.014)
World definition # = wall cell . = empty cell H = Home E = Energy M = Minerals B = Base	########## #........# #.H....B.# #........# #........# #........# #........# #.E....M.# #........# ##########
Active cells (x y variable delay increment)	2 2 H 10 1.0 7 2 B 10 1.0 2 7 E 15 1.0 7 7 M 7 1.0
Initial position	5, 5
Desired genome size (number of links)	50

The best-evolved RNN was actually a feedforward network. It had three nodes in the hidden layer and did not have any recurrent nodes. The total number of weights was 87. This RNN was found in generation 736. The input-to-hidden weight matrix and the hidden-to-output weight matrix of this RNN are listed below (truncated to 3 places after the decimal point):

Input-to-hidden weight matrix:

```
-0.993 -0.779  0.383
-0.701 -0.726  0.063
-0.897 -0.853  0.867
 0.700  0.925 -0.113
-0.865 -0.998 -0.528
 0.820 -0.478 -0.816
-0.791  0.150  0.518
 0.261  0.004 -0.429
-0.138  0.469  0.007
-0.730  0.211  0.050
-0.346  0.765 -0.010
 0.405 -0.275  0.406
 0.898 -0.145 -0.944
 0.715 -0.983  0.099
-0.972 -0.682  0.729
-0.805 -0.198 -0.650
-0.944 -0.686 -0.446
```

Hidden-to-output weight matrix:

```
-0.507  0.583 -0.970  0.942 -0.158 -0.703 -0.702  0.661  0.140
-0.491 -0.451 -0.863 -0.762 -0.823 -0.013  0.592 -0.907 -0.593
-0.871 -0.993 -0.260 -0.727 -0.969 -0.756  0.185  0.920  0.234
 0.451  0.041 -0.394  0.588 -0.527 -0.222  0.193  0.385 -0.748
```

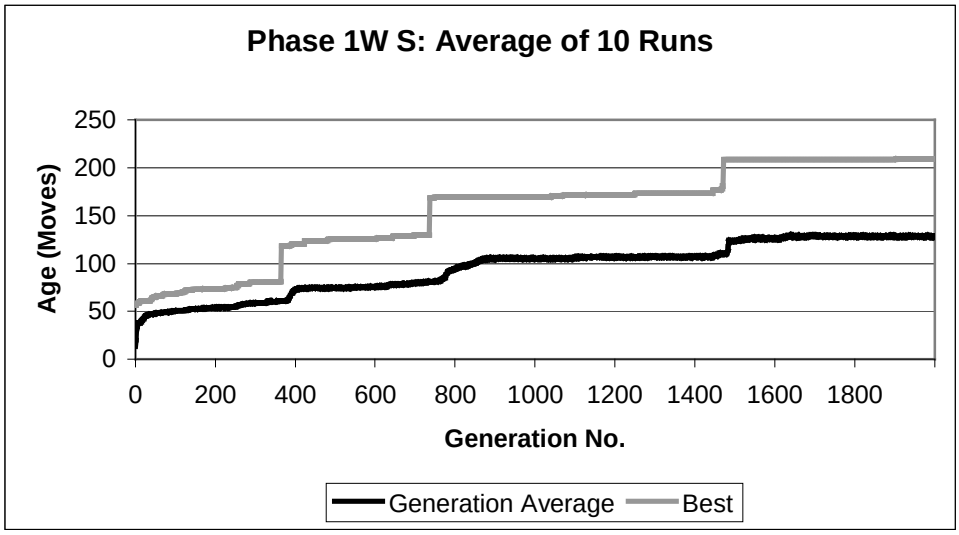

Figure 5: Evolution Progress of Phase 1W S Averaged Over 10 Runs

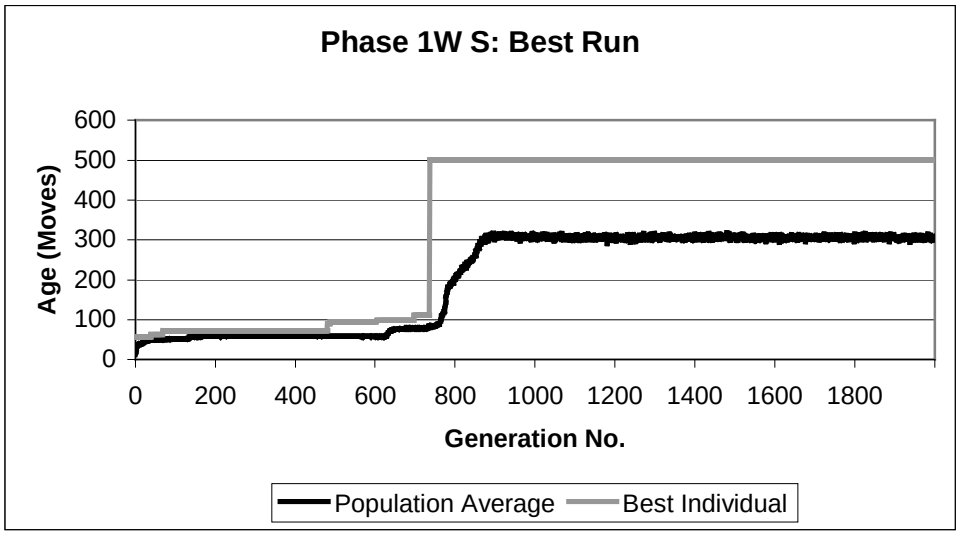

Figure 6: Evolution Progress of the Best Run in Phase 1W S

One can clearly observe two main stages in the shape of the evolution progress graph. The first stage is characterized by very low fitness score that does not improve over time. The length of this period depends primarily on the random number seed, as well as on the other evolution parameters. The second stage shows significant improvement in the fitness. The agent rapidly learns how to live longer until it finally reaches the maximum age within a small number of generations. It is suspected that this phenomenon is a result of the environment design. The relatively small world size does not facilitate slow incremental learning. On the contrary, it divides the age scale into a few discrete steps where each step corresponds to consuming an

active cell. In order to verify this hypothesis, further experiments need to be performed that use relatively large worlds with numerous active cells.

One major weakness of the agents evolved in this phase is that they are unable to live in the same world they were trained in if the starting position changes. This indicates that the agents are trying to encode a path in their RNN that may or may not be dependent on the inputs. The next testing phase overcomes this limitation.

4.2. PHASE 1W R

The objective of this phase was to evolve robust agents with global view of the world. The agents are required to encode the world in their RNN to be able to go to the desired active cell from any position in the world.

The fitness function consists of averaging the age of the agent over a number of test cases. For each test case, the initial position of the agent is chosen randomly. Because the fitness function does not favour partial solutions where the agent is successful to raise the value of only one internal variable, it is important to make the change rate of the internal variables sufficiently different. Doing so enables agents who found only one active cell to live longer than those who did not find any. Consequently, these fitter agents are able survive to the next generation and continue searching for the other active cells.

A world size of 7x7 was chosen for the runs. Every genome was evaluated starting from 15 randomly selected initial positions. The number of internal variables was reduced to three to shorten the training time. Table 3 lists various parameters that were used in the training.

Table 3: GA Evolution Parameters Used in Phase 1W R

Parameter	**Value**
Population size	2000
Number of generations	2000
Maximum age	512
Internal variables (variable, rate of change)	(E, -0.02), (H, -0.018), (M, -0.016)

World definition
`# = wall cell`
`. = empty cell`
`H = Home`
`E = Energy`
`M = Minerals`
	`#######`
	`#.....#`
	`#.H...#`
	`#....E#`
	`#.....#`
	`#.M...#`
	`#######`

Active cells (x y variable delay increment)	`2 2 H 10 1.0`
	`5 3 E 15 1.0`
	`2 5 M 7  1.0`

Number of evaluations	15
Initial position	Random
Desired genome size (number of links)	200

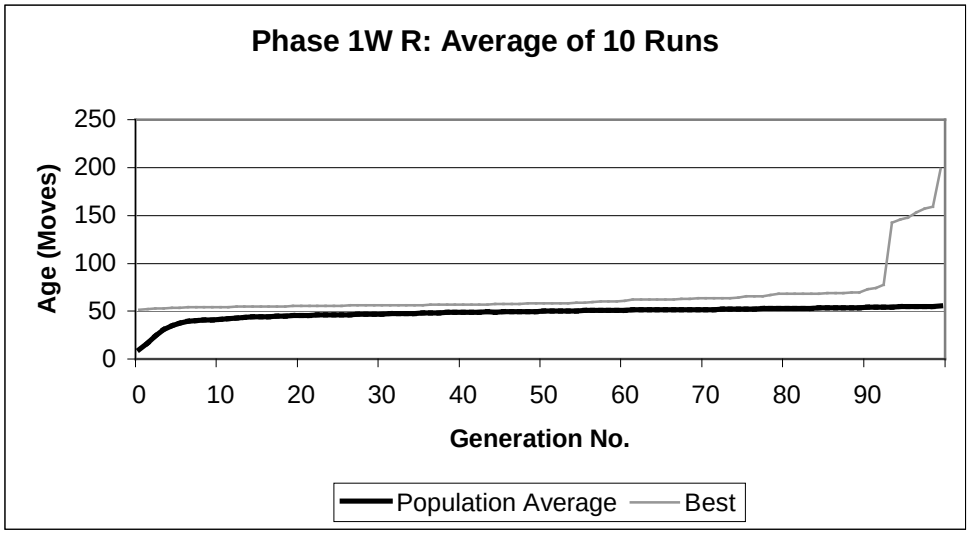

Figure 7: Evolution Progress of Phase 1W R Averaged Over 10 Runs

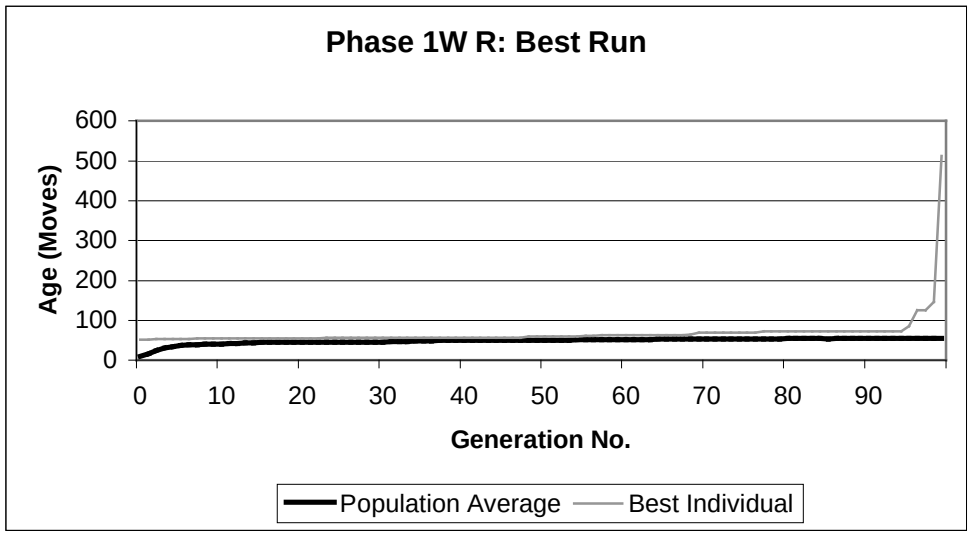

Figure 8: Evolution Progress of Best Run in Phase 1W R

Out of ten runs, nine found an agent that can live up to the maximum age of 512. The number of generations that it took to find the solution varied between 99 generations (best run) and 454 generations (longest run). Since generally not all of the runs will be able to find a solution in the desired number of generations, and since only one solution is desired for a particular world configuration, it is more feasible to terminate all the runs once a solution is found by one of them. This strategy enables running more simultaneous runs for a shorter period of time, and consequently, achieving finding a solution in a reasonable time without wasting a lot of computation power.

The evolution progress averaged over the ten runs is shown in Figure 7. The number of generations was truncated to 99 generations, which is where the termination criteria were met. Figure 8 displays the evolution progress of the best run. The solution it evolved had a size of 84, which corresponds to three nodes in hidden layer with no recurrent links. Again, the resulting network was a feedforward neural network.

4.3. PHASE *n*W R

The objective of this phase was to force the agent to employ judgement on current local input as well as generalized global knowledge of the active cells in the world.

Table 4: GA Evolution Parameters Used in Phase *n*W R

Parameter	**Value**
Population size	2000
Number of generations	2000
Maximum age	505
Internal variables (variable, rate of change)	(E, -0.02), (H, -0.018)
Number of world definitions	2
World 1 definition # = wall cell . = empty cell H = Home E = Energy	########## #........# #.H......# #........# #........# #........# #........# #........# #.......E# ##########
World 1 Active cells (x y variable delay increment)	2 2 H 10 1.0 8 8 E 15 1.0
World 2 definition # = wall cell . = empty cell H = Home E = Energy	########## #........# #........# #.H......# #........# #........# #........# #........# #.......E# ##########
World 2 Active cells (x y variable delay increment)	2 3 H 10 1.0 8 8 E 15 1.0
Number of evaluations	30
Initial position	Random
Desired genome size (number of links)	100

Two distinct but similar world definitions were used to train the ALA. The two worlds had a size of 10x10 with two active cells. The only difference between the two worlds was that one of the two active cells was moved down by one cell. This is actually a much bigger difference than what it appears to be for a number of reasons. Firstly, the agent is not required to ram over all of the 100 cells. In fact, the world is surrounded by wall cells, which reduces the number of safe

cells to 7x7 = 49. Considering that the agent needs to know only a few cells along the path between the two active cells, the total number of cells visited is estimated to be around 20 cells. As a result, this gives 10% change in the world definition as might be seen by the agent. Secondly, the world has only two active cells, which means that 50% of active cell definitions is different between the two worlds. Combining these two factors gives a rough estimate of the difference to be about 8%. This factor is indirectly reduced by specifying low rates of change for the internal variables, which allows the agent to explore the world more freely before being required to visit an active cell. Table 4 lists the parameters that were used to train ALA in this phase.

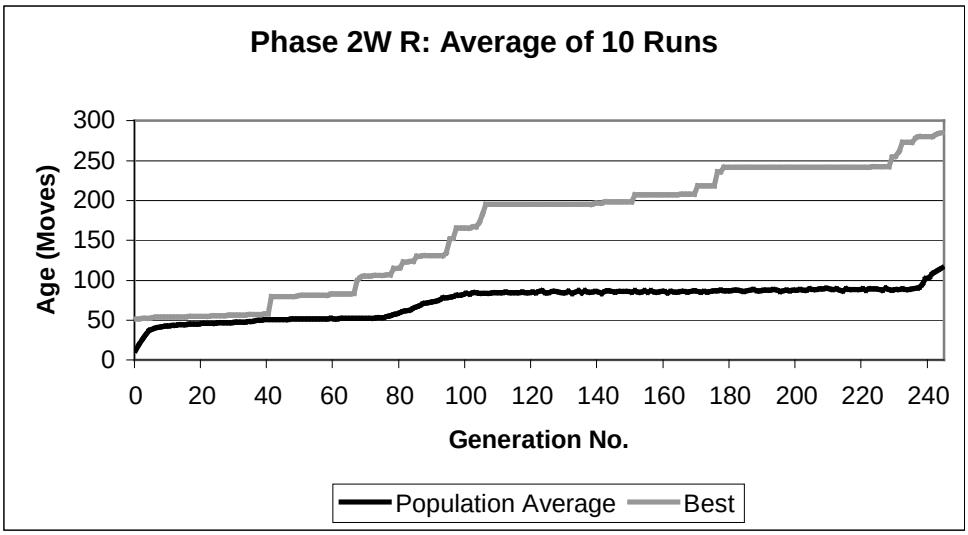

Figure 9: Evolution Progress of Phase 2W R Averaged Over 10 Runs

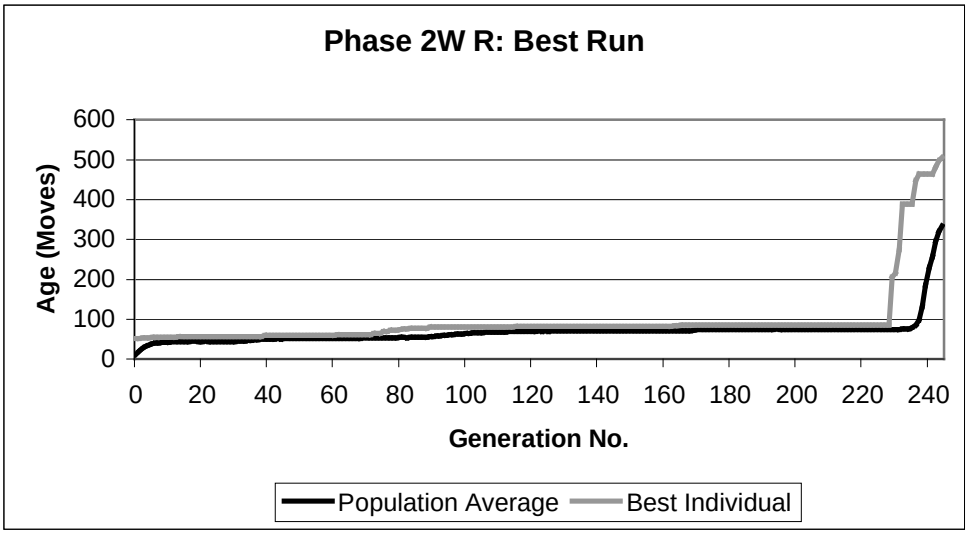

Figure 10: Evolution Progress of Best Run in Phase 2W R

Out of ten runs, eight found an agent that can live up to the maximum age. However, two of the eight successful runs evolved genomes with size of 105, which is greater than the desired size of at most 100. The best solution was found in generation 244 with size of 57, which corresponds to two neurons in the hidden layer and no recurrent links.

The evolution progress averaged over the ten runs is shown in Figure 9. The number of generations was truncated to 244 generations, which is where the smallest ALA was found. The evolution progress of the run that evolved the smallest ALA is show in Figure 10.

4.4. PHASE *n*W R X

The objective of this phase was to test whether the advantageous properties of using neural networks were preserved in the ALA architecture. In particular, this phase was directed towards observing the agent's generalization and fault tolerance abilities. These are rather ambitious intentions. The results of this testing phase, both positive and negative, are the basis for directing future research.

The six RNNs that met the termination criteria in the previous phase (*n*W R) where tested on a new world definition that was not presented during the training. The testing world differs from the two training worlds only by the position of one of the active cells, which is now located one cell away from its position in the training worlds. As discussed in the previous section, this difference is significant. Table 5 lists the parameters that were used for testing.

Only one ALA from the six candidates ALAs was able to live in the testing world. Its RNN had three nodes in the hidden layer with one of them having a recurrent link. The performance of this agent in the testing world is shown in Table 6.

Table 5: ALA Testing Parameters Used in Phase *n*W R X

Parameter	**Value**
Maximum age	501
Internal variables (variable, rate of change)	(E, -0.02), (H, -0.018)
World 1 definition # = wall cell . = empty cell	########## #........# #..H.....# #........# #........# #........# #........# #........# #.......E# ##########
Active cells (x y var. delay incr.)	3 2 H 10 1.0 8 8 E 15 1.0
Number of evaluations	20
Initial position	Random

The initial output of the recurrent link was -0.085. The input-to-hidden weight matrix and the hidden-to-output weight matrix are of this RNN are listed below (truncated to 3 places after the decimal point):

Input-to-hidden weight matrix:

```
 0.867 -0.648  0.371
 0.629  0.649 -0.166
 0.939  0.674 -0.312
 0.690 -0.535 -0.801
-0.697 -0.157 -0.782
 0.146  0.519 -0.703
-0.313  0.116  0.298
-0.976  0.766  0.709
 0.003 -0.642 -0.177
 0.144  0.686 -0.608
 0.564 -0.732  0.766
-0.639 -0.697  0.845
 0.186  0.950 -0.622
 0.993  0.894 -0.607
 0.451  0.958  0.938
 0.663  0.534 -0.015
```

Hidden-to-output weight matrix:

```
-0.145 -0.850  0.933 -0.059 -0.091 0.617  0.371  0.341  0.362
 0.719 -0.065  0.619  0.493 -0.415 0.681  0.293  0.790 -0.775
 0.556 -0.332 -0.198 -0.766 -0.774 0.629 -0.694  0.650 -0.169
 0.709 -0.727  0.986  0.801 -0.403 0.738 -0.866 -0.798 -0.945
```

Table 6: ALA Performance in Testing World of Phase *n*W R X

Case No.	Initial Position	End Position	Age
1	5, 8	4, 4	505
2	1, 4	3, 2	505
3	6, 8	2, 2	505
4	1, 2	3, 2	505
5	8, 7	1, 8	49
6	5, 7	3, 3	505
7	7, 5	3, 2	505
8	5, 1	1, 8	49
9	8, 5	3, 2	505
10	4, 5	3, 2	505
11	5, 3	5, 5	505
12	1, 3	3, 2	505
13	5, 6	5, 5	505
14	3, 3	3, 2	505
15	4, 1	3, 2	505
16	6, 8	2, 2	505
17	3, 4	3, 2	505
18	6, 5	4, 4	505
19	5, 4	1, 8	49
20	2, 6	2, 3	505

The agent succeeded to live seventeen times and failed three times. The cause of agent's death was attempting to move into a wall cell, which is prohibited. A quick look at Table 6 shows that the agent failed when it moved to cell (1, 8). Since this cell is the further away from the path between the two active cells, it is very unlikely that the agent explored this cell during the training. Moreover, this was a corner cell. The agent had only three possibly useful actions out of the nine actions. Thus, the probability of selecting a good action in this situation was only 0.33. This value is significantly low and one could not depend on. To eliminate this phenomenon the agent must be trained in world environment with wall cells positioned randomly. The agent in this case would learn how to avoid wall cells in general, and not only the ones that are on the edge. Consequently, the agent would be able to select an appropriate action even in unfamiliar situations.

CHAPTER 5: CONCLUSION

This work presents a unique approach to evolving RNNs where a direct numeric representation is used to encode the definition of the hidden layer. The successful results of the four phases of experimenting lead to a number of significant discoveries, which are discussed in the following sections.

5.1. WORLD COMPLEXITY

The scalability of the new ALA Training System is confirmed when the agents were successfully trained on more complex world definitions than the ones used in our previous work. In this work, worlds of size up to 10x10 were used with agents with up to four internal variables. The efficiency of the new system makes it possible to conduct such experiments even with limited computing resources.

5.2. ALA FLEXIBILITY

The evolved agents have great robustness, flexibility, and generalization abilities. ALAs are able to survive in environment that is unseen during the training. The initial position of the agent is also irrelevant. The agent is capable of memorizing the approximate positions of the active cells during the training process and then applying this knowledge to new environment.

5.3. TOPOLOGY EVOLUTION

The evolution of the topology of the RNN has lead to discovering a number of interesting results. Firstly, the required number of nodes in the hidden layer is much smaller than what is used traditionally. The experimental results show that it is possible to evolve agents with as few as one node in the hidden layer and still have good agent performance in relatively complex world definitions. Secondly, The literature review that we carried out in our previous and current work indicates that recurrent neural networks have always been used (as opposed to feedforward neural networks) to control autonomous agents. More specifically, recurrent neural networks have always been the only choice whenever the gradient function is not available (the correct output is unknown). However, we found that recurrent links are not required in the neural network to successfully control autonomous agents in environment with hidden system states (e.g. time delay in active cells and rate of change of internal variables).

5.4. SUMMARY

A new ALA Training System has been developed, which evolves both the topology and the link weights of recurrent neural networks that control autonomous life agents. The new system is robust, flexible, fully configurable, and efficient. The contribution of this research extends beyond building training and simulation environment. It leads to discovering that a few nodes in the hidden layer of the RNN are sufficient to control ALAs in complex world definitions. Furthermore, this research showed through empirical experiments that recurrent links do not enhance the capabilities of neural networks in controlling autonomous agents.

5.5. FUTURE WORK

Further experiments on the scalability of the system are required. Ideally, this would involve the evolution of ALAs with practical applications. Building a physical ALA robot is one of the long-term goals of this research.

The current system is discrete and not very dynamic. Our objective is to extend the current architecture to t support continuous input and output spaces, as well as introducing a mutli-agent environment.

From a more general point of view, we are interested in comparing the training time and the performance of feedforward networks vs. recurrent networks using both conventional and evolutionary training algorithms.

BIBLIOGRAPHY

[1] T. Abou-Assaleh and J. Zhang, "*Autonomous Life Agent Using Recurrent Neural Networks and Genetic Algorithms*", Late Breaking Papers - Proc. Genetic and Evolutionary Computation Conference, pp. 1-5, 2000.

[2] R.K. Belew, J. McInerney, and N.N. Scharaudolph, "*Evolving Networks: Using Genetic Algorithms with Connectionist Learning*", CSE technical report CS90-174, University of California at Dan Diego, La Jolla, CA, 1990.

[3] M.T. Hagan, H.B. Demuth, and M. Beale, "*Neural Network Design*", Boston, MA: PWS Publishing Company, 1996.

[4] S.A. Harp, T. Samad, and A. Guha, "*Towards the Genetic Synthesis of Neural Networks*", Proc. Third International Conference on Genetic Algorithms, pp. 360-369, 1989.

[5] J.H. Holland, "*Adaption in Natural and Artificial Systems*", MIT Press, 1992.

[6] J.R. Koza and J.P. Rice, "*Genetic Generation of Both the Weights and Architecture for a Neural Network*", Proc. IEEE International Joint Conference on Neural Networks, pp. II-397 - II-404, 1990.

[7] Jm.M Molina, A. Torresano, I. Galván, P. Isasi, and A. Sanchis, "*Evolution of Context-free Graph Grammers for Designing Optimal NN Architecture*", Proc. Genetic and Evolutionary Computation Conference Workshop Program, pp. 61-63, 2000.

[8] A. Onat, H. Kita, and Y. Nishikawa, "*Recurrent Neural Networks for Reinforcement Learning: Architecture, Learning Algorithms and Internal Representation*", Proc. IEEE International Conference on Neural Networks, pp. 2010-2015, 1998.

[9] D. Polani and R. Miikkulainen, "*Eugenic Neuro-Evolution for Reinforcement Learning*", Proc. Genetic and Evolutionary Computation Conference, pp. 1041-1046, 2000.

[10] T. Ragg, H. Braun, and H. Landsberg, "*A Comparative Study of Neural Network Optimization Techniques*", Proc. International Conference on Artificial Neural Nets and Genetic Algorithms, pp. 341-345 1997

[11] J.D. Schaffer, D. Whitley, and L.J Eshelman, "*Combination of Genetic Algorithms and Neural Networks: A Survey of the State of the Art*", Proc. International Workshop on Combinations of Genetic Algorithms and Neural Networks, pp. 1-37, 1992.

[12] A.P. Wieland, "*Evolving Neural Network Controllers for Unstable Systems*", Proc. IEEE International Joint Conference on Neural Networks, pp. II-667 - II-673, 1990.

APPENDIX A: ALA TRAINING SYSTEM USER MANUAL

Introduction

The ALA Training System is designed to evolve autonomous life agents (ALAs) that are able to live in the user-defined environments. It uses the genetic algorithms (GA) approach to evolve recurrent neural networks (RNN) that represent agents. It consists of two parts: (1) command line program and (2) graphical user interface (GUI). The command line program, `ala`, is an implementation of the training engine using C programming language. The complete set of parameters is described in the Command Line Syntax subsection. There are two types of input files that are accepted by `ala`: (1) world definition file and (2) genome definition file. In addition to the output to the standard output stream, the program produces three output files: (1) evolution statistics file (`*.stt`), (2) global variables definition file (`*.glob`), and (3) best genome definition file (`*.bst`). ALA Viewer is a GUI that is written in Java, which enables the user to visualize how the agent lives in the environments. The ALA Viewer is uses the `ala` program to perform all the necessary computations and produces graphical interpretations of `ala`'s output.

System Requirements

Both UNIX and WINDOWS executable file of the `ala` program are provided. The source code is completely cross-platform compatible and may be compiled on any system. The ALA Viewer requires a Java Virtual Machine (JVM). A Pentium class processor with at least 48 MB or RAM is recommended to run the two applications.

ALA Command Line Syntax

```
ala world <n> <w1> <w2> <..> <wn> [maxage <ma>] [popsize <ps>]
[gmr <gmrate>] [wcp <wcprob>] [wmr <wmrate>] [wmp <wmprob>] [lmp
<lmprob>] [elites <e>] [tsize <ts>] [gens <g>] [hidden <h>] [vars
<v>] [delta <d1> <d2> <..>] [seed <rseed>] [minsize <msize>]
[evals <ev>] [mode <m>] [base <name>] [best <initbest>] [pos <x>
<y>]
```

Parameters:

- `world <n> <w1> <w2> <..> <wn>`
  Load <n> world definitions in the memory. `<w1>` to <wn> are the file names for the world definitions.
- `maxage <ma>`
  Set the maximum age to `<ma>`. Default 100.
- `popsize <ps>`
  Set the population size to <ps>. Default 10.
- `gmr <gmrate>`
  Set the genome mutation rate to <gmrate>. Default 0.01.
- `wcp <wcprob>`

Set the weight crossover probability to <wcprob>. Default 0.25.

- `wmr <wmrate>`
  Set the weight mutation rate to <wmrate>. Default 0.90.
- `wmp <wmprob>`
  Set the weight mutation probability to <wmprob>. Default 0.05.
- `lmp <lmprob>`
  Set the layer mutation probability to <lmprob>. Default 0.70.
- `elites <e>`
  Set the number of elites to <e>. Default 1.
- `tsize <ts>`
  Set the tournament size to <ts>. Default 3.
- `gens <g>`
  Set the number of generations to <g>. Default 10.
- `hidden <h>`
  Set the number of hidden layers to <h>. Default 1.
- `vars <v>`
  Set the number of internal variables to <v>. Default 1.
- `delta <d1> <d2> <..>`
  Set the rate of change of each internal variable to <d1>, <d2>, .. , respectively. Default -0.1.
- `seed <rseed>`
  Set the random number seed to <rseed>. Default: value return by the C function `time(NULL)`.
- `minsize <msize>`
  Set the desired size of genome to <msize>. -1 disables size comparison during evolution. Default 500.
- `evals <ev>`
  Set the number of random evaluations for each genome to <ev>. 0 disables the random evaluation and forces using the values specified by pos. Default 50.
- `mode <m>`
  Set the mode to <m>. 0 is training, 1 is testing, 2 is displaying, and 3 is debugging. Default 0.
- `best <initbest>`
  Load an initial globally best genome from the file <initbest>. Default: a random genome is created.
- `pos <x> <y>`
  Set the initial position of the agent to <x>, <y>. Default 1, 1.

How to Run ALA Viewer

To run the ALA Viewer type the following at the command prompt:

```
java ALAViewer
```

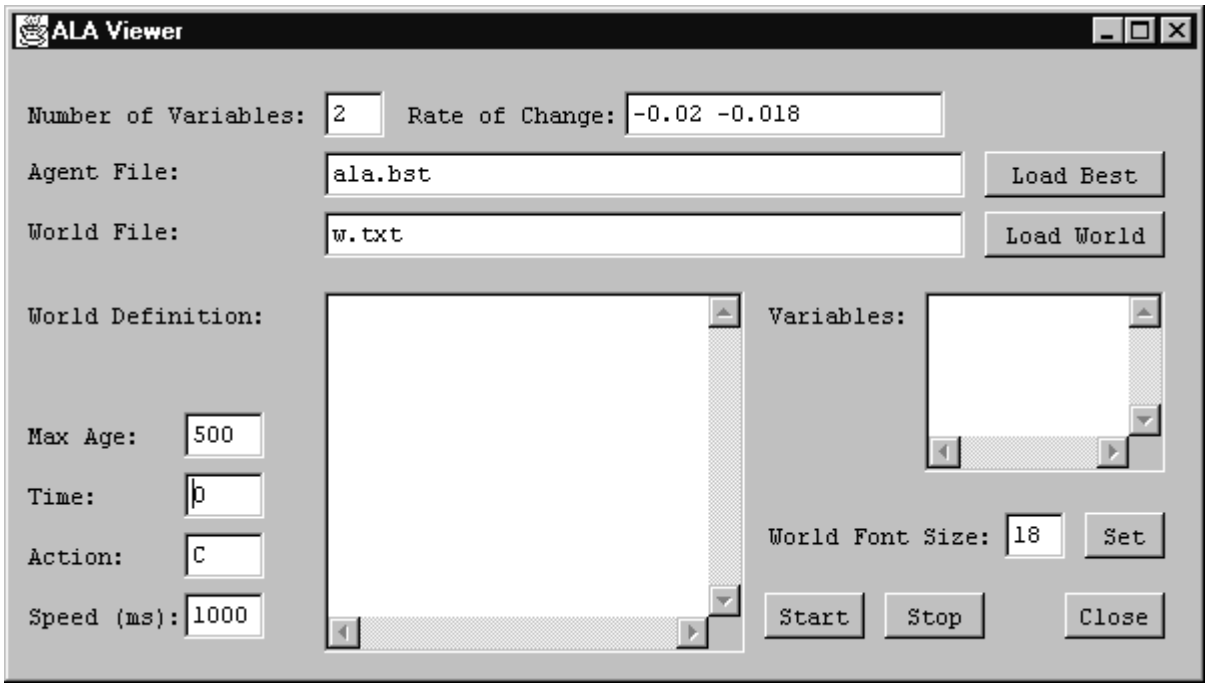

Figure 11: ALA Viewer

A window will appear similar to Figure 11. First, Fill in the number of internal variable and the rate of change for each internal variable. Second, Load an agent definition file and a world definition file. Finally, specify the maximum age and the speed (the time delay between consecutive actions in milliseconds). The button "Start" will begin displaying the agent's actions and the button "Stop" will end the process. The font size of the world display may be modified using the font size text box and the button "Set." The "Close" button terminates the application.

Format of World Definition File

The first two values specify the number of rows and columns in the world. Following that there is one line for each active cell. Each of these lines begins with two values specifying the X and Y coordinates of the active cell. Next, a single character defines the type of the cell. After the cell type comes a one-word title of the cell. The next value specifies the index of the internal variables that is affected by this cell (starting from 1). The cell's time delay is the next value. The last value is the amount of increment that this cell provides. The following is an example of a world definition file:

```
7
7
2 2 H Home     1 10 1.0
5 5 E Energy   2 15 1.0
```

Format of Genome (Agent) Definition File

The first value is the number of hidden layers. After that, there is one value for each hidden layer specifying the number of nodes in that layer. Next is one value for each hidden layer specifying the number of recurrent links in that layer. The next value is total number of weights (including initial output values for recurrent links). Next, there is one value for every weight. The following is an example of a genome definition file:

```
1 4 0 105
 -0.487791510 -0.388796829 -0.992042040 0.101039236 0.861947644 0.780476216
0.564192216 -0.838847545 0.411977381 0.487523938 -0.252951606 0.228211799
-0.479043128 0.142837403 -0.763041791 -0.436325832 -0.543671335 -0.776513655
0.729281704 -0.624349773 -0.050145479 0.047527424 0.802059433 -0.937069897
-0.308992261 -0.161285452 -0.292102159 -0.431027178 0.222042862 -0.211699285
0.072821333 0.875483108 -0.643474489 -0.718366230 0.985342289 0.221464961
0.731207644 0.967866225 0.586116998 0.998661121 -0.037277534 0.753794277
0.586178925 0.402931890 -0.393218164 0.545127764 -0.702465110 -0.375675640
-0.407809684 -0.405100958 0.334371493 0.655562825 0.191089886 -0.519542450
-0.970116595 -0.362874918 -0.630915121 0.049618508 0.267308980 -0.239554693
-0.461757576 0.822227075 -0.822413990 -0.737555120 0.437324289 0.070144599
0.679762671 0.790847624 -0.024072026-0.813318106 -0.512227574 0.785647854
-0.714510770 -0.960406410 0.603458938 0.081742290 0.916167998 0.780476216
0.812438679 0.450494624 -0.442589143 0.064592102 -0.178728435 -0.932840938
0.989301183 -0.503072300 -0.431811368 0.932210570 -0.412029235 -0.020749068
0.950362170 -0.811025132 -0.576479298 0.582961767 -0.200455176 -0.718854519
-0.581839548 0.545984370 0.405305683 0.655469203 -0.898106380 -0.603546223
0.881154843 0.030360403 0.871659079
```

Format of Evolution Statistics File

The evolution statistics file contains a number of lines where each line represents a generation. Each line begins with the generation number. Next is the population average age. The next two values are the age and the size of the best genome so far. The last two values are the age and the size of the best genome in the current population. The following is an example of an evolution statistics file:

```
0    12.381    52   57   52   57
1    19.450    52   57   52   60
2    27.143    54   57   54   57
3    33.101    54   57   53   57
4    36.306    54   57   54   57
5    39.151    55   57   55   57
```

Format of Global Variables Definition File

The global variable definition file consists of the following fields:

- maximum age
- population size
- genome mutation rate (vs. genome crossover rate)
- weight crossover probability (for n-point crossover)
- weight mutation rate (vs. topology mutation rate)

- weight mutation probability (for n-point mutation)
- layer mutation probability (for topology mutation)
- number of elite genomes to be replicated
- tournament size for tournament selection
- maximum number of generations
- number of hidden layers
- number of internal variables
- rate of change of each internal variable
- number of inputs in the RNN
- number of outputs in the RNN
- the random number seed
- initial x position
- initial y position
- desired genome size
- number of random evaluations
- base name for the output files
- operation mode
- genome definition of the best genome (as define in the format of genome definition file)
- number of worlds
- for each world: the dimensions, a character view of the world, and the definition of each active cells.

The following is an example of a global variables definition file:

```
512
2000
0.300000
0.250000
0.900000
0.050000
0.700000
3
4
2000
1
3
 -0.020000 -0.018000 -0.016000
15
9
977969475
1
1
200
15
ala_3v_1w_r
0
1 4 0 109
 -0.214550741 -0.248076952 0.010579784 -0.804799145 -0.801876465 -0.374390835
0.396650477 -0.956694973 -0.869582776 0.464848267 -0.162858453 -0.804475390
```

```
0.944154038 0.244434542 0.926674729 0.270597927 0.909197370 0.557101074
0.357524880 -0.647127980 -0.169116178 -0.627607894 -0.702211774 -0.687195626
0.935703482 -0.554548450 -0.002533579 0.896712752 -0.126801941 -0.911425880
-0.574732944 -0.317665268 -0.247221028 0.619665751 -0.875078181 0.965815112
-0.022019832 0.315835115 0.541181993 -0.617638180 -0.906930094 0.145279359
-0.297663170 -0.837688360 -0.818266471 -0.873111702 0.860677224 -0.598688314
-0.021303054 0.937771542 -0.961463807 -0.460164247 0.268569032 -0.791710539
-0.205383172 -0.172651090 -0.221388235 0.992802577 -0.016751012 -0.191259759
0.085527298 0.103927631 0.708229602 -0.620400470 -0.018122506 -0.177390213
-0.989229700 -0.142034463 0.612491287 0.534098653 -0.393139525 -0.936762741
-0.824691409 0.883259340 -0.594509049 -0.343633414 -0.072391846 0.766519287
-0.501851964 0.038890249 0.433747405 -0.840701849 -0.930429919 0.395953174
-0.157623413 -0.162858453 0.167847017 0.902435617 0.308123084 0.409173926
0.499610168 0.425365403 0.066499267 -0.795992795 -0.121202648 -0.704990844
0.454944391 -0.603993861 -0.987037981 0.238003517 0.241578716 -0.027926904
0.529325153 0.897679159 -0.854058571 -0.874492119 0.599165931 -0.641871843
0.940190612
1
7
7
#######
#.....#
#.H...#
#....E#
#.....#
#.M...#
#######
2 2 H Home 1 10 1.000000
5 3 E Energy 2 15 1.000000
2 5 M Minerals 3 7 1.000000
```