Evolving Artificial Neural Network Controllers for Autonomous Agents Navigating Dynamic Environments

Robert A. Lucas

B.Sc., University of Northern British Columbia, 2002

Thesis Submitted In Partial Fulfillment Of

The Requirements For The Degree Of

Master of Mathematical, Computer, and Physical Sciences

in

Computer Science

University of Northern British Columbia

2008

Your file Votre référence
ISBN: 978-0-494-48781-5
Our file Notre référence
ISBN: 978-0-494-48781-5

ABSTRACT

This thesis presents and discusses a potential method for solving the dynamic obstacle avoidance problem using contemporary work with artificial neural networks (ANNs) and genetic algorithms (GAs) in combination with an imitation of a biological genetic process called segmental duplication. ANNs, GAs and segmental duplication are merged in the project to form SDNEAT, a type of evolutionary artificial neural network (EANN) system based on NeuroEvolution of Augmenting Topologies, or NEAT. The system is then used to develop an artificial neural network system that attempts to navigate environments incorporating both static and dynamic obstacles.

TABLE OF CONTENTS

Contents

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

First I would like to thank Dr. Charles Brown. He was always supportive of me during my unsteady progression through this thesis. He was always interested in my thoughts and excited about my research, and he kept me on the path when I wandered off. I would like to thank my friends and co-workers for being patient with me, and offering kind words and keen insights. Hopefully I can return the kindness and support, should you need it one day.

I would like to thank my parents; I wouldn't be here without them. They taught me to think about a situation rather than take it at face value and to work hard for something I want, since nothing is ever free.

Finally I would like to thank my fiancée Naomi. I never would have finished this without you. You were my sounding board for ideas and frustrations, and you took it all in stride. You supported me the entire time, even when you were more frustrated than I was. Your incredible literary skills shine in this document and your intelligence helped me find solutions that should have been obvious.

I have achieved something with this thesis that I wasn't sure I could. I consider this a solid foundation for my future, and I look forward to continuing my research.

Chapter 1

NEURAL NETWORKS AND GENETIC ALGORITHMS: AN INTRODUCTION

Human beings are extremely complex systems that move about performing tasks in the real world without much care for other entities. They are capable of setting a goal, planning to achieve that goal, carrying out the tasks involved in the plan while adapting to changes in the environment and finally achieving their goal. Computer systems have yet to achieve this level of ability in simulated or real-world systems. Currently even the most advanced systems cannot adapt to major changes in their environments. These changes can cause the entire system to fail completely. In order for computer systems to be able to integrate more fully into the real world they clearly need to learn how to adapt to drastic environmental changes.

The challenge of complete integration into a real-world environment is enormous. The complexities of the problem quickly become apparent when one attempts to define how a robotic system should behave when confronted with dynamic obstacles. A robot should not be incapacitated if a person moves in front of it and refuses to get out of the way. Similarly, if a large crowd moving at a quick pace is coming towards the robot and blocking its planned path, the robot should be able to manoeuvre through it. These are just two examples of what a robot may have to handle in a real-world environment; the number of possible scenarios is too large to count. Therefore a robust system must be developed to help robots handle complex and unforeseen situations. The problem can be explored in a virtual environment using virtual robots called "agents". Using neuroevolutionary programming, the agents' control systems can be evolved to navigate in environments that include both static and dynamic obstacles. This project's purpose is to use machine learning techniques to evolve an agent control system that can cope with a dynamic environment such as the one in the example below.

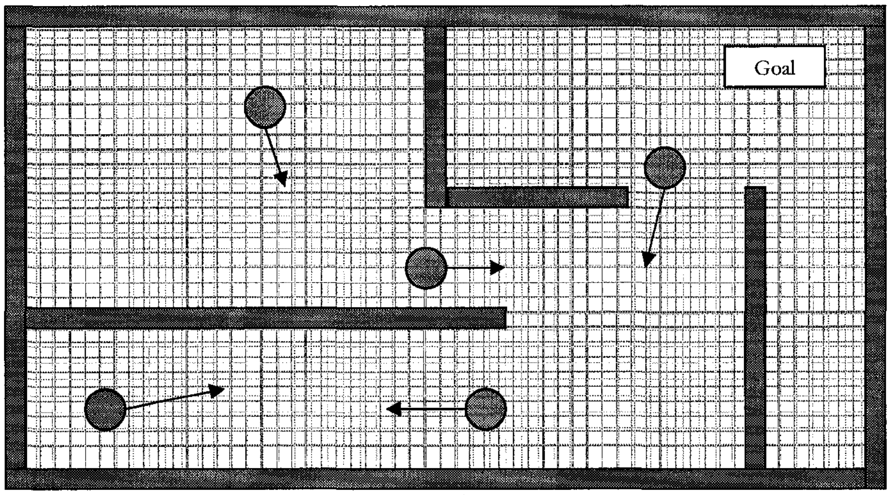


Figure 1.1 Example Environment: Autonomous Agent obstacle avoidance example situation.

This project explores this problem using a combination of artificial neural networks and genetic algorithms, both of which are machine learning systems, to evolve a simulated autonomous agent that can navigate in various test environments from its starting point to a goal while perceiving said environment through a predefined sensor package and avoiding all static and dynamic objects in its way.

In order to develop such a system, a control architecture that efficiently handles changes to the environment is necessary. It must be relatively easy to train and not overly complex. An artificial neural network is the ideal tool for this task. However, the optimal solution will also incorporate an optimal neural network topology. Therefore an evolutionary approach to artificial neural network construction should be employed to create a relatively efficient autonomous agent controller.

An evolutionary artificial neural network (EANN) is a union of two different branches of computer science: Artificial Neural Networks (ANNs) and evolutionary algorithms or genetic algorithms (GAs). In essence, EANNs are a specific type of artificial neural network that use a different method for learning in addition to the standard ANN methods. While standard ANNs can adapt to dynamic environments, EANNs' combination of evolutionary and neural learning allows them to adapt more quickly (Yao, 1999) and take advantage of temporal information as well (Nelson, Grant, Galeotti, & Rhody, 2004). In this regard, EANNs can be considered generic adaptive systems, which means they can change their architectures and learning methods to suit the problem without human involvement.

This project's autonomous agent controllers require an environment in which to learn. Since the required amount of machine learning would take prohibitively longer to achieve in the real world, a simulated world must developed for the autonomous agents. This environment must incorporate static obstacles, dynamic obstacles and real-world physics. It must also be capable of providing accurate sensor information for the simulated sensors. This sensor information is attenuated using a sensor noise value so that the simulated autonomous agents would be more likely to perform well in a real-world environment in which the hardware-based sensors are imperfect and would be incapable of providing ideal data.

Among the issues investigated in this project is whether or not an autonomous agent using EANNs can learn to avoid static as well as dynamic obstacles and still manoeuvre. This project's primary goal is to develop a system to evolve an efficient neural network controller that can learn to effectively operate an autonomous agent in multiple different dynamic environments.

1.1 Requisite Knowledge

To implement an EANN, several different types of technology are required. The first requirement is an artificial neural network which functions as the controller and learning component of the agent. The second requirement is a genetic algorithm that performs the optimization procedure on the agents. The third component is the concept of a virtual autonomous agent, or alternatively, a system that can function in a real-world environment without human intervention. Finally, due to the problem at hand, it is also necessary to understand the concept of a dynamic environment and the unique problems that occur in such an environment. A discussion of these basic concepts will follow before an examination of current work in the field.

1.1.1 Artificial Neural Networks

Artificial Neural Networks are a very common technology used in many production systems today. They are currently the leading electronic simulation of the way a living neural system functions and as such they will probably remain in common use for some time. Since their conception they have been developed for several practical applications in the real world; some are discussed in (Knoblock, 1996), such as voice recognition and biometric systems.

An ANN is a mathematical model for information processing that uses a connectionist approach to computation. It is based on the neuroelectric systems of the human brain. The smallest unit of a neural network is a neuron, which stores a small portion of data about information to which the neural network has been exposed. This small portion of data can be referred to as the neuron's "weight". These neurons are interconnected to form a network of nodes that can perform complex recognition based on the set of inputs with which they have been trained.

ANNs are widely thought of as a black box form of machine learning. The term *black box* refers to the idea that, once information is captured in an ANN, it cannot be readily retrieved from the same ANN in a useable form. This is not entirely accurate, as the information can be retrieved and understood. The mathematics of ANNs are extremely complex and can be difficult to decipher. However, there are several methods of rule extraction for ANNs (Tsukimoto & Hatano, 2003) as well as methods for visualization of the information stored in an ANN.

Today, ANNs are considered one of the best methods for solving complex nonlinear multidimensional problems (Tsukimoto & Hatano, 2003). ANNs lend themselves well to solving difficult real-world problems that cannot be solved using a straightforward algorithmic method (Knoblock, 1996). A simple multilayer perceptron with one hidden layer is provably capable of approximating any continuous function with arbitrary accuracy (Cybenko, 1989). ANNs are also computationally complete; they are equivalent in class to Turing machines. ANNs can do anything a computer can (Cybenko, 1989), and do not require a complete set of data to learn to accomplish a task (Knoblock, 1996). Usually, only a small amount of input data is necessary to train the network to approximate whatever is required of it. ANNs are used in several fields today including aerospace, banking, robotics and linguistics (Knoblock, 1996).

This project does not consider ANNs alone; they are used here in conjunction with another advanced problem-solving system known as a genetic algorithm.

1.1.2 Genetic Algorithms and Evolutionary Methods

Genetic Algorithms are among the most advanced search algorithms available today. They are capable of searching extremely convoluted and complex multidimensional search spaces and finding optimal solutions in an acceptable amount of time (Janson & Frenzel, 1993). GAs were invented by John Holland (Koza, 1998; Srinivas & Patnaik, 1994). Holland developed GAs in cooperation with his students and coworkers in the early 1970s. Genetic programming, which is a variation of GAs, was developed in the early 1990s by John Koza (Koza, 1998).

Genetic algorithms are a type of evolutionary computing and are inspired by Darwin's theory of evolution (Srinivas & Patnaik, 1994). Genetic algorithms are directly based on biological systems; terms used in relation to them include *gene*, *chromosome*, *recombination*, *mutation* and *crossover*. In a GA, a gene is a representation of the data being evolved by the GA. Working with a GA involves the management of a population consisting of a set of genes. Each set of genes can serve as a parent generation for the next set of genes. Crossover and mutation are the two operators that a GA employs. Crossover occurs when two genes are split at related locations and their respective elements switch places with each other to form offspring. Mutation occurs when smaller portions of those genes change to form a new gene with different characteristics from the original.

All GAs follow the same basic algorithm:

Start – Generate a random population of n chromosomes.

Fitness –Evaluate the fitness f(x) of each chromosome x in the population.

New Population – Create a new population by repeating these steps:

> **Selection** – Select two parent chromosomes from a population based on their level of fitness.
>
> **Crossover** – If determined by the defined crossover probability, crossover the parents to form a new offspring. If there is no crossover, the offspring is an exact copy of the most fit parent.
>
> **Mutation** – If determined by the defined mutation probability, mutate the offspring.
>
> **Accepting** – Place offspring in the new population.

Replace – Use the newly generated population for the next run of the algorithm.

Test – Test for the end condition. If it is found, output the solution. Otherwise, return to the Fitness stage.

When developing a GA a programmer must carefully consider several issues, such as how to create genes from the data. If the genes are poorly encoded, the algorithm may be extremely inefficient or unable to use both the crossover and the mutation functions. At the same time, a method for performing mutations and crossovers must be developed. The programmer must also choose the size of population to use. Typically, a modest population works well, but this is not always the case.

Among the most important issues the programmer must manage is the development of a fitness function. The function that determines the fitness level of each gene must be neither too simple nor too complex. If the function is too simple, the networks may not effectively localize. If it is too complex, they may not localize at all or may reach a non-optimal solution. The programmer must also choose how many parent genes will be selected from the population to create a new generation, and whether or not elitism should be used. Elitism can be thought of as "survival of the fittest"; when it is used, the fittest genes from a generation are passed on to the next generation without undergoing any crossover or mutation.

In a sense, a GA is a directed search in that there is a goal and the algorithm checks many possible solutions to see if any of them work. The GA expands its search in the direction of whichever possible solutions have been determined to be closest to a working solution. GAs are capable of searching spaces that cannot be visualized or perceived, and are used in various fields today including automated design, distributed computing, protein folding and scheduling.

This project integrates GAs with ANNs. This presents a problem, as ANNs store very complex information. There are several different methods for encoding the information in an ANN and applying a GA effectively to an ANN system. These methods will be discussed later in this paper.

1.1.3 Autonomous Agents

Autonomous agents are software and robotic entities that are capable of independent action such as reacting to their environments, interpreting and planning in an open and unpredictable environment. Autonomous agents are an extremely important field of research today in computer science and robotics.

An autonomous agent can set out to perform a complex task and complete that task without any human intervention. Because programming an agent to perform these tasks is a very complex problem, researchers have been using evolutionary programming to solve it and they have met with some great initial success (Sharkey, 1997).

Currently, most autonomous agents are small robots designed to do simple tasks like find their way through a maze (Floreano & Mondada, 1994) or pick up paper balls in a certain area (Mondada & Floreano, 1995). Other autonomous agents are large robots working in a production environment and are extremely complex systems capable of performing several tasks completely independently of human intervention (Xu, Van Brussel, Nuttin, & Moreas, 2003).

Some of the problems with autonomous agent systems in existence today can present serious obstacles to development. Currently most of the training of autonomous agents occurs in simulation; however, these simulated environments are not as demanding as the real world. A great deal of current research focuses on creating effective means of training autonomous agents in simulation so that minimal training is required in the real world (Miglino, Lund, & Nolfi, 1995). The generational systems that train these agents would take far too long to complete their training in real-world time.

Another problem that researchers have encountered is that several of their systems are developed for small robots which are not capable of performing significant physical tasks. These robots are limited to pushing small light objects that serve no practical purpose (Mondada & Floreano, 1995). When these robots are scaled up to a larger size, new problems are presented that presented no difficulty to a smaller robot. For example, a robot that is five centimetres in diameter might not damage itself badly if it impacts a wall at full speed. A robot with a two metre diameter colliding with the same wall at the same speed might be likely to destroy both itself and the wall.

Despite these problems autonomous agents are an extremely popular research topic; the benefits of having effective autonomous agents greatly outweigh the costs of their initial training.

1.1.4 Dynamic Environments

In a static environment, nothing ever changes. In a dynamic environment, things can change unpredictably. Even people who are unfamiliar with how an autonomous agent works can see that navigating through a static environment would be much easier for a robot than navigating through a dynamic environment. In a static environment every object can be used as a landmark for navigation. In a dynamic environment no feature can be considered static; even the walls could move. Therefore a different approach must be used for navigation.

An autonomous agent navigating through a dynamic environment must reconsider everything in its movement plan every time it considers a change of course. This increases computational time for any system that does significant planning work. This computational time can be prohibitive even with the powerful computer hardware of today. A system working in a dynamic environment should be able consider the current state of the environment and choose a new course extemporaneously. The calculation should not be dependent on some far-off landmark, but should be based on the current situation the autonomous agent perceives in its immediate vicinity.

Some of the inherent problems in a dynamic environment present significant challenges to an autonomous agent. The agent has to be able to calculate expected trajectories of dynamic obstacles in its environment so that it can plan early for avoidance and thus plan an efficient route to its goal or avoid danger.

However, the agent must also be highly adaptive; if it plans too early, the dynamic obstacle may change its trajectory and the agent's plan will no longer be viable. If the agent can plan its course extemporaneously it will be able to adapt quickly to such changes. An agent that cannot consider course changes extemporaneously would also be quickly overwhelmed by large numbers of dynamic obstacles that present a danger of collision. An adaptive agent would only be concerned with dynamic obstacles in its immediate vicinity, and would ignore more distant obstacles. Through machine learning methods and a properly honed fitness function, adaptive behaviour should emerge from highly-evolved autonomous agents.

Dynamic environments present a challenge that is beyond the capabilities of existing autonomous agent control systems. There is great potential for research in this field; to date, the relevant research is limited. I believe that EANNs can provide an effective means for the creation of a control structure that can handle this problem.

Chapter 2

CURRENT PRACTICE

Among researchers there is presently a great deal of interest in EANNs. There have been many recent breakthroughs and several new types of EANNs have emerged from research projects (Yao, 1999). Some of this research has been applied to the areas of autonomous agents and obstacle avoidance (Floreano & Mondada, 1994; Floreano & Mondada, 1998). There has been very little research concerning dynamic obstacle avoidance with autonomous agents using EANNs (Aguilar & Jose, 1994). The current state of the field suggests that these components may work well together to reach this research project's goal of evolving an agent control system that can cope with a dynamic environment. To show how GAs and ANNs can be brought together to solve this problem, this section will review the current research in the relevant fields.

2.1 Evolution of Artificial Neural Network Systems

While standard ANNs are powerful tools for problem solving, they do present difficulties as was discussed in Chapter 1. In an attempt to circumvent these problems various methods have been investigated for the creation and honing of new types of ANN systems. These new systems are known as Evolutionary Artificial Neural Networks, or EANNs (Yao, 1999). EANNs differ from standard ANNs in that they have an extra stage of adaptation and learning based on an evolutionary or genetic system (Yao, 1999). There is a variety of types of EANN systems available for use, and they can be broken down into four categories (Yao, 1999). For the sake of simplicity the first three of these four categories of EANN, which were defined by Xin Yao, (Yao, 1999) will be referred here as weight-evolving algorithms (WEAs), topology-evolving algorithms (TEAs) and hybrid evolving algorithms (HEAs). The fourth category incorporates a wide variety of other types of EANNs.

One type of EANN system involves the evolution of network weights. As was explained in Chapter 1, an ANN is a set of weighted nodes and connections between those nodes. The weights contained in the nodes are in the form of matrices that contain information from prior input data. Through backpropagation, these weights are updated and the overall network is trained to recognize certain patterns. The process of backpropagation can be long and computationally intensive, and in some cases it does not result in an effective solution. In such a case a weight-evolving algorithm (WEA) can be applied, which may speed up the search for a solution (Janson & Frenzel, 1993).

A second type of EANN involves the evolution of architectures or topologies of ANNs. Instead of modifying the learning algorithm the ANN uses or augments it with a GA, the topology-evolving algorithm (TEA) relies on standard backpropagation while attempting to find the best ANN structure for the problem. An ANN with a better structure can learn faster or result in a more optimal solution in less time than a less well-structured counterpart. This method is particularly well suited to an EANN algorithm and appears to be a more popular method (Yao, 1999).

Hybrid evolutionary systems are a third type of EANN. Hybrid evolutionary algorithms (HEAs) are an attempt to merge the first and second methods of evolving weights and structure into one algorithm (Yao, 1999). Hybrid evolutionary EANN algorithms are typically more complex than the first and second types of EANNs and need to take into account more variables in the systems they are designed for. However they can be extremely efficient and powerful in finding an efficient ANN structure and weight set (Nissinen, Koivo,

& Koivisto, 1999; Stanley, Bryant, & Miikkulainen, Evolving Adaptive Neural Networks with and without Adaptive Synapses., 2003).

Finally, some researchers have taken novel approaches to developing EANNs and have created radically different systems that are sometimes similar to the three methods discussed earlier but diverge enough from them to be a considered a different type of system altogether (Aliev, Fazlollahi, & Vahidov, 2001; Arifovic & Gencay, 2001; Golubski & Feuring, 1999; Tsukimoto & Hatano, 2003; Capi & Doya, 2005). Some of these systems will be discussed here but they will not be a major focus of this section.

The various types of EANNs have been extensively studied; many of the following points stem from the work of prior researchers. This chapter discusses and summarizes the prior research as well as presents some examples of algorithms that can be applied to the problem of mobile object avoidance in autonomous robotic agents.

2.1.1 Weight-Evolving Algorithms (WEAs)

Weight-evolving algorithms, or WEAs, use genetic algorithms to evolve the weights of an EANN's nodes. Most systems that take this approach use a method that minimizes an error function such as the mean squared error (Yao, 1999). This is how an ANN is trained; backpropagation and conjugate gradient algorithms, which are standard ANN training algorithms, already take the mean squared error into account. This is a difficulty with ANNs, as they can often become trapped in a local minimum of the problem space. Standard GAs are less likely to become trapped in these local minima unless the search space of the ANN is extremely convoluted.

In such a case, a WEA can help overcome this problem without sacrificing the power of an ANN. In a WEA system, the set of weights in the network nodes is evolutionarily adapted. Standard backpropagation would perform the same feat, but also could become trapped in a non-optimal solution. Using a GA, this is less likely to occur. In order to use a WEA, first a representation of the data must be chosen. There are two popular formats: binary and real number. The second phase of developing the WEA involves choosing the operators for mutation and crossover and deciding whether or not either or both will be used. In addition, different representation schemes can lead to radically different performance and as such should be selected carefully.

Binary representation is commonly used to represent data in GAs. It makes the operations of mutation and crossover easy to perform but consistency checking must be applied so that offspring are functional rather than illegal or inoperable. It is simple to use binary representation of the data. First, an algorithm is defined to extract the weights from the ANN in a specific order. Then the weights are converted into a fixed length binary string. Once the data is converted, the GA is performed on the dataset and the information is converted back to its standard form with a reversal algorithm. Finally, the information is placed in an offspring for the next iteration of the GA (Janson & Frenzel, 1993; Tsukimoto & Hatano, 2003; Yao, 1999).

Real number representations can also be used to encode the weights of an ANN. The same method is used as in binary representation to extract and then re-encode information back into the ANN. However in real number representations, instead of changing the extracted weights to binary, they are represented by a single real number (Alsultanny & Aqel, 2003; Yao, 1999). While this scheme is easy to encode and decode, its primary operator is mutation and crossover is considerably harder to implement here than in a binary representation. This can hinder the efficiency of the algorithm but will not completely halt its progress; it has been shown that GAs can operate effectively using only one of their two major operations (Siebel, Krause, & Sommer, 2007).

WEAs have been used effectively in many circumstances. Training neural networks to identify the most efficient width of a CMOS circuit has been a problem that is not easily programmed but it can be accomplished using a WEA (Janson & Frenzel, 1993). When this was achieved the ANN used did not initially appear to solve the problem. Upon further investigation it was found that the search space was extremely convoluted and could not easily be searched even using a GA. Therefore a penalty function was employed to force the GA to search in areas that were closer to a solution. This involved manipulation of the problem, which required domain knowledge. Such knowledge may not always be available.

WEAs have also been effectively used in image pattern recognition. In that case the network was large and complex but the WEA was nevertheless able to adapt relatively quickly. It offered excellent results when detecting the orientation of a picture of a jet airplane (Alsultanny & Aqel, 2003).

Using a slightly different method, WEAs have also been used to increase the functional localization of an ANN (Sexton & Gupta, Comparative evaluation of genetic algorithm and backpropagation for training neural networks., 2000). In some cases an ANN can be developed and trained, and may give excellent results, but can be functionally localized and therefore is not the most efficient implementation of that network. To detect this problem, an algorithm can be implemented that extracts Boolean functions for each of the hidden layer nodes of an ANN. If the extracted function is too convoluted it can be deemed nonlocalized and the WEA can be used to localize the function further (Tsukimoto & Hatano, 2003).

There are many different types of WEAs and they seem to be effective methods for learning.

2.1.2 Topology-Evolving Algorithms (TEAs)

The next type of EANN is the TEA, which evolves ANN architectures or topologies. An ANN can be accurately represented by a graph. An ANN is a graph-like structure and has an architecture or topology that can be modified. Changing an ANN's topology can drastically improve or deteriorate its performance. In the past, engineering the topology of an ANN has been a job for a human being; this was a trial-and-error procedure. Since there is an infinite set of possible network structures available to solve each problem, a human being may not be able to find an efficient architecture. However, a TEA can be employed to find an efficient ANN topology that solves the problem.

This system can be more complex than the WEA method. This is because the entire structure of the network may be changed by the TEA and then must be completely retrained. However, it can also be more robust. The changed structures of the network may be capable of retaining very different patterns of information. The algorithm may find a structure that performs excellently that the human designers may never have conceived.

Similarly to WEAs, there are two main things to consider when implementing a TEA: the representation of the ANN or the genotype, and the GA method used to evolve the ANN architecture. When deciding how to represent the ANN in its genome there are two different extremes that may be considered. In one extreme, all the information in the ANN is precisely encoded. This is referred to as *direct encoding*. The other extreme involves the encoding of only the information about the structure of the network that is deemed important, such as how many hidden layers there are, how many inputs there are, how many outputs there are and so on. Once the encoding scheme is chosen the programmer must decide whether to use mutation, crossover or both in the GA. Finally, as applicable, mutation and crossover must be defined so that they can operate on the genome. Once these two points have been settled, the TEA can operate until an effective ANN structure is found (Yao, 1999).

TEAs seem to be more popular than WEAs. This may be because they can be easier to comprehend if a straightforward type of encoding is used. TEAs have been effectively employed in several different situations (Boozarjomehry & Svrcek, 2001; Castillo, Merelo, Prieto, Rivas, & Romero, 2000; Janson & Frenzel, 1993). TEAs have also been modified to perform optimization as well as topographical evolution (Sexton, Dorsey, & Sikander, Simultaneous optimization of neural network function and architecture algorithm., 2004). One of the problems with TEAs is that the ANNs developed with them can grow to be extremely large and convoluted. Fortunately the algorithm can be adapted to perform self-pruning as it is evolving more efficient ANNs. Unnecessary weights and hidden nodes can thus be identified and removed from the ANN, which keeps the network smaller and more efficient (Blanco, Delgado, & Pegalajar, 2000; Castillo, Merelo, Prieto, Rivas, & Romero, 2000; Sexton, Dorsey, & Sikander, Simultaneous optimization of neural network function and architecture algorithm., 2004).

Other modifications of TEAs allow the algorithm a broad capability to adapt to their problems, even allowing the algorithms to define their inputs to the constructed ANNs. While this is a complex problem it allows for an extremely efficient ANN to develop (Arifovic & Gencay, 2001; Nissinen, Koivo, & Koivisto, 1999).

Some other implementations use a graphical representation of the ANN as an encoding scheme for its GA. This method has some similarities to genetic programming and can result in ANNs that are extremely large and inefficient. This method can be modified to restrict the size of the evolved ANNs and eventually evolve an efficient ANN for the problem (Golubski & Feuring, 1999).

Some EANNs use drastically different encoding methods when implemented rather than using direct or indirect encoding. These systems are more akin to a programming language than an EANN but can be used as a basis for the TEA. These languages can be convoluted and difficult to apply to certain domains. However they can also be very efficient in describing the information in an ANN, and they scale well to handle large problems. (Boozarjomehry & Svrcek, 2001; Ilakovac, 1995).

2.1.3 Hybrid Evolutionary Algorithms (HEAs)

The third type of EANN systems, HEAs, is a unification of the two systems described above. These systems adapt both the weight and topology of an ANN. This can be a complex process, but it can also be extremely effective. Both the adaptation of ANN weights and the adaptation of their topologies are effective means for searching a problem space. Combining these two techniques can result in a faster method for finding a solution (Stanley, Efficient Evolution of Neural Networks through Complexification, 2004).

When planning the development of a hybrid evolutionary system one must consider the problems presented by both WEAs and TEAs. In some ways these problems are quite similar. Like WEAs and TEAs, HEAs require a genome representation for which both the mutation and crossover operators are well defined. This representation is critical for a functional HEA.

HEAs have been implemented effectively and they show some very good results which are at least on par with results demonstrated by TEAs and WEAs (Stanley, Bryant, & Miikkulainen, Evolving Adaptive Neural Networks with and without Adaptive Synapses., 2003; Abbass, 2003). One type of HEA involves what is called neurogenetic learning (Janson & Frenzel, 1993; Kitano, 1994). This type is a standard GA combined with ANNs but it uses the GA to develop the structure of the network simultaneously with the weights of the network, rather than randomly inserting weights in the network after its structure is defined by crossover and mutation. A few complex systems are used to determine the values for each stage of the GA: a graph grammar interpreter for structural evolution and a CAM (Cell Adhesion Molecule) matrix for weight evolution. This method is heavily based on biological techniques (Kitano, 1994).

A method similar to neurogenetic learning refers to the problem as a multiobjective optimization problem, or MOP. An ANN is described as a MOP and presented to a mimetic, which is a GA augmented with a local search, to develop an effective HEA (Abbass, 2003).

Another algorithm designed to work as a hybrid EANN is NEAT, or NeuroEvolution of Augmenting Topologies. This algorithm uses a GA to evolve topologies of ANNs and to develop initial weights for the evolved ANNs. NEAT defines an effective method for crossover and mutation while maintaining a fairly simple representation of the ANNs adapted by the system. It also offers some very effective results (Stanley & Miikkulainen, Efficient Evolution of Neural Network Topologies, 2002; Stanley & Miikkulainen, Evolving Neural Networks through Augmenting Topologies, 2002).

There is little research done in the field of hybrid EANN systems, as they are complex. However they do offer effective and efficient search results for EANNs.

2.1.4 Other Methods

Finally, some EANNs do not fit neatly into any of the three described categories, but do share some characteristics with WEAs, TEAs and HEAs. These other EANNs take a more novel approach to one or more of the previous systems and are only mentioned here to indicate the wide range of possible solutions that are being researched.

One method involves the use of co-evolution to speed up the EANN process. In this method there are two GAs competing against each other in the same domain, which drives them both to reach their respective solutions faster than a standard GA would (Sato & Furuya, 1996).

Another method which could be considered an EANN uses fuzzy neural networks (FNNs) instead of ANNs. An FNN differs from a standard ANN in that it can have fuzzy weights or fuzzy inputs, or both. This can present a problem for training as all common ANN training algorithms require static values for weights and inputs, not the range of values a fuzzy variable can represent. However a GA can be used to effectively train an FNN (Aliev, Fazlollahi, & Vahidov, 2001). GAs are very effective training mechanisms; they have been shown to be more effective at training ANNs than standard backpropagation methods (Sexton & Gupta, Comparative evaluation of genetic algorithm and backpropagation for training neural networks., 2000), and have also been shown to train cellular neural networks effectively (Zamparbelli, 1997). Cellular neural networks are a type of distributed neural network, which means they are another variety of EANN (Zamparbelli, 1997).

2.2 EANNs for Autonomous Mobile Agents

As mentioned, autonomous agents are a leading research area in computer science and robotics. However, these systems are inconvenient to program; it is difficult to predict the problems that an agent will encounter when attempting to perform a task in the real world. Unpredictable factors can lead to undesired emergent behaviour. Sensor noise and echoes can greatly affect how a robot perceives its environment. The way light casts a shadow on a wall can affect how a visualization system interprets a corner. It is for this reason that most research into autonomous mobile agents today involves evolved artificial neural networks. Using EANNs, autonomous agents can be developed and tested in a simulated environment and then exported to a non-virtual robot. Then they can be tested again in the real world before being deployed.

Many modern autonomous agent systems have one of two different types of EANNs at their cores: either a WEA or a TEA. Most of today's autonomous systems are also initially developed in a simulated environment before being deployed in the real world.

This section will discuss some of the implemented EANN systems and contrast the problems inherent in simulated and real-world training environments. Finally, this section will review some of the current research involving the application of EANN systems to autonomous mobile agents.

2.2.1 Artificial Neural Network Configurations

The two most commonly used types of EANNs are WEAs (Floreano & Mondada, 1998; Lee, 2003; Miglino, Lund, & Nolfi, 1995; Mondada & Floreano, 1995) and TEAs (Nelson, Grant, Galeotti, & Rhody, 2004; Ward, Zelinsky, & McKerrow, Learning to Avoid Objects and Dock with a Mobile Robot, 1999; Xu, Van Brussel, Nuttin, & Moreas, 2003). These are also the most common types used for the evolution of autonomous agents. Typically if a robotic agent is small and simple it will be controlled by a basic ANN that is only modified through a WEA. A basic ANN is appropriate for such a problem because small robots generally have limited processing capacities and would not be able to handle the processing required by a more complex ANN (Floreano & Mondada, 1998; Miglino, Lund, & Nolfi, 1995; Mondada & Floreano, 1995). There are also robotic systems implemented with far more complex onboard processing systems which could easily handle an adaptive ANN structure outside of a simulated environment. However, typically the topologies of robotic systems are developed in a simulated environment before deployment into non-virtual robotic systems (Nelson, Grant, Galeotti, & Rhody, 2004; Ward, Zelinsky, & McKerrow, Learning to Avoid Objects and Dock with a Mobile Robot, 1999; Xu, Van Brussel, Nuttin, & Moreas, 2003).

2.2.2 Simulated Environments and Real-World Environments

There are two ways to develop and train an EANN. One is to develop the entire system, including both the autonomous agents and their training environment, in a software simulation. Simulated environments are used to evolve most autonomous agent EANN systems because simulated environments are not limited by the constraints of real-world time. A full EANN training simulation and then a full generational cycle can take mere minutes to complete on a sufficiently powerful computer. The second way to develop and train an EANN involves running a similar generational cycle on a computer, then transferring the agents to their robotic bodies, performing the training cycle and finally transferring the agents' control systems back to the generational system. This method could take more than an hour to complete one generational training cycle. Because a true EANN system typically requires several hundred generations to sufficiently evolve, the length of time required to perform each training cycle is extremely important.

Whether an autonomous agent system can effectively be trained in a simulated environment and then deployed in a real-world environment without needing to be retrained in the real world is a matter of much debate. Some systems that are evolved in simulation are subsequently implemented in real-world hardware to prove that the resulting system is realistically functional (Floreano & Mondada, 1998; Miglino, Lund, & Nolfi, 1995; Mondada & Floreano, 1995; Nelson, Grant, Galeotti, & Rhody, 2004; Ward, Zelinsky, & McKerrow, Learning to Avoid

Objects and Dock with a Mobile Robot, 1999; Xu, Van Brussel, Nuttin, & Moreas, 2003). It is effectively impossible for a virtual environment to simulate all of the subtle details and variations of a real-world environment. However, programmers creating a simulation can partially compensate for unpredictable real-world environmental factors by adding noise to the simulated sensors. They can also perform tests on non-virtual system sensors to see how they behave in the real world, and then incorporate their results into their simulations. (Miglino, Lund, & Nolfi, 1995). When the simulated autonomous agents are transferred to their physical robotic systems, a few more training cycles are performed to adapt the networks to their new sensor inputs. It has been shown that only a few more training cycles are required for the ANNs to adapt and begin behaving as they did in the simulation (Floreano & Mondada, 1998). The major learning was already done in the simulation and they only needed to adapt to the changes in their sensory input (Floreano & Mondada, 1998; Mondada & Floreano, 1995).

In some cases, it is absolutely necessary to train an EANN in a simulated environment instead of in the real world. One existing EANN system is a set of large and powerful robots designed to move palettes of products around in a warehouse. If this system had been completely trained in its real-world environment, several of these inordinately expensive machines would have been required to undergo generational learning and the systems that were poorly adapted to the environment could have destroyed themselves, other autonomous agents or large portions of the building (Xu, Van Brussel, Nuttin, & Moreas, 2003). This shows that simulated environments are necessary for training some types of autonomous agents; if simulated environments were unavailable then certain problems might never be solved.

Although EANNs can be trained in either the real world or a simulated world, the two types of training can complement each other and some real-world problems would not be solved with EANNs if both training options were not available.

2.2.3 Obstacle Avoidance

The problem of obstacle avoidance with respect to EANNs and autonomous agents is not frequently studied. Most research focuses more on goal finding and path finding than on obstacle avoidance. This may be because robots evolved using EANNs learn to avoid hitting walls as a part of their training (Floreano & Mondada, 1996; Floreano & Mondada, 1994).

Other papers directly focus on dealing with obstacle avoidance (Kluge, Kohler, & Prassler, Fast and Robust Tracking of Multiple Moving Objects with a Laser Range Finder., 2001; Kluge, Bank, & Prassler, Motion Coordination in Dynamic Environments: Reaching a Moving Goal while Avoiding Moving Obstacles., 2002). Since obstacle avoidance is at least somewhat important for all EANNs that handle autonomous agents, it is essential to closely examine exactly what is meant by "obstacle avoidance" in this context and to survey the various types of obstacle avoidance systems.

In this context, "obstacle avoidance" means "to avoid a collision with an object that is blocking the path of a planned direction of motion". There are two basic kinds of blocking objects, or obstacles: static and dynamic. Because static obstacles do not move they are relatively easy for an autonomous agent to avoid. Dynamic obstacles are more complex because the agent cannot predict with certainty where the obstacle will be in the next time frame. The motivation for dealing with dynamic obstacles comes from observing that humans easily avoid each other in crowded environments (Kluge, Illmann, & Prassler, Situation Assessment in Crowded Public Environments., 2001).

2.2.3.1 Static obstacles

As mentioned, static obstacle avoidance is an easier problem to solve than dynamic obstacle avoidance. Many implemented systems attempt to move around obstacles in their paths (Floreano & Mondada, 1994; Floreano & Mondada, 1996). Others simply halt and wait for obstacles to get out of the way (Kluge, Kohler, & Prassler, Fast and Robust Tracking of Multiple Moving Objects with a Laser Range Finder., 2001; Kluge, Bank, & Prassler, Motion Coordination in Dynamic Environments: Reaching a Moving Goal while Avoiding Moving Obstacles., 2002).

Some obstacle avoidance systems are designed to move toward a goal or follow another moving object or agent (Kluge, Bank, & Prassler, Motion Coordination in Dynamic Environments: Reaching a Moving Goal while Avoiding Moving Obstacles., 2002; Neruda, 2007). While following is a complex task to train an agent to do, it is easier for an agent to avoid obstacles when following because the obstacle avoidance tasks are passed on to the agent or other object being followed. In research, static object avoidance is not heavily studied, presumably because it is considered to be a consequence of agents learning to do their other tasks.

2.2.3.2 Dynamic obstacles

Dynamic obstacle avoidance is more complex than static obstacle avoidance. For an agent to avoid dynamic obstacles, it must be able to predict where an obstacle may be next; an agent must use strategic planning to avoid dynamic obstacles.

Much of the work that has involved agents learning to follow has contributed to the search for a solution to the dynamic obstacle avoidance problem. This is because agents that can follow must be capable of tracking a dynamic object, and for an agent to avoid a dynamic obstacle, it must be able to track it. (Floreano & Mondada, 1994; Floreano & Mondada, 1996).

Other research that is relevant to this project focuses on detecting moving obstacles (Kluge, Kohler, & Prassler, Fast and Robust Tracking of Multiple Moving Objects with a Laser Range Finder., 2001). A variation on this theme directly focuses on a situation in which the dynamic obstacles are people (Scheutz, Cserey, & McRaven, 2004). Such research is critically important to the goal of implementing an autonomous agent in a real-world environment. One research project used a very complex mathematical approach to enabling agents to avoid dynamic obstacles while proceeding towards a moving goal (Kluge, Bank, & Prassler, Motion Coordination in Dynamic Environments: Reaching a Moving Goal while Avoiding Moving Obstacles., 2002). The agents described in that paper used a system that tracked and predicted where objects were going to be in the next timeframe while planning their motion towards their moving goals. This was a very complex system but it worked well.

While several systems exist to track and/or avoid static and dynamic obstacles, few of these systems use an agent controller that was developed using an evolutionary approach (Neruda, 2007). Only one of these systems is specifically designed to deal with the problem of dynamic obstacle avoidance, and the agents of that system do not operate in a busy environment (Kluge, Bank, & Prassler, Motion Coordination in Dynamic Environments: Reaching a Moving Goal while Avoiding Moving Obstacles., 2002).

2.3 EANNs as Autonomous Agent Controllers in Dynamic Environments

Several of the required components for solving the problem of autonomous agents seeking goals in dynamic environments have already been researched and proven in the research projects surveyed here. Neural networks can be trained in a simulated system. There are effective ways to negate the problematic perfection of information provided by simulated sensors. Neural network topologies can be evolved using genetic algorithms although the process may require extension to include more complex topological structure. Even extremely convoluted problem spaces can be searched relatively effectively using genetic algorithms. It is possible to create a neural network that can handle obstacle avoidance. It is also possible to create an autonomous agent that is capable of avoiding dynamic obstacles while tracking a dynamic goal. In addition since multilayer neural networks are provably equivalent to Turing machines (Cybenko, 1989) it is likely that multilayer neural networks can be used to solve the problem of autonomous agent dynamic obstacle avoidance.

Neural networks can be trained to control robots. However, the networks that are required to control the robots in an unpredictable and dynamic environment have not yet been developed and may be difficult to create. Dynamic obstacle avoidance has not been extensively studied in robotics. Robots have been created that can avoid obstacles, but their ability to avoid moving obstacles is limited or nonexistent (Neruda, 2007). This may be because it is a difficult problem to solve.

Many ANN systems have been effectively developed by EANN systems. Some ANN systems are used in real-world environments (Sharkey, 1997). One robotic system using ANNs functions effectively in a static real-world environment (Ward, Zelinsky, & McKerrow, Learning to Avoid Objects and Dock with a

Mobile Robot, 1999). This suggests that it may be possible to use EANNs to evolve autonomous agents that can reach a goal in a dynamic environment. However at present there has been very little research about this particular type of robotic obstacle avoidance.

2.4 Conclusions

Despite the lack of practical research in this particular field of robotics and EANN development, the literature suggests that it might be possible to combine EANNs and simulated environments to evolve agents that can reach a goal in a dynamic environment. It is critical that the machine learning technique both evolves an efficient topology and works towards this solution efficiently. If the EANN does not use an efficient approach, producing an evolved solution may take an excessive amount of time. In order for an EANN to produce a set of solutions it must be run several times, and each run can take days to complete. If the algorithm does not produce efficient solutions, the necessary computational time can increase dramatically. Therefore it is essential that any approach using EANNs must be efficient. This dissertation describes an attempt to develop an efficient approach to solving the dynamic obstacle avoidance problem.

Chapter 3

METHODS AND APPROACHES

This chapter describes the system developed for this project. The project uses a neuroevolutionary system that develops an initial set of agents. This population of agents is trained on a benchmark set of scenarios designed to teach the agents basic static and dynamic obstacle avoidance. The trained set of agents is then used to evolve a new generation of agents, which is placed in a simulated environment and evaluated. This process is repeated until the population of agents has reached a satisfactory level of performance without improvement or has completed a predetermined number of epochs. The project employs genetic algorithms using the well-known neuroevolutionary method NEAT to create a new generation of autonomous agents. The performance of this algorithm will then be compared and contrasted against Segmental Duplication NEAT, the new neuroevolutionary algorithm this thesis introduces.

This chapter begins with a discussion of NEAT and SDNEAT, the algorithms which will be implemented in the Neuroevolutionary System. It also reviews the construction of this system using the components SIMBAD, PicoEvo and PicoNeuro to form an EANN-based autonomous agent simulation system which supports the experiments described in Chapter 4.

3.1 Algorithms

Because the problem of dynamic obstacle avoidance in dynamic environments is so complex, a new neuroevolutionary algorithm designed to allow evolution of more complex solutions in a shorter time span was developed for this project. The new strategies introduced in this innovative neuroevolutionary algorithm do not affect the existing genomes' learned behaviour and offer the possibility that more complex behaviours may arise quickly from low levels of complexity. This new algorithm is called Segmental Duplication NEAT (SDNEAT). It is predominantly based on the methods that NEAT employs but introduces a new mutation method called segmental duplication. This method is based on the process of segmental duplication in biological life forms. Segmental duplications can be an advantage to evolution in biological life forms by facilitating high amounts of mutation and innovation while maintaining a low probability that the existing genome will be completely disabled (Bailey & Eichler, 2006).

Testing the performance of the new SDNEAT algorithm will entail comparing its performance with the original NEAT algorithm developed by Kenneth O. Stanley and Risto Miikulainen (Stanley & Miikkulainen, Evolving Neural Networks through Augmenting Topologies, 2002). The following section discusses NEAT as it is described in Stanley's dissertation (Stanley, Efficient Evolution of Neural Networks through Complexification, 2004) and how it works as a genetic algorithm. It then reviews how SDNEAT extends the existing NEAT algorithm.

3.2 NeuroEvolution of Augmenting Topologies (NEAT)

NEAT is an efficient system for evolving artificial neural networks in a genetic algorithm. Partly because like all HEAs it modifies both structure and weights, NEAT can evolve extremely complex minimalist solutions to a variety of problems (Stanley, Efficient Evolution of Neural Networks through Complexification, 2004). NEAT is also one of the few neuroevolutionary systems that can perform evolution using both mutation and crossover operators. It is not always obvious how to perform a crossover operation in a neural network because the structures of different neural networks are not necessarily related.

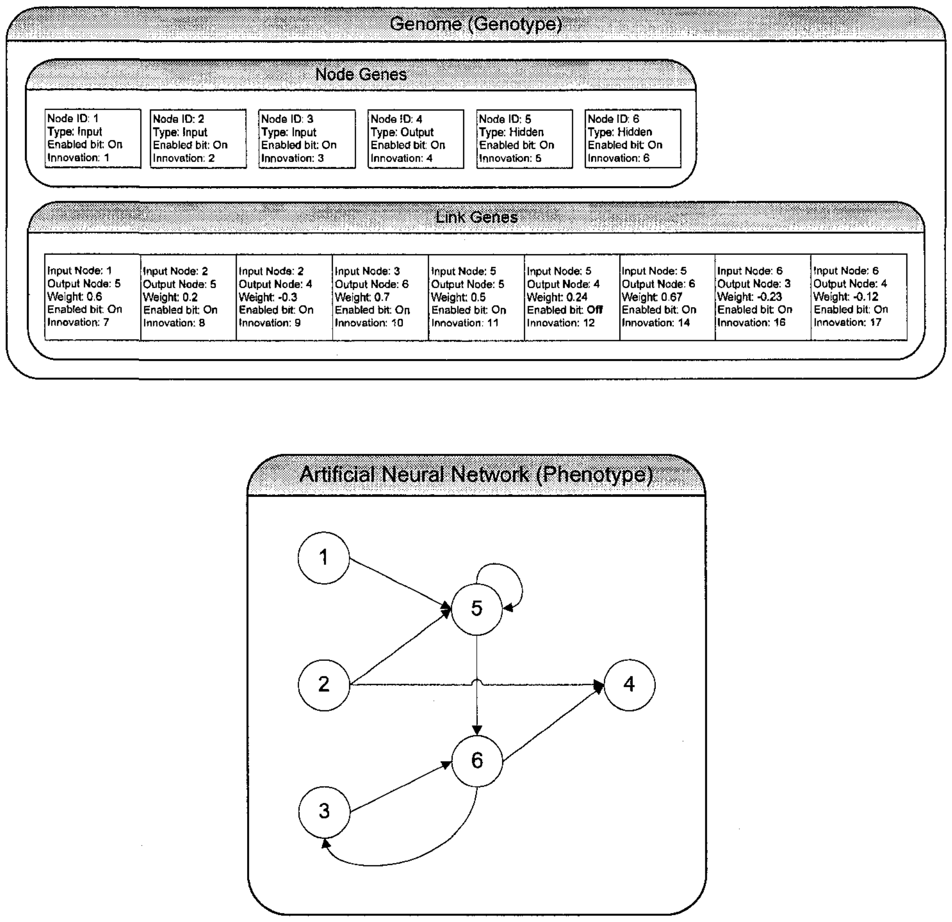
Therefore crossover operators have to perform complex analysis of the network structure to find appropriate points for crossover of the neural networks or phenomes. This problem is compounded by the fact that the genomic representation of the phenome does not clearly indicate where crossover can and cannot occur.

NEAT solves this problem by using historical markings in each element of each genome. Each node and link in NEAT added after the initial population of genomes is created has a historical marking attached to it. This guarantees that new innovations in structure are identifiable and recorded. This is just one of the ways in which NEAT is an innovative neuroevolutionary algorithm.

3.2.1 Genetic Encoding for NEAT

This section explains how genetic encoding works in NEAT. The genetic encoding for NEAT is slightly more complex than for some other neuroevolutionary algorithms simply because a NEAT gene encodes more information.

Each genome in NEAT contains a list of links and a list of nodes. Each link and node in these lists is referred to as a gene of the genome, and a genome comprises the entire set of hereditary information for an individual. Each link contains values for its input node and output node, the connection weight, information about whether the link is enabled or disabled and an innovation number. The innovation number serves as the link's historical marking, denoting hereditary information about the gene. The innovation number allows the crossover algorithm to detect if its gene is similar to another innovation in a different genome. A node contains slightly less information than a link, including a unique node identification number, an activation response value, a disable bit variable and an innovation number. When the genome is converted to its neural network manifestation, it is referred to as a phenome. A phenome is the virtually-physical interpretation of the genotypes of the genome. Proper phenome structure must be conserved stringently during mutation and crossover operations. The link and node connectivity information comprise the physical characteristic, or genotype, information of the genome. When the genotype information is expressed physically, the observable characteristics are called the phenotypes.



Genome (Genotype)

Node Genes

Node ID: 1
Type: Input
Enabled bit: On
Innovation: 1

Node ID: 2
Type: Input
Enabled bit: On
Innovation: 2

Node ID: 3
Type: Input
Enabled bit: On
Innovation: 3

Node ID: 4
Type: Output
Enabled bit: On
Innovation: 4

Node ID: 5
Type: Hidden
Enabled bit: On
Innovation: 5

Node ID: 6
Type: Hidden
Enabled bit: On
Innovation: 6

Link Genes

Input Node: 1
Output Node: 5
Weight: 0.6
Enabled bit: On
Innovation: 7

Input Node: 2
Output Node: 5
Weight: 0.2
Enabled bit: On
Innovation: 8

Input Node: 2
Output Node: 4
Weight: -0.3
Enabled bit: On
Innovation: 9

Input Node: 3
Output Node: 6
Weight: 0.7
Enabled bit: On
Innovation: 10

Input Node: 5
Output Node: 5
Weight: 0.5
Enabled bit: On
Innovation: 11

Input Node: 5
Output Node: 4
Weight: 0.24
Enabled bit: **Off**
Innovation: 12

Input Node: 5
Output Node: 6
Weight: 0.67
Enabled bit: On
Innovation: 14

Input Node: 6
Output Node: 3
Weight: -0.23
Enabled bit: On
Innovation: 16

Input Node: 6
Output Node: 4
Weight: -0.12
Enabled bit: On
Innovation: 17

Figure 3.1: Genotype and Phenotype Example for the NEAT Algorithm. Above is an example of a genotype that represents the displayed phenotype. There are six nodes: three input, two hidden and one output. There are nine links, two of which are recurrent and one of which is disabled. The disabled gene (connecting nodes 5 and 4) is not displayed.

3.2.2 NEAT mutation operations

Mutation operations in NEAT can change connection weights, node activation values and network topology. All these mutations occur randomly, constrained by a fixed probability which is defined for each individual simulation. Weight and activation value mutations occur when a random node or link is chosen and its weight is perturbed by a constrained random value. NEAT mutation operations change network topology by adding links and nodes. When the GA mutates a link, it randomly chooses two nodes and inserts a new link gene with an initial weight of one. If a link already existed between the chosen nodes but was disabled, the GA re-enables it. Finally if there is no link between the chosen nodes and an equivalent link has already been created by another genome in this population this link is created with the same innovation number as the previously created link as it is not a newly emergent innovation. A node mutation is similar to a link mutation but differs from it in that instead of choosing two nodes and inserting a link, the GA chooses and disables an existing link and inserts a node. The GA inserts this new node with a random activation value, as well as two link genes to connect the node to the now-disabled link's previous input and output nodes. The GA then transfers the weight from the disabled link gene to the new link gene, which is connected to the old output neuron. The weight of the link gene inserted between the new neuron and the old input node is set to one so as not to disturb any learning that has already occurred in this connection. Introducing a new node where a link once existed may fragment some evolved knowledge in the phenome. Copying the original link weight to one of the new node's links while setting the other connecting link weight to one minimizes the disturbance in learning.

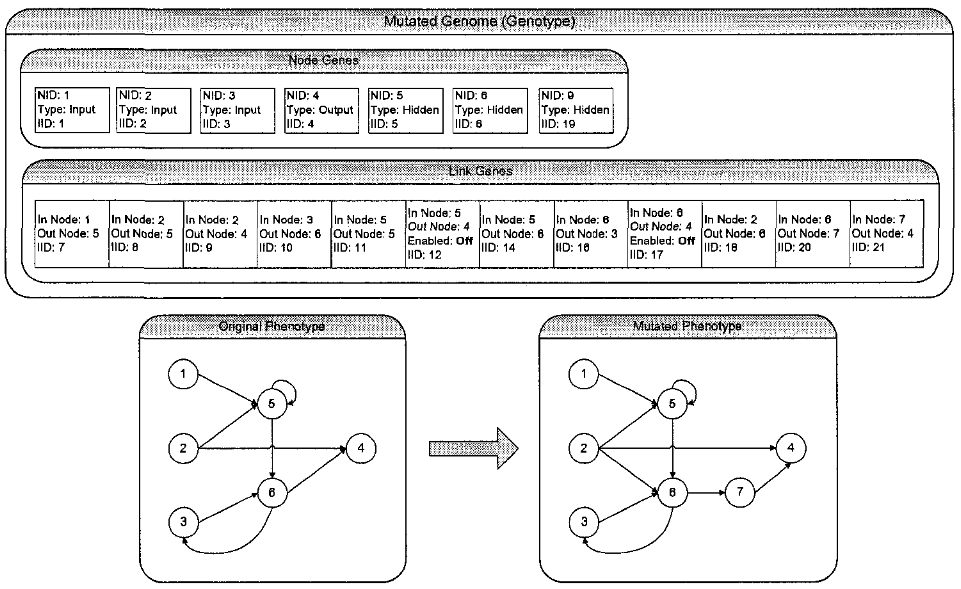

Mutated Genome (Genotype)
Node Genes
NID: 1 Type: Input IID: 1
NID: 2 Type: Input IID: 2
NID: 3 Type: Input IID: 3
NID: 4 Type: Output IID: 4
NID: 5 Type: Hidden IID: 5
NID: 6 Type: Hidden IID: 6
NID: 9 Type: Hidden IID: 19
Link Genes
In Node: 1 Out Node: 5 IID: 7
In Node: 2 Out Node: 5 IID: 8
In Node: 2 Out Node: 4 IID: 9
In Node: 3 Out Node: 6 IID: 10
In Node: 5 Out Node: 5 IID: 11
In Node: 5 Out Node: 4 Enabled: Off IID: 12
In Node: 5 Out Node: 6 IID: 14
In Node: 6 Out Node: 3 IID: 16
In Node: 6 Out Node: 4 Enabled: Off IID: 17
In Node: 2 Out Node: 6 IID: 18
In Node: 6 Out Node: 7 IID: 20
In Node: 7 Out Node: 4 IID: 21
Original Phenotype
Mutated Phenotype

Figure 3.2: Mutation Example: A mutated form of the original genome displayed in figure 3.1 is shown here. Both a link and a node mutation have occurred to create a new phenotype. A link genome has been added to connect node 2 and 6. During the node mutation the link between node 6 and 4 was disabled and a new node (node 7) was added. In the definition of the link genes, the values once contained in the now-disabled link have been added to the link between nodes 7 and 4 and the link between nodes 6 and 7 has been set to a value of one to preserve the original learned value.

These mutation functions introduce complexity into the initial population, referred to as *base genomes*, and gradually grow a solution to the given fitness function. Since the processes are pseudo random a diverse population of genomes will evolve. The crossover function must be able to recombine these inherently different topologies efficiently. It does this using historical markings also known as innovation numbers.

Because the innovation numbers are unique to innovations and not to genes, it is possible to compare any two genomes in the population and determine which genes they share. If two genes share the same innovation number they also share the same manifestation, or *phenotype*. Innovations are preserved among genomes in the same population. In order for mutation and crossover to function in NEAT, the system must maintain a database of all the innovations that have occurred since the first generation of each simulation. When a new innovation occurs it is checked against the database of innovations to ensure that it is not identical to an existing innovation. If its originality is confirmed, a global innovation number is incremented and assigned to it and it is recorded in the innovation database. This guarantees that while each genome might have a different structure with different weights, all related genes are identical. When a crossover operator is applied to two genomes the offspring inherits the same innovation number in each gene. This preserves the historical markings through generations. This preservation of historical markings prevents the crossover operation from becoming too computationally intensive and the networks from exploding in size because of crossover.

3.2.3 Crossover in NEAT

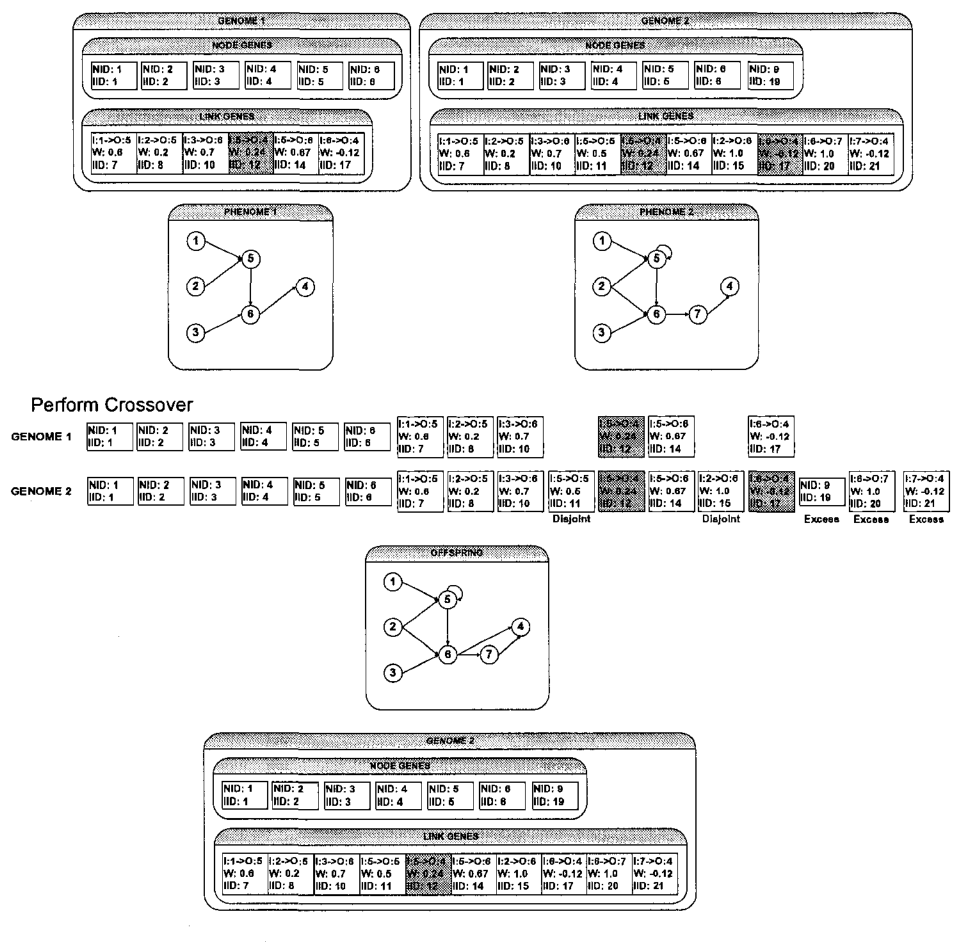


Figure 3.3 Crossover Operation: Although the parents are structurally different, their innovation numbers show that they are very closely related. Crossover happens easily without requiring structural analysis.

During a crossover operation, NEAT can quickly determine how to line up the two parents' genes. Once they are aligned it is easy to see which portions are similar and which are different. Any genes that do not share innovation numbers with genes in the other parent's genome are referred to as *disjoint* and are added to the child during crossover. If either parent has genes that are newer than any of the genes in the other parent, they are considered excess and are also added to the child during crossover. All genes that are shared by both parents are inherited by the child from the parent with the highest fitness. A gene that is disabled in one parent but enabled the other has a chance of being re-enabled in the offspring.

This method of crossover allows NEAT to build increasingly complex ANN structures without restricting compatibility between genomes. Unfortunately this level of complexity works against the genetic algorithm as it cannot support such diversity in its population. A structural innovation that could exceptionally improve performance in a later generation may introduce a major change in a given genome, but since it requires a few generations to reach its full potential it may be erased from the population before it has a chance to affect performance. This is why NEAT employs speciation: to protect genomic innovation.

3.2.4 Speciation

In natural evolution entities that once shared a common genome sometimes diverge so much that they can no longer mate with one another. This divergence is known as speciation. In NEAT, as the genomes in a population grow complexity a new innovation in their topology may result in greater performance for the population's agents. NEAT uses speciation to protect such innovations. When an agent's structure diverges far enough from that of the other agents in the population NEAT identifies it and places it in its own species. Using innovation numbers NEAT can calculate the distance between two genomes. The distance is defined by the following function:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \bar{W}$$

Figure 3.4 Distance Between Genomes: The distance δ between two genomes is the sum of the number of excess (*E*) and disjoint (*D*) genes, and the average of the weight differences of the two genomes ($\bar{W}$). The coefficients c_1, c_2, and c_3 modify the weights of each of the variables and N is the number of genes in the larger of the two genomes.

Initially one species is formed from the entire first generation. The first genome in the generation becomes the champion of that species since the population is uniform. As the algorithm proceeds and more complexity is introduced distances between genomes will increase until they are larger than the distance threshold. At this point, NEAT designates this structurally different genome a new species and names it as the species champion. As other genomes' distances from their species champion increases, they may be placed in a different existing species if their distance from that species champion becomes small enough.

NEAT maintains species through generations to protect innovation and as an evaluation method for the effectiveness of an innovation. If no members of a species rise above their existing champion in fitness for a set number of generations, the entire species is terminated, unless its champion is the population champion.

To determine the number of genomes each species can introduce into the next generation, NEAT uses explicit fitness sharing. Each species is assigned a certain number of reproduction spots based on the sum of the species' adjusted fitness values. Each genome's adjusted fitness score is based on its distance from every other genome in the population. The lowest-performing fraction of each species does not reproduce, and the highest performer from each species carries over to the next generation via per-species elitism. Any remaining reproduction spots are filled through random selection.

If a species becomes too large, its genomes cannot reproduce productively because they do not have enough reproduction spots in the next genome. This keeps the species' sizes reasonable and is necessary for speciation-based evolution systems. If species size were not restricted one species could grow to dominate the entire population and the benefit of speciation would be lost. Most genomes in a species are structurally similar because new structural innovations are slowly added to the phenotypes, reducing the generation by generation structural variation. Hence speciation protects innovation.

The NEAT algorithm is a robust EANN. It uses speciation to protect innovation, and it uses innovation numbers to perform all GA operations efficiently. The efficiency of the GA operators helps NEAT limit increasing complexity. This combination allows NEAT to search a broad solution space efficiently while minimizing the complexity of its solutions.

3.3 Segmental Duplication NEAT (SDNEAT)

Segmental Duplication NEAT is based on NEAT and inspired by recent research of the human genome. This recent research claims to show that large segments of the human genome that are purely duplicate genetic information may be critical requirements for the advancement of the species (Bailey & Eichler, 2006).

Nearly 14% of the human genome consists of segmental duplications. In comparison, the mouse genome is approximately 7% segmental duplications and the chimpanzee genome is only about 5% segmental duplications (Bailey & Eichler, 2006). These segmental duplications also appear to be at least somewhat non-random. Segmental duplications show a higher rate of copy variation, or mutation. They appear to be favoured in gene selection and several functional categories that are realized in human beings appear to be enriched by segmental duplications (Bailey & Eichler, 2006). One example of this enrichment is the human immune system (Bailey & Eichler, 2006). The high percentage of segmental duplications in the human genome seems to imply that they are key to faster innovation through genetic processes. They protect the genome from harmful mutations because they are duplicates, and most mutations to them will not affect the existing genomic functionality. The human male gender chromosome (the 'Y' chromosome), shows a very high amount of segmental duplications; approximately 50% of its genes are segmental duplications. This may imply that segmental duplications prevent genetic stagnation in the male of the species; the 'Y' chromosome routinely undergoes mutation (Bailey & Eichler, 2006). This amount of mutation is required as the 'Y' chromosome never performs crossover with other chromosomes.

All these reasons support the development of a new version of the NEAT algorithm. Segmental Duplication NEAT (SDNEAT) is be based on the NEAT algorithm but includes a new mutation operator which will identify a segment of genetic information, duplicate that segment, heavily mutate it, and integrate it back into the genome. This duplicated segment may offer an evolutionary leap, and cause the algorithm to find new solutions to the problem. Using SDNEAT, innovations are still protected by speciation so all the advantages of the NEAT algorithm are preserved. It is important to note that NEAT would be capable of evolving any solution SDNEAT can evolve. However the chances of NEAT evolving exactly the same segmental duplication are quite low as it would require multiple new node innovations in a particular sequence.

3.3.1 A Segment

In order to develop a segmental duplication operation a segment must first be defined, because the concept of segments does not exist in the original NEAT algorithm.

Definition: A segment is an array of n nodes and m links:

$$S_n : (n > 0)$$

$$L_m : (m > 1)$$

S contains only hidden nodes.

$$S_1^x : S_x \ is \ not \ an \ input \ or \ output \ node.$$

The segment is connected to both an input and output node.

$$L_0 : arrives \ from \ an \ input \ node$$

$$L_m : connects \ to \ an \ output \ node$$

The segment is not recurrent.

$$L_1^{m-1} : L_x \ connects \ S_n \ to \ S_{n+1}$$

This definition states that all segments for SDNEAT begin at an input node, end at an output node and must contain at least one hidden node to a maximum of *n* hidden nodes. There are no recurrent or loopback connections in a valid segment. Some of the problems of identifying a valid segment algorithmically are eliminated by limiting a valid segment to this subset. The portion of the algorithm that identifies a valid segment need only walk a path through the neural network from an input node to an output node. It can ignore recurrent connections along the way.

3.3.2 Segmental Duplication

Because identification of a segment is simple and the beginning and endpoint of a segment is limited to an input node and an output node, inserting the duplicated segment is also straightforward. The identified segment is already a valid path in the neural network so duplicating it and inserting it between the same input and output nodes does not destroy the genome, but it does modify a substantial portion of the genome's genetic code. This enhanced rate of growth does not significantly increase complexity as it relies on the original NEAT methods for topological growth and cannot evolve any structure that NEAT could not. It simply causes generational leaps to happen faster. In fact, no segmental duplications can occur without original NEAT node mutations. The initial genomes contain only input and output nodes and because, by definition, a segment cannot contain an input and output node.

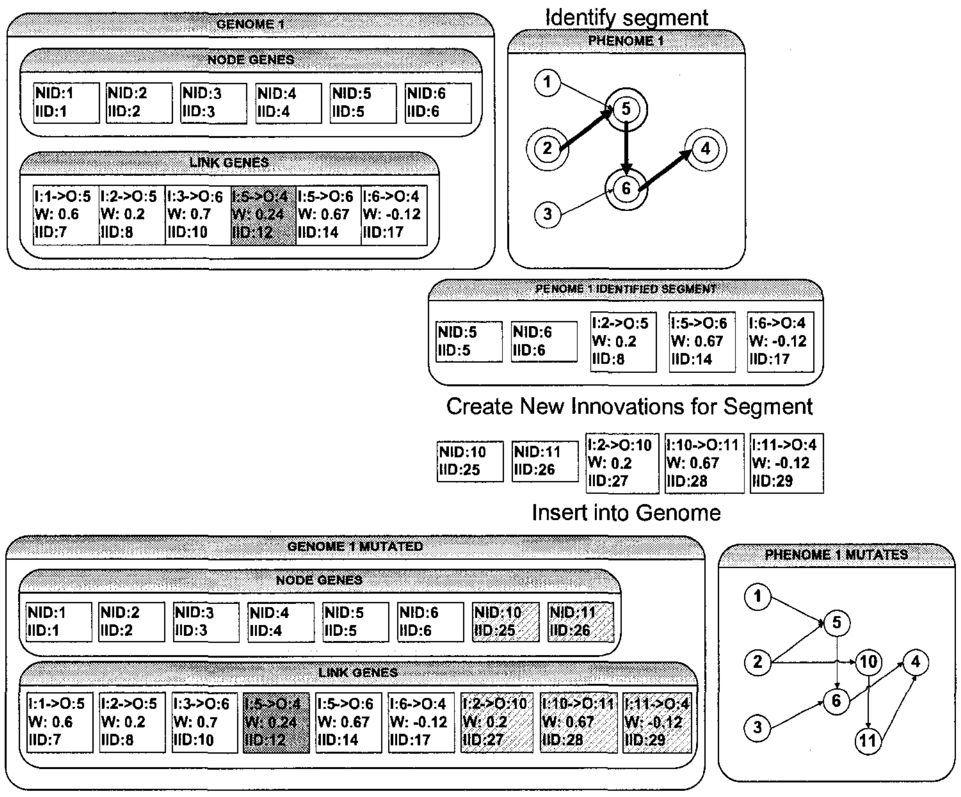


Figure 3.5 Segmental Duplication. In Genome One at the top of this diagram, a segment has been identified. The nodes with solid circles are added to S_n and the highlighted links connecting the nodes with broken circles surrounding them are added to L_m. This segment is then duplicated and new innovation numbers are created for all the components, the node identifiers are properly incremented and the links are adjusted to connect to the new nodes. These nodes and links are then appended to Genome One and the new phenome is displayed at bottom right.

The historical markings or innovation numbers are an important aspect of NEAT. When SDNEAT inserts a new segmental duplication, it is creating a copy of active genes. In SDNEAT, all segmental duplications are treated as new innovations. In the original NEAT algorithm, if a node mutation occurs which disables a link, then later that link is re-enabled and an equivalent node mutation occurs on the same link the innovation list identifies this as an old innovation.

Although the innovation list has identified it as an old innovation, NEAT considers it to be a new innovation and has the innovation list assign it a new innovation number. This is the base case of a segmental duplication, and consequently all segmental duplications are treated as new innovations.

To identify a segment, the SDNEAT algorithm first randomly selects an input node from the genomes set of input nodes. Then the algorithm attempts to find a path to an output node, randomly chooses an output link from its current node and, if the output link is not recurrent, the algorithm follows that link to the next node and repeats the previous steps. Each time the algorithm steps to a new node it copies the link and node to its arrays for duplication. If a step from an input node arrives at an output node of the neural network, a new input node is randomly chosen and the algorithm starts again, as by definition a segmental duplication cannot consist of one link. If the algorithm finds an output node, it has found a valid segment. The algorithm duplicates the valid segment by creating new innovations for each link and node in the segment's link and node arrays. The weights from the original nodes and links are duplicated but the innovation numbers are updated. The segment's weights are then mutated at a higher than average mutation rate. Once mutation of the segment is complete the new links and nodes are appended to the genome being mutated.

SDNEAT maintains the efficiencies and capabilities of NEAT, including all operations and speciation, but it introduces a new operator: the segmental duplication mutation. This new operator can drastically mutate an existing genome without affecting the capabilities of the existing NEAT algorithm. This drastic mutation has the potential to broaden the search area of NEAT to include elements that would not otherwise be searched for several generations. This mimics the genetic behaviour recently identified in the human genome. In order to evaluate this new algorithm and its ability to evolve an efficient neural network controller that can learn to effectively operate an autonomous agent in multiple different dynamic environments, a neuroevolutionary simulation system must be built.

3.4 Neuroevolutionary Solver

There are several systems available that use the NEAT algorithm to solve various problems. There are also various robot simulation packages. Systems that combine NEAT with robot simulation environments appear to not exist or are scarce. A system that combines these components and is flexible enough to support various simulation platforms and the addition of the SDNEAT algorithm had to be developed for this project. A combination of available open source simulation software and NEAT demonstration code is used to develop a neuroevolutionary solver (NS) with the described requirements. This section will review the various software packages and the modifications made to them to form the simulation system.

3.4.1 Requirements

The NS is a large and complex system, but as mentioned, some of the components have already been developed, which can decrease development time for this project. It is important to have an effective simulation system that works on a time slice basis, meaning that each instant of computation is one frame of animation. A system that works in this way is effective for robot simulation as each agent is given time to analyze its environment in simulated real time. When an evolved system is removed from the simulated environment and deployed in the real world, it no longer learns and its computational requirements decrease. This allows its reaction time to increase; if the agent were required to learn in the real world, real time behaviour would not be possible. This is the benefit of using a time sliced simulated environment.

The simulation system must also be easy to integrate with a neural network package and an evolutionary algorithm package. The methods behind both neural networks and evolutionary algorithms are well known and software packages that implement them are common. An open source solution is favourable, as the evolutionary algorithm package must be combined with the NEAT and SDNEAT algorithms.

Effective visualisation of both the simulation and the neural network components is also necessary. Visual inspection of evolution as it is occurring and the ability to review agents and neural networks after they have been evolved is essential to evaluation of the performance of the system. It is also necessary to record statistics about each evolutionary experiment.

3.4.2 Chosen components

The SIMBAD robot simulation system developed by Louis Hughes and Nicolas Bredeche (Hughes & Bredeche, 2007) was chosen to act as the core of the NS. SIMBAD is a Java 3D-based robot simulator. It is an open source system and was designed for research and learning so some of the requirements listed above are integrated into it. SIMBAD is a time sliced system. Each frame of simulation is a distinct computational time slice. All components of the simulation that require computational time share the computational pipeline; it is not a multi-threaded system. If it were, there would be more unpredictable behaviour with respect to simulated computation.

The SIMBAD system allows users to quickly develop their own test environments and robotic agent control systems. Agents with various control systems are easily integrated into different environments. There is a variety of sensors and actuators available to the simulated agents. The system is even capable of simulating Khepera, a common hardware platform for evolutionary robotics experiments. SIMBAD provides a three-dimensional simulation environment for single and multiple agent simulations. It also provides a batch mode simulation environment designed for high throughput of simulations. Since the dynamic obstacle avoidance problem will require large amounts of simulation and large quantities of tests with several generations must be run to fully realize the capabilities of an EANN, the required computation time is enormous. The batch mode of SIMBAD will significantly decrease the computation time requirements.

Because the SIMBAD system was specifically built for machine learning and autonomous robot simulation, its developers recommend a neural network package and evolutionary algorithm package for EANN simulation.

The PicoEvo and PicoNeuro packages were developed by Nicolas Bredeche and they are designed to be integrated with the SIMBAD system. PicoEvo is a GA system that implements the standard GA algorithmic method discussed in Chapter One. It is a very robust and modular system. It was designed with future expansion in mind and it supports the use of static and dynamic arrays of values as genetic encoding. It does not support any encoding of neural network topology.

PicoNeuro is a complete neural network system. It supports several well-known network architectures including perceptrons, multi-layer perceptrons, feed-forward neural networks, backpropagation neural networks, recurrent neural networks, and self-organizing maps. It provides a visualization system for viewing the topology of neural networks as well as investigating the adjusted weights of both the nodes and links. Importantly, PicoNeuro supports recurrent neural networks; this is the type of network required by NEAT and consequently by the NS. PicoNeuro is also a modular system designed for expansion and it directly integrates with PicoEvo. PicoEvo does support using PicoNeuro for EANN research but the system is limited to a WEA type of EANN.

These three components combine to form an effective and extensible EANN system. Integration of the NEAT and SDNEAT algorithms into PicoEvo is not difficult as the system is easily extensible, but the amount of modification required is large. The modifications to PicoEvo and PicoNeuro elevate the system from a WEA to an HEA. The HEA-capable PicoEvo and PicoNeuro integrate with SIMBAD to complete the NS system.

3.4.3 High Level Overview

The core of the NS is comprised of two artificial intelligence systems and one simulation system. Applications of the NS are built on top of this core component. These applications include experiments and simulation playback systems. The two artificial intelligence components the simulation component and the NS applications are contained within the NS. The AI components are contained within the simulation system, SIMBAD. This defines a component hierarchy for the NS.

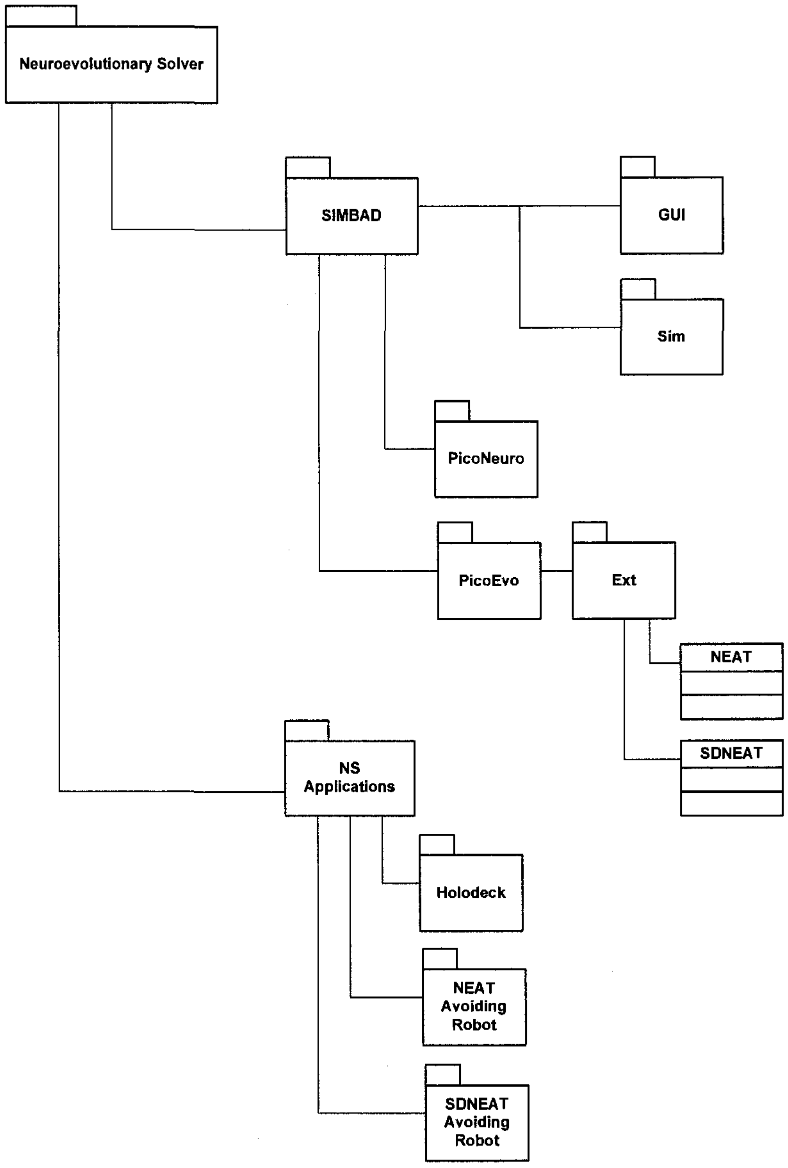


Figure 3.6: NS Component Structure: This is the hierarchical class structure of the Neuroevolutionary Solver. The simulation system, SIMBAD, is the highest level component of the core system. PicoNeuro and PicoEvo are integrated into it. NEAT and SDNEAT are implemented at the extension layer of PicoEvo. The simulation applications and the visualization system for simulation playback (Holodeck) are built at the NS application layer.

3.4.4 SIMBAD

The SIMBAD simulation system is the core component of the NS system. In order to function as an effective robotic simulation system SIMBAD implements several components:

- A graphical user interface (GUI), for visualization of autonomous robot and virtual environment simulations

- The simulator, which acts as the time slice simulation processor as well as handling agent computation, world computation and limited physics

- The batch processor, a component of the GUI which is separate from normal simulation.

The batch processor performs fast simulation with limited rendering and is required to complete EANN simulations in a reasonable amount of time. The GUI handles most of the visualization processing of the system. It also renders the onscreen controls and agent inspector displays. The GUI customizes itself to the simulation, displaying as many agent inspector displays as are required for a particular simulation.

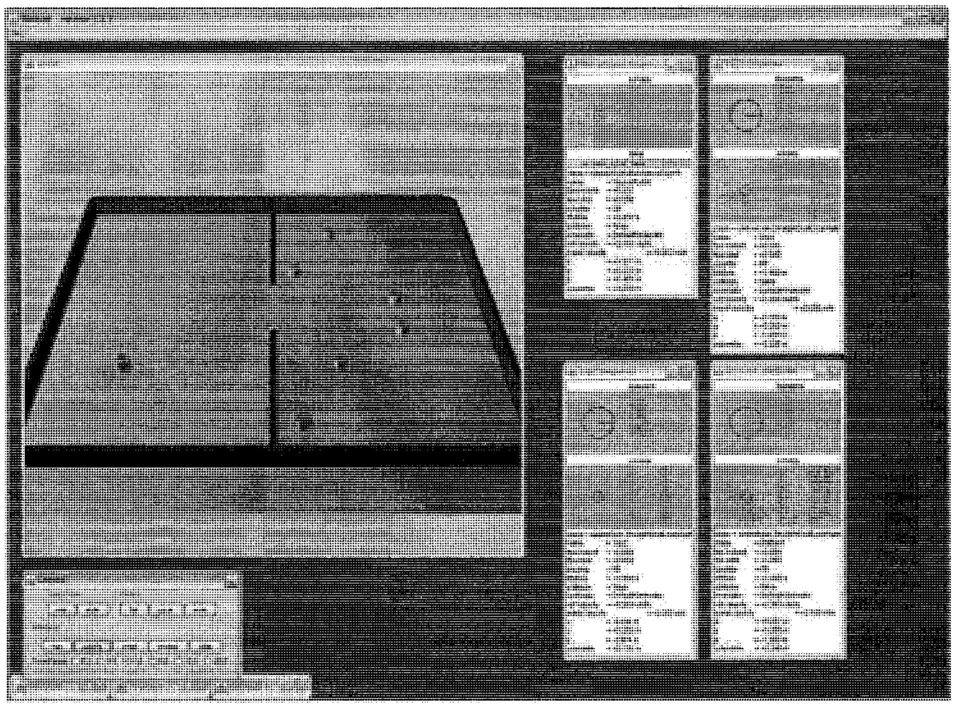


Figure 3.7 SIMBAD Graphical User Interface: In the SIMBAD GUI shown above, several agent inspector displays appear on the right. The main virtual world display is shown at the top left and the simulation controls are visible at the bottom left.

The SIMBAD simulator performs most of the computation and simulation. All the built-in agents are available through the simulator package. The simulated sensor packages and actuators that the agents use in their respective simulations are also implemented in the simulator package. The simulator package includes time slice management and simulated world physics. As such, the data representation for the simulated world is handled by the simulator package. The SIMBAD batch processor is implemented in the GUI package but it implements its own GUI; it uses a light version of the SIMBAD GUI. The light SIMBAD GUI visualizes the world but it only displays one in several hundred frames of computation. It does not implement any controls or agent interface displays; it is designed to perform autonomous robotic simulations as quickly as possible.

3.4.5 Agents

The SIMBAD simulation system already supports several agents and is easily extended to include other autonomous agent designs. There are several sensors and actuators implemented for use in autonomous agents, including:

- A camera sensor for visualizing the three-dimensional world at the agent level
- A gripper actuator that allows agents to grapple objects in the simulated environment
- A lamp actuator that can be switched on and off by the agent or be set to a flashing state
- A light sensor, which allows agents to detect sources of light
- The range sensor belt, which can be configured to simulate laser range finders, sonar, radar and bump sensors.

In the NS system extra agents are implemented to perform required tasks. Since the system uses ANNs for the control systems of the learning autonomous agents, the NS supports a neural agent. This neural agent is used as the model for all the learning robots attempting to solve the dynamic obstacle avoidance problem in this project.

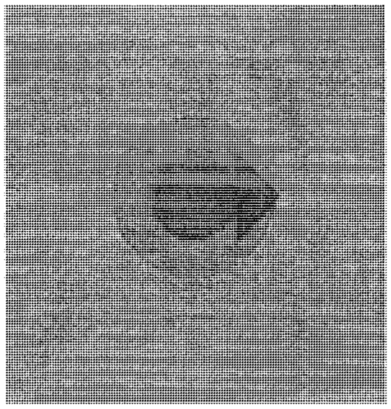


Figure 3.8: Neural Agent Top Down View. The SIMBAD neural agent is configured with twelve laser range finders distributed evenly around the circumference of the agent. The top of the agent is equipped with a lamp actuator that lights up when the agent detects incoming collisions. The agent uses two stepping motors for movement; these are not visible.

The extra agents implemented in the NS also use limited global positioning system (GPS) devices. These devices allow each agent to know how far it is from a given goal coordinate. The simulated agents must have a goal to move towards in order to engage in path-finding and obstacle avoidance. The neural network controllers for the neural agents each have thirteen input nodes and two output nodes. The input nodes take readings from the twelve laser range finders and the GPS as their inputs and the output nodes control the agent's translational and rotational velocity. Because this is a simulation the agents have ideal conditions to learn in. Simulations that are evolved in ideal environments do not fare as well in real-world environments. To help mitigate this problem the neural agent introduces random noise into its input sensor data, which can be equivalent to several centimetres of variance in range and distance readings.

Not all the agents in the simulated environment are neural agents. Several dumb agents are implemented to introduce a dynamic element to the training environments. Straight-to-goal agents start at one location, turn towards their goal, and move directly towards it. They avoid obstacles using rudimentary turn-to-avoid protocols. Once an obstacle has been avoided the dumb agent resumes its direct course to its goal and stops when it reaches it. Straight-to-goal loop agents work exactly the same way as straight-to-goal agents except that once they arrive at their goals, their goal points are changed to their original start points and they turn to move towards their new goals. Chaos agents randomly move around the environment in an erratic manner; they have no goals and do not stop moving unless they get stuck.

3.4.6 Environments

Several simulation environments are available in the SIMBAD simulator. For the dynamic obstacle avoidance problem, the development of the NS required the addition of three specific environments to this selection. These include a maze, a busy hallway, and a busy room environment.

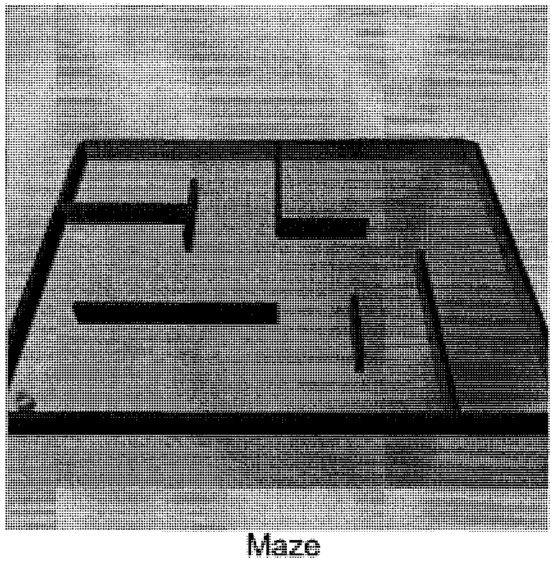


Figure 3.9: Maze Environment. The maze environment is designed to help develop the agents' ability to avoid static obstacles and perform rudimentary path-finding. The agent starts in the bottom left corner of the environment and its goal is located at the top right corner of the maze.

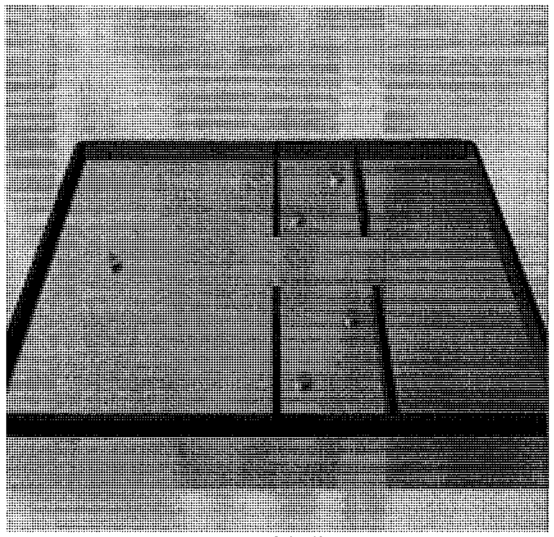


Figure 3.10: Busy Hallway Environment. The busy hallway is used to introduce the agents to path-finding through a dynamically changing environment. The straight-to-goal loop agents in the middle hallway move diagonally to the opposite end of the hallway. The neural agent starts in the middle of the left room and its goal is located in the middle of the right room.

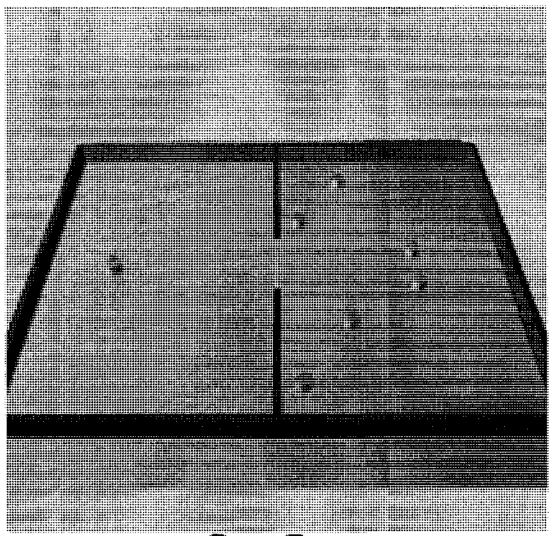


Figure 3.11: Busy Room Environment. The busy room is a more difficult version of the busy hallway. The four dumb agents in the middle behave the same way as the busy hallway agents. The two dumb agents to the far right are straight-to-goal loop agents. Their goals are located in the middle of the north and south ends of the left room. They move diagonally towards those goals. The neural agent's goal is in the same location as it was in the busy hallway environment.

The maze environment is meant to help the agents evolve rudimentary path-finding and wall avoidance behaviour. The busy hallway and busy room environments are designed to help them evolve dynamic obstacle avoidance behaviour. These environments are meant to increase in difficulty as the agent attempts them in order. The maze environment requires no dynamic obstacle avoidance behaviour of the agent, the busy hallway requires the agent to dodge obstacles that are moving perpendicular to its goal direction and finally the busy room environment has several agents that all collide near the opening between the neural agent's starting room and its goal location in the adjacent room.

3.4.7 Holodeck

SIMBAD includes a virtual environment that can be viewed as experiments are running. However, there is nothing in the system that can replay experiments after they have taken place. For this project, the holodeck application was added to the NS to solve this problem. Code components that store and retrieve trained neural networks from disk were also added. The experiments that the NS performs for this project span several hundred generations; without a method to restore these trained networks there would be no practical way to evaluate how the agents perform. A visualization of how the agents perform given a certain level of fitness is valuable for tuning the fitness function.

The holodeck is very similar to the SIMBAD simulator environment. It differs from SIMBAD's simulator in that its specific purpose is to simulate trained agents in any environment that the holodeck supports. If the environment does not specifically support neural agents as well as load neural networks from the stored neural agents, it will not work in the holodeck. The holodeck could easily be extended to support more simulation environments. It could also load trained agents into environments into which they have never been introduced, provided the environment supports this. This tool speeds analysis of the neural agents as the experiments of an EANN cannot all be viewed simultaneously. The holodeck allows targeted viewing of agents.

3.4.8 Extensions to PicoEvo

As has been mentioned, the PicoEvo system initially only supported WEA-style EANNs. In this project's NS, PicoEvo was extended to support components to implement NEAT and SDNEAT, HEA-style EANNs, and a statistics-gathering package. The components added or modified to support this more complex form of EANN include:

- the NEAT Gene
- the NEAT individual
- the NEAT population
- the NEAT population innovation list
- the NEAT population species list
- NEAT and SDNEAT parameter sets
- the NEAT and SDNEAT statistics package
- NEAT and SDNEAT selection operators
- the NEAT element variation operator
- the NEAT individual variation add-link operator
- the NEAT individual variation add-node operator
- the NEAT population variation crossover mutation operator
- the SDNEAT individual variation segmental duplication operator

The NEAT Gene serves as the basic gene for the genetic encoding of the PicoNode-based ANNs. There are two types of gene in NEAT and SDNEAT. They have some matching characteristics. For example, they both use innovation numbers. NEAT Gene stores these values. The two types of gene are as follows:

- The NEAT LGene is the extension to the NEAT gene that allows the storage of link gene-specific information in NEAT and SDNEAT.

- The NEAT NGene is similar to the LGene in that it is the extension to the NEAT gene that allows storage of, in this case, node-specific information.

The NEAT individual serves as the actual genome. This component contains all the NGenes and LGenes that compose one NEAT or SDNEAT genome. It also provides the function to convert genomes into phenomes.

The NEAT population includes all the individuals that move through the GA. The number of individuals in a population is limited only by the hardware's capability. NEAT populations are compatible with SDNEAT populations.

The NEAT population innovation list works in conjunction with the NEAT population. It tracks all the genomic innovations that happen through link and node mutation, or in the case of SDNEAT through segmental duplication mutation.

The NEAT population species list handles speciation of the population. The population does not directly separate all the genomes into their different populations; instead, the population species list keeps track of which agents are in which species population and presents that data as required to the GA.

NEAT and SDNEAT parameter sets are the sets of variables that control how the algorithm executes. They control all the probabilities of crossover and mutation operations. The parameter sets define the size of the population, the number of generations, the degree of mutation, the range of weight perturbations that can occur during a link or node mutation, and other parameters that are fully defined for each experiment. In the case of SDNEAT, extra parameters are required to control how often a segmental duplication occurs and by how far to exceed the normal mutation rate during a segmental duplication.

The NEAT and SDNEAT statistics package records statistical data for each generation of every experiment performed in the NS. The statistics include:

- Generation versus Fitness: Maximum, Minimum, Mean, Median, Best Current, and Best Ever.
- Generation versus Connections: Maximum, Minimum, Mean, Median, Best Current, and Best Ever.
- Generation versus Innovation: Number of Innovations and Number of New Innovations.
- Generation versus Nodes: Maximum, Minimum, Mean, Median, Best Current and Best Ever.
- Generation versus Species Size: This statistic keeps track of all species from the beginning of an experiment and logs their size versus the generation. This information is valuable as it shows which species performed the best, which had the most population at any point, and how long that species lived.

NEAT and SDNEAT selection operators are separate classes in the PicoEvo implementation. The selection operator chooses which genomes are allowed to mate and handles all operations, including crossover and mutation. Since SDNEAT implements an extra mutation operator there must be a separate selection operator for it.

The NEAT element variation operator perturbs the weights in both links and nodes when a weight mutation occurs. The NEAT individual variation add-link operator performs a link mutation when the selection operator performs the mutation, and the NEAT individual variation add-node operator performs a node mutation when the selection operator performs the mutation.

The NEAT population variation crossover mutation operator performs crossover on two genomes as defined by the NEAT algorithm. Crossover is the same in NEAT and SDNEAT.

Finally, the SDNEAT individual variation segmental duplication operator performs a segmental duplication, as defined in the SDNEAT algorithm, on a random segment of a genome chosen by the selection operator.

3.4.9 Extensions to PicoNode

The original PicoNode supports almost all ANN operations required by the EANNs NEAT and SDNEAT. The one operation added to the original PicoNode package for this project is a function that serves to update a genotype. When performing a NEAT or SDNEAT experiment the genomes must be converted to phenotypes in order to be evaluated in the SIMBAD virtual environment. Once the evaluation is complete the update-genotype function updates the original genome from the trained phenome.

3.4.10 Neuroevolutionary Solver Applications

The combined components of SIMBAD, PicoEvo and PicoNeuro, with the added implementations of NEAT and SDNEAT, allow for the development of several test applications.

The XOR simulation uses the simple problem of evolving a neural network to approximate the XOR function as a benchmark for the performance of implemented versions of NEAT and SDNEAT. Since the XOR function can be solved by a neural network with a minimum of one hidden node, both NEAT and SDNEAT should find a solution easily and efficiently. SDNEAT will not perform any better than NEAT at this task as SDNEAT does not gain any benefit over NEAT until multiple hidden nodes have been introduced to the population of genomes.

The avoider robot application is explicitly created for this research project. It implements several test environments using a batch simulation method, moving the agent being evaluated between the different test environments before calculating a final fitness score for the agent. This system takes full advantage of the capabilities of the NS and can be run in both the NEAT and SDNEAT versions.

The complete NS system allows for broad experimentation using both the XOR simulation and the avoider robot application. Several experiments that attempt to solve the dynamic obstacle avoidance problem are evaluated in the next chapter. These experiments also allow for an objective comparison of NEAT's and SDNEAT's performances.

Chapter 4

IMPLEMENTATION AND RESULTS

This chapter explores a set of experiments performed in both the XOR application and the avoider robot application in an attempt to solve the dynamic obstacle avoidance problem. To establish a benchmark for performance, both the NEAT and SDNEAT algorithm implementations in the NS system are evaluated with multiple experiments using the XOR application. In a second set of experiments the NEAT and SDNEAT algorithms are used in conjunction with the avoider robot application of the NS to search for a solution to the dynamic obstacle avoidance problem. The results of this set of experiments are also explored in detail in this chapter.

4.1 XOR

The XOR problem can be used as a basic benchmark for the capability of a TEA or HEA to solve complex problems using neural networks. XOR is a binary logic function. Logic functions are used in both computer software and hardware to solve logic problems.

x	y	x + y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4.1: Truth Table for XOR. Since XOR is a logic function, the only possible input values for it are true (1) and false (0). This table shows that XOR's output value is false whenever its two inputs are equivalent and true when its inputs are different.

XOR's output values are not linearly separable. This means that the two types of output values, one and zero, cannot be separated by a single linear function. The XOR function cannot be solved by neural networks that have no hidden nodes. This makes XOR a good function with which to evaluate a TEA's or HEA's ability to solve problems that require topological growth.

The XOR experiment shows that the implementations of NEAT and SDNEAT later used in this project have the capacity to find solutions with efficient topological structure. The minimal neural network structure required to implement (but not solve) the XOR problem comprises one output node and two input nodes. The minimal structure required to solve the XOR problem requires the addition of one hidden node that is connected to both input nodes and the output node.

NEAT has been shown to solve the XOR problem efficiently (Stanley, Efficient Evolution of Neural Networks through Complexification, 2004). The goal of this experiment is to show that SDNEAT can solve XOR equally efficiently, or nearly so. Since SDNEAT is based on the NEAT algorithm it should be able to solve XOR. However, the addition of the segmental duplication mutation may hinder the algorithm's capacity to find simple solutions due to its increased rate of node mutations. The optimal XOR solution may not evolve before a segmental duplication needlessly complicates the network's structure.

4.1.1 Evaluation

The fitness of the XOR networks was evaluated based on the output they delivered. All possible inputs were tested against the trained networks and the output was evaluated based on expected values. If the output value of the output node was at or above 0.50 it was deemed to be a one and if the output value was less than 0.50 it was deemed to be a zero. This evaluation of output was appropriate because this implementation of NEAT and SDNEAT used only log-sigmoid activation functions in the neural network nodes.

The initial population of agent neural networks had no hidden nodes and only had links from the input nodes to the output node. The weights of the links were all set to one. The bias value of each node in the neural networks was set to one. The bias was not allowed to mutate during evolution, nor was it adjusted through training.

The sums of the distances of the output values from their respective correct output values were subtracted from four and then squared to obtain the fitness values of solutions. The sums of the distances were subtracted from four so that higher fitness values equated to better fitness, and squared so that the relative values of the solutions were represented.

4.1.2 Experimentation

Fifty experiments were performed using XOR, twenty five using NEAT, and twenty five using SDNEAT. Neither algorithm found the optimal solution of one hidden node. It has been shown in the past that NEAT can evolve the optimal solution (Stanley, Efficient Evolution of Neural Networks through Complexification, 2004) but it does not always find it. On average NEAT found a solution in 30.2 generations and SDNEAT found a solution in 24.52 generations. On average, the NEAT solutions used 3.64 hidden nodes and the SDNEAT solutions used 4.4. It is not surprising that SDNEAT found solutions in a shorter amount of time. There are several solutions for the XOR problem that use multiple hidden nodes. SDNEAT's solutions are larger in structure, and these more complex solutions, while less efficient than their simpler counterparts, still effectively solve the XOR problem. A comparison of the two algorithms' solutions suggests that SDNEAT can find efficient solutions to complex problems as well as NEAT.

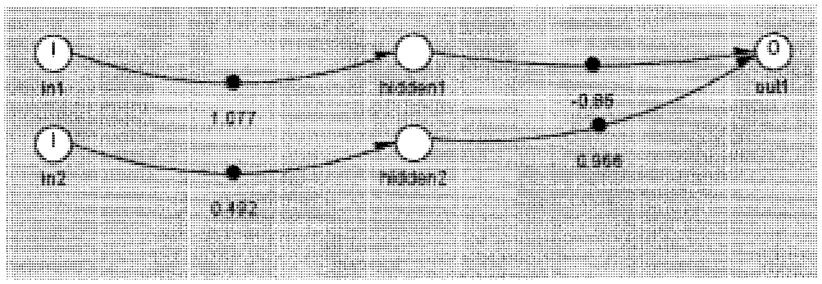


Figure 4.2: Two-Node NEAT Solution. The simplest topology found in the twenty five NEAT experiments used two nodes: one between each input, leading to the output node.

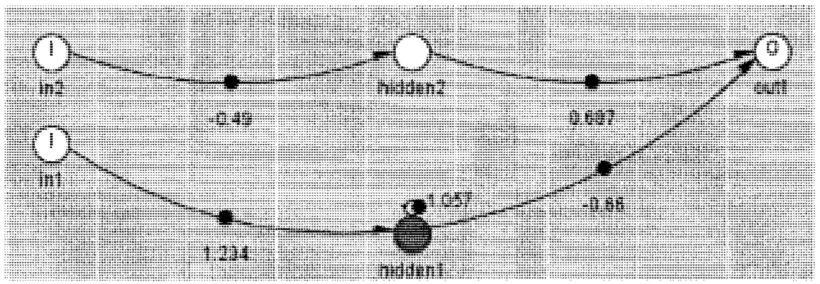


Figure 4.3: Two-node SDNEAT solution. The simplest topology found by SDNEAT used two nodes and one recurrent link. SDNEAT essentially found the same minimum structure as the NEAT implementation with the random introduction of one extra link.

NEAT and SDNEAT found similar minimal topologies to solve the XOR problem. The average number of generations it took for the algorithms to solve the problem indicates that SDNEAT can find efficient topologies for complex problems faster than NEAT can.

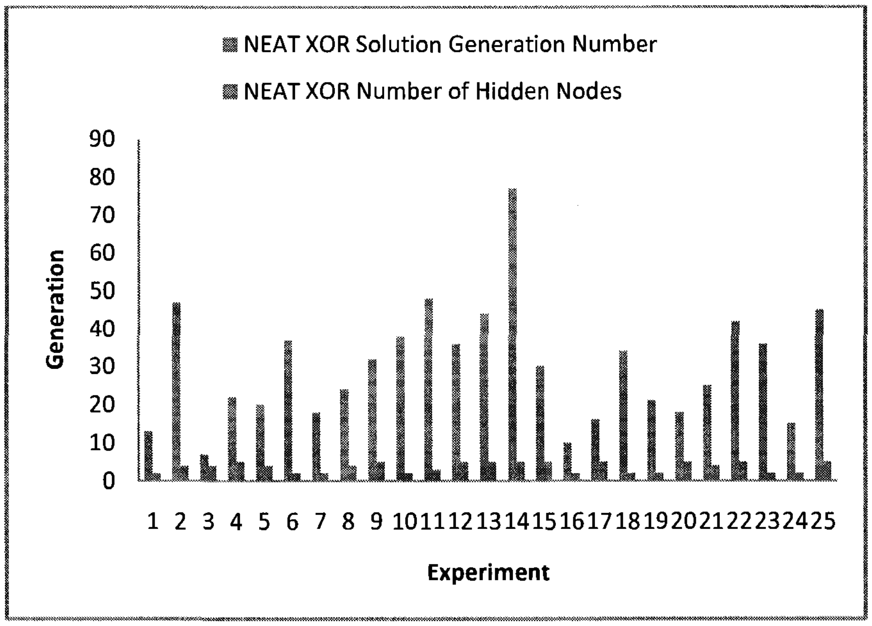


Figure 4.4: NEAT XOR Performance. NEAT found several solutions ranging in complexity from two to five hidden nodes. The shortest time-to-solution was seven generations and the longest was nearly eighty generations.

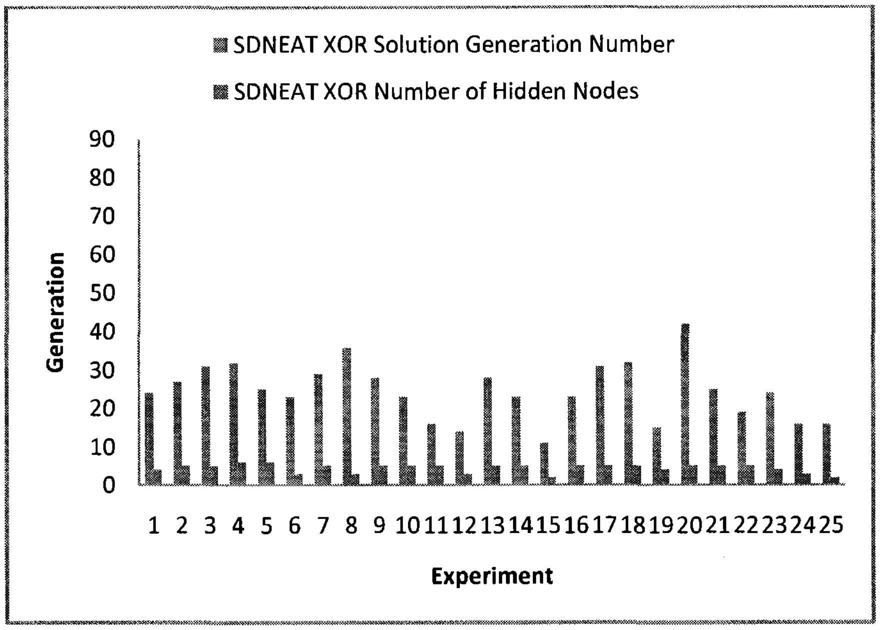


Figure 4.5: SDNEAT XOR Performance. The SDNEAT algorithm found solutions ranging in complexity from two to six hidden nodes. The length of time it took to find those solutions was more consistent; the shortest length of time was eleven generations and the longest was nearly 50.

The NEAT and SDNEAT implementations used in the NS quickly found relatively efficient topologies for solutions to problems that required introduction of new topological structure. This reliably shows that NEAT and SDNEAT can probably be used to solve complex problems.

4.2 Dynamic Obstacle Avoidance

The primary goal of this thesis is to show that the dynamic obstacle avoidance problem can be solved using a neuroevolutionary algorithm. Because the problem of dynamic obstacle avoidance is so broad and the possible solution methods are so diverse, a small subset of the problem was defined as the problem area for this thesis. A solution for this subset was sought using the NS system and both the

NEAT and SDNEAT algorithms. This section includes a definition of the experiment, a description of the process used to achieve results and a discussion of the results of the experiments.

4.2.1 Problem Domain

The problem of dynamic obstacle avoidance is huge. It can vary enormously in scale, involving small simulated autonomous agents avoiding tiny obstacles in a maze, or powerful non-virtual robots moving pallets around in a warehouse. To attempt this problem effectively, a domain must be defined to perform experiments in and gather results from.

The problem domain used in this thesis is a simulated set of static and dynamic environments, which were described in Chapter Three. The environments include a maze, a busy hallway and a busy room. The three environments are designed to require an increased level of complexity in the avoidance behaviour required to master them. During an experiment, the agent being trained is placed in the maze environment first, then the busy hallway environment and finally the busy room environment. Training takes place in all the environments and the agents' fitness is based on their performance in all three environments.

The maze environment does not require dynamic obstacle avoidance. Agents that solve the maze must be able to navigate from the south west corner to the north east corner where a goal has been placed. An efficient solution in this environment would take a direct line past the center walls of the maze and through a gap in the north east interior wall to reach the goal.

The busy hallway environment requires dynamic obstacle avoidance where the obstacles are not likely to be in the way most of the time. The dumb agents in the busy hallway scenario move from one end of the hallway to the other, crossing each other's paths diagonally. This can create complex agent traffic patterns in the opening through the hallway, but an optimal solution would simply move through the hallway while the dumb agents are not obstructing the opening.

The busy room environment is similar to the busy hallway environment since the northern and southern agents move in exactly the same pattern. However, this last environment adds another level of complexity. There are two agents in the east room that move to goals in the west room. These are timed to arrive at the hallway at the same time as the hallway agents converge at the opening. This creates a complex random traffic pattern as all six agents attempt to avoid each other. Their turning avoidance algorithms result in numerous collisions. This environment is designed to force the learning agent to collide with other agents.

4.2.2 Evaluation

The evaluation function for the avoider robot application is defined as the set of environments the agents perform training in. This means that the evaluation method for each individual in the population of genomes in each experiment is the set of environments containing the maze, the busy hallway and the busy room. For these experiments each genome in each population is placed in each environment for thirty thousand time slices and is allowed to train its neural network controller for that amount of simulation time. A population being evaluated for a specific number of generations is referred to as an experiment.

An agent moving at full speed from one end of an evaluation environment can arrive at the other end of the environment in approximately one thousand time slices of simulated time. This amount of time was increased by thirty times during the training scenarios to allow the agents ample time to arrive at any goal in the simulated environment. Since the agents are evaluated for their performance in three separate environments, the total amount of training time per agent is ninety thousand time slices.

The initial population of genomes in each experiment was comprised of identical genomes. Each genome had thirteen input node genes, two output node genes and twenty six link node genes, and their initial weights were set to one. All of the nodes used log-sigmoid activation functions and their bias values were set to one. The bias values could not be changed by mutation or neural network training. The input nodes accepted values from their simulated neural agent's twelve laser range finders and single GPS range measurement. The output nodes provided values to the simulated agents for translational and rotational velocity.

Each individual in each generation's population was evaluated in sequence; there was no parallelization of evaluation, only parallelization of experiments. Each experiment was run in a separate instance of the NS with a separate population.

The fitness of each genome in each population was defined as the sum of its calculated fitnesses in each evaluation environment. The fitness function was based on an original fitness function developed by Floreano and Mondada (Floreano & Mondada, 1998) for a WEA system. The fitness function used in the NS was adapted to include several factors that were important to the development of agents in the three evaluation environments.

$$f = s \cdot (c_1 - |a|) \cdot (c_2 - m) \cdot \left(\frac{c_3}{d}\right)$$

Figure 4.6: Fitness Function. This function was used for fitness evaluation in all the experiments performed using the autonomous agent system. The variable s is the speed of the neural agent, a is the angular velocity of the neural agent, m is the maximum sensor value currently detected by a laser range finder and d is the current distance from goal. The constants c_1, c_2 and c_3 were set to 1.0, 1.6 and 1.0 respectively. The maximum sensor value from a laser range finder is 1.5. However, because the sensor readings may incorporate random noise with a maximum value of 0.1, the value of c_2 is 1.6 so as to preclude negative fitness values.

The fitness function used in the NS incorporates a distance variable. This variable causes fitness to rise sharply as the agent approaches its goal and keeps fitness low when the agent is far from its goal. The function is replaced by a static fitness value of five when the agent has arrived at its goal. This value is substantially higher than any fitness value that can be generated by the fitness function and serves to dramatically increase the fitness values of agents that reach their goals. The static value also mitigates the problem of division by zero when the agent is exactly on top of its goal. The agent is considered to have arrived at its goal when it is within 0.5 simulated meters of it.

The fitness function evaluates the fitness of an individual genome for one time slice. The genome's fitness values for each time slice are summed over the course of its navigation through each simulation environment to produce the overall fitness value for that generation of the genome.

4.2.3 Experimentation

Eighty experiments were performed to solve the dynamic obstacle avoidance problem. Forty were performed using NEAT and forty using SDNEAT. Neither algorithm found a complete solution to all three simulated environments, but SDNEAT succeeded in finding solutions for all three environments. NEAT did find solutions for the busy hallway and busy room scenarios but failed to evolve a solution for the maze. SDNEAT evolved a solution for the busy hallway and busy room scenarios that nearly solves the maze problem as well. A few generations after this solution was evolved, a solution that navigates the maze environment was found. That agent was directly related to the best SDNEAT solution agent but unfortunately the agent that solved the maze had lost its ability to solve the busy hallway and busy room scenarios.

SDNEAT found substantially more high-fitness genomes than NEAT did using the same given GA parameters and the same number of experiments. A fit genome was defined as any genome that scored above 1000. The average score for an unfit genome was approximately 350. Most genomes that scored over 1000 did approach their goals to some extent. Agents that scored above 20,000 were considered high-fitness genomes. These genomes kept their speed high, their angular velocity low, received very little sensory input and approached their goals somewhat. A fitness of 20,000 could not be achieved otherwise and is a good benchmark fitness for agents that performed well in the dynamic obstacle avoidance experiments.

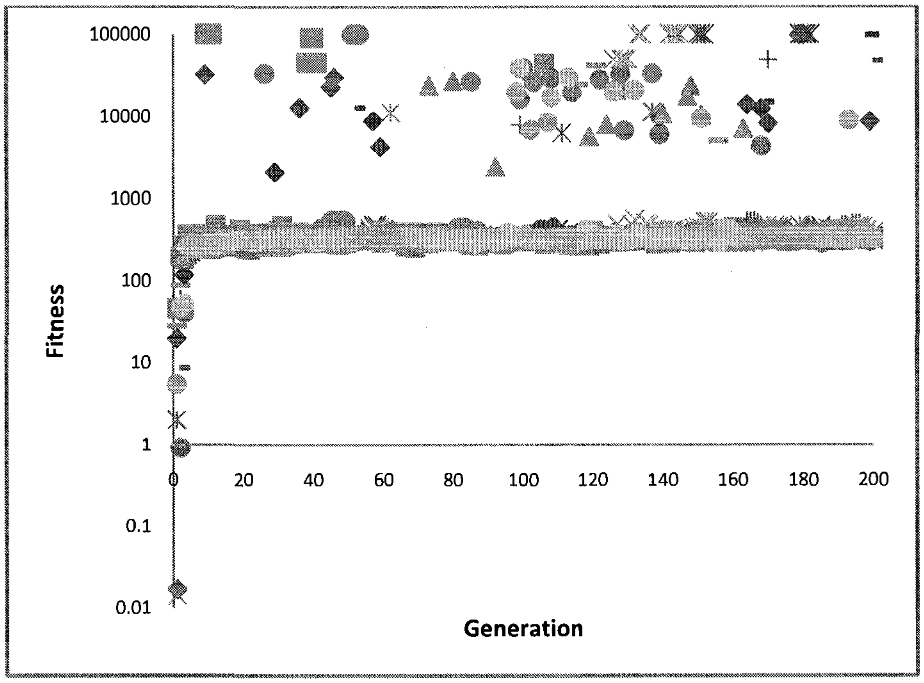


Figure 4.7: NEAT's Best Generation Fitness versus Generation. This graph charts the fitness values of the highest fitness genomes in each generation for all forty NEAT dynamic obstacle avoidance experiments. The various colours and shapes are representative of each of these unique experiment data series. The y-axis is a logarithmic scale. NEAT does evolve some very high fitness solutions quickly.

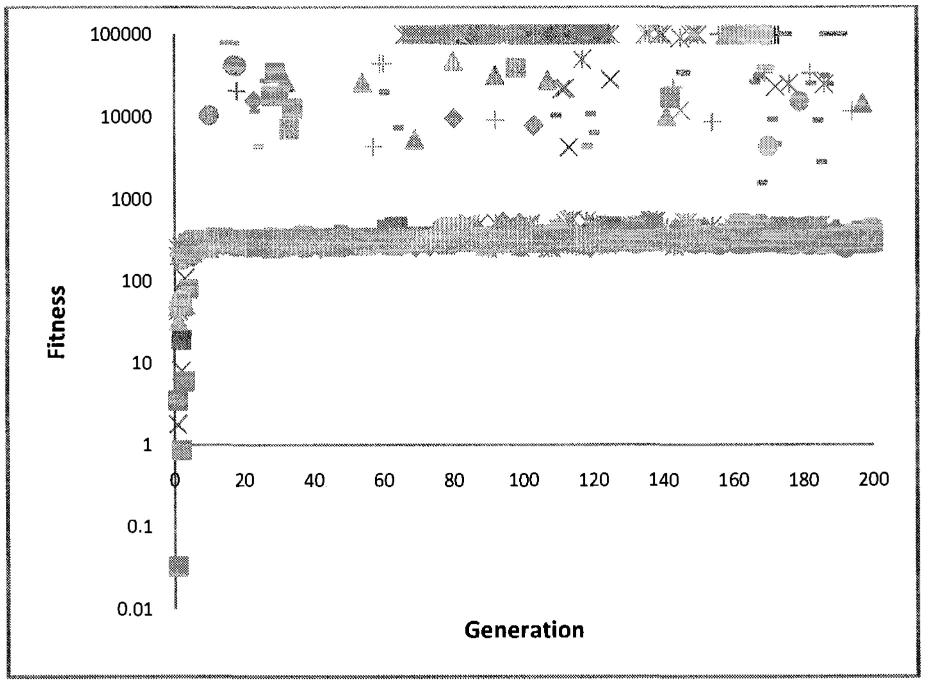


Figure 4.8: SDNEAT's Best Generation Fitness versus Generation. This graph shows the fitness values of the highest-fitness genomes in each generation for all forty SDNEAT dynamic obstacle avoidance experiments. The various colours and shapes are representative of each of these unique experiment data series. The y-axis is a logarithmic scale. SDNEAT evolves a substantial number of high-fitness solutions.

Both NEAT and SDNEAT produce a substantial number of basic solutions. The above graphs show a significant grouping of solutions with fitness values between ten and one thousand. This is representative of the simplest solution in the dynamic obstacle avoidance problem search space. The solutions with fitness scores scattered between one thousand and ninety thousand represent localized maximum solutions in the search space. Most of these genomes produce solutions for one of the three evaluation environments. The third major grouping of solutions represents genomes with fitness values of almost one hundred thousand. These solutions succeed in solving two of the evaluation environments and in some cases nearly solve all three. If this subset of the dynamic obstacle avoidance problem is solvable by one neural agent then there is a third tier of solutions in the solution space with fitnesses above one hundred thousand.

The following comparisons of NEAT to SDNEAT are limited to experiments that successfully evolved multiple high-fitness solutions. Any experiment with less than two high-fitness solutions is excluded as it does not significantly contribute to the solutions of the dynamic obstacle avoidance problem. When only experiments with high-fitness solutions are taken into account, NEAT does not appear to perform as well as SDNEAT. The set of NEAT experiments resulted in only fifteen experiments with multiple high-fitness solutions. SDNEAT's experiment set resulted in twenty experiments with multiple high-fitness solutions. The high-performance NEAT experiments generated 66 high fitness solutions for an average of 4.4 solutions per experiment; the SDNEAT experiments generated 165 high fitness solutions for an average of 8.25 per experiment.

A high-fitness solution here can be an individual solution or a sequence of solutions. Sequences of solutions arise from elitism; when elitism takes effect, a solution genome passes through to the next generation. This genome may learn new behaviour from its training in the environment but it is considered the same solution for the purposes of these statistics.

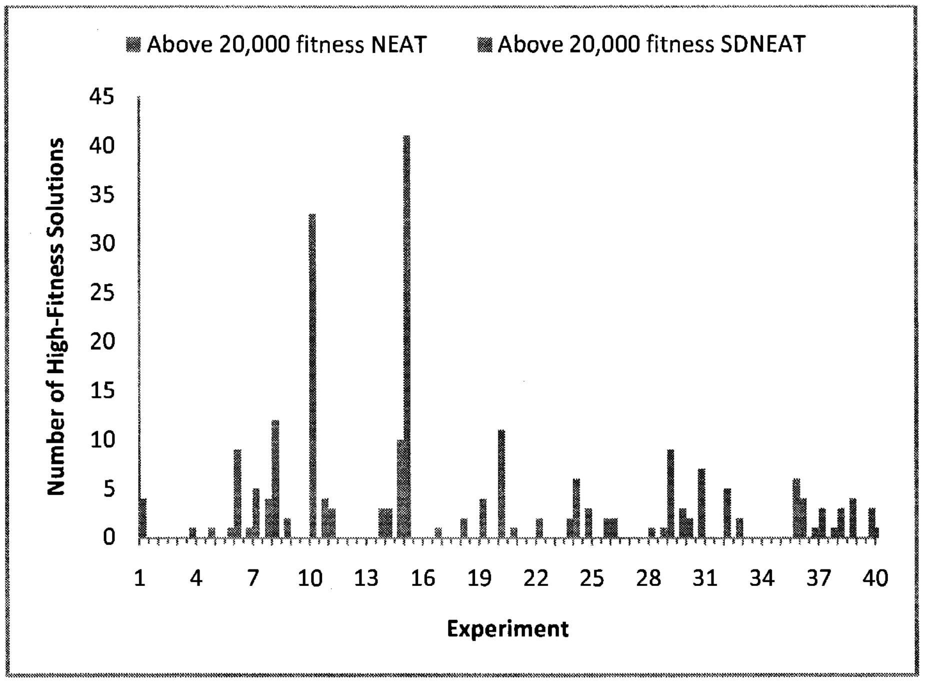


Figure 4.9: Number of High-Fitness Solutions versus Experiment Number. SDNEAT performed substantially better than NEAT in several experiments. SDNEAT's number of high-fitness solutions exceeded NEAT's by 250%.

These results suggest that segmental duplication mutation does increase the rate at which high-fitness solutions can be found for a given problem. In order to compare the relative fitness of these solutions, it is necessary to categorize the solutions based on the behaviour generated by the evolved solutions. Upon review of the behaviour of the agents in each of the three environments, it was found that their solutions for the busy hallway and busy room environments were very similar. As a result, the agents' methods of solution for both the busy hallway and busy room environments are described here with respect only to the busy hallway.

4.2.4 Solutions

All 231 high-fitness solutions were observed and categorized based on the behaviours exhibited by the evolved neural agents in the maze and busy hallway environments. During categorization each sequence of elite genomes was treated as an individual solution. During each generation each genome is evaluated ninety thousand times to form its fitness score. While it is being evaluated, it is also learning, which can change its behaviour both during evaluation and in future generations. These changes in behaviour can cause it to be categorized differently. In this project, when changes in the behaviour of an elite agent were drastic enough to warrant a different categorization, they were considered new agents. This expanded the number of high-fitness NEAT solutions to 94 and the number of SDNEAT solutions to 190 for a total of 284 high fitness solutions.

Eight categories of behaviour emerged from the high-fitness solutions. The following list of categories starts with the simplest solution and progresses towards more sophisticated and complex solutions. Category 8 is the best solution found.

4.2.4.1 Category 1:

In the maze, the agent moves towards the nearest wall and gets stuck against it. In the busy hallway, the agent moves in small circles in a southeast direction. When near a wall the agent continues to turn in small circles and follows the wall north towards the hallway opening. Once at the hallway, it turns towards the goal and attempts to move through the hallway, still turning in small circles, and avoiding the dumb agents until it reaches its goal. In some variations the agent gets stuck against the inner hallway walls while attempting to move through the hallway. Some Category 1 agents, when performing wall-following and moving towards the hallway, move next to the wall without turning in circles before switching back to circular movement as they move through the hallway.

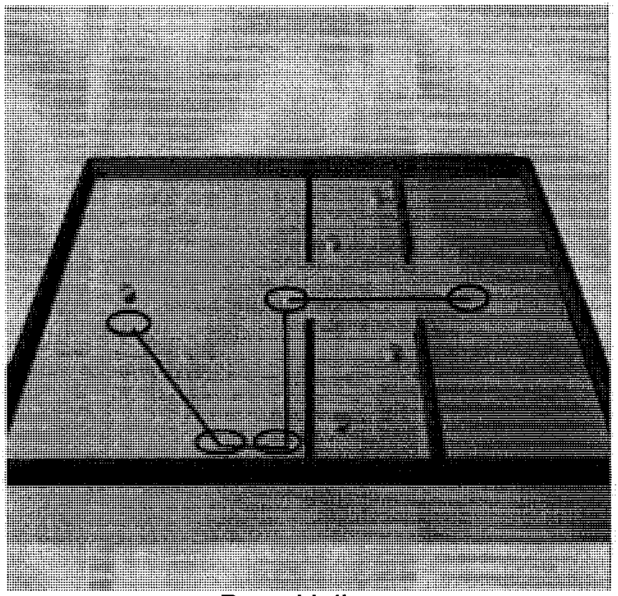


Figure 4.10: Category 1 Solution. This solution is the most common in both NEAT and SDNEAT.

4.2.4.2 Category 2:

The agent spins in the corner of the maze. It may move further into the corner or very slightly out of the corner. In the busy hallway environment the agent moves in a northeast direction in a circular pattern. The circles may be large or small. When the agent moves close to the north east corner of the east room it enlarges the turning radius of its circular movements and makes a large sweeping curve through the hallway and into the west room. The agent may or may not attain the goal. If it does not attain the goal it gets stuck against the first wall it contacts.

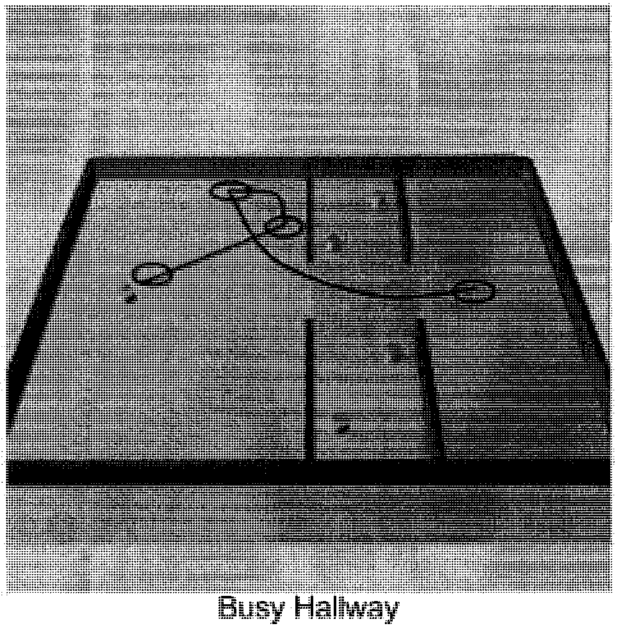


Figure 4.11: Category 2 Solution. This solution is much less common than the other solutions. It is also inaccurate and prone to missing the goal.

4.2.4.3 Category 3:

In the maze, the agent moves towards the nearest wall and gets stuck. In the busy hallway scenario the agent moves directly south, and while keeping its turning radius as large as possible it turns towards the east and orients itself towards the goal location in the west room. The agent then speeds up, decreases its turning radius to zero and moves straight towards the goal. There are several slight variations on this theme. The agent may move slowly or quickly through the turn, but it always moves quickly through the straight portion. The agent also may alter its directional vector to avoid the edge of the southern wall but after it has passed the wall it straightens its course and, typically, arrives at its goal.

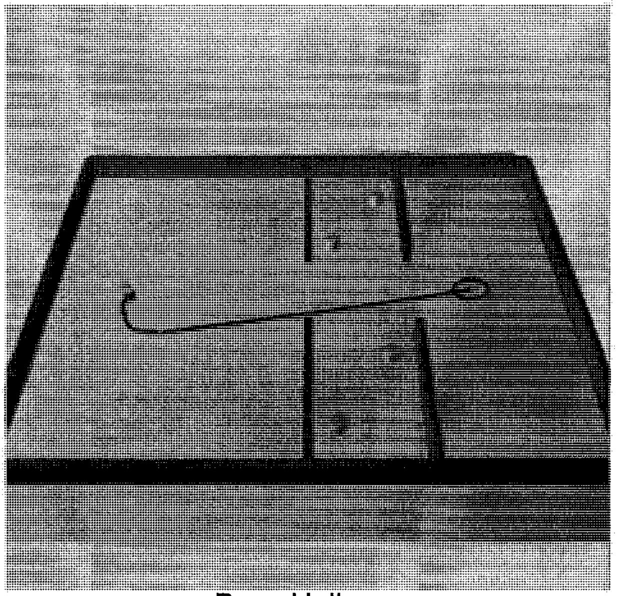


Figure 4.12: Category 3 Solution. This is the most common solution in both NEAT and SDNEAT. The agent typically performs slight course corrections to avoid the first southern wall. Since it then usually proceeds to the goal as fast as possible, it avoids the hallway agents completely.

4.2.4.4 Category 4:

The neural agent spins in the corner of the maze and makes some movement outward from the maze corner, either to the north or the east. The agents in this category behave the same way as Category 3 agents in the busy hallway and busy room scenarios. These agents are considered separate from Category 3 agents because they are evolutionary precursors to later solutions.

4.2.4.5 Category 5:

In the maze environment the neural agent either spins in the corner, or makes small erratic movements away from the corner but gets stuck in the middles of hallways and does not progress to its goal. In the busy hallway and busy room scenarios Category 5 agents have an interesting solution. They move directly southeast in an elongated ellipse pattern. Then they curve back towards their starting point, adjust their trajectories when they near the western wall, and then move along a long curve through the hallway and to the goal.

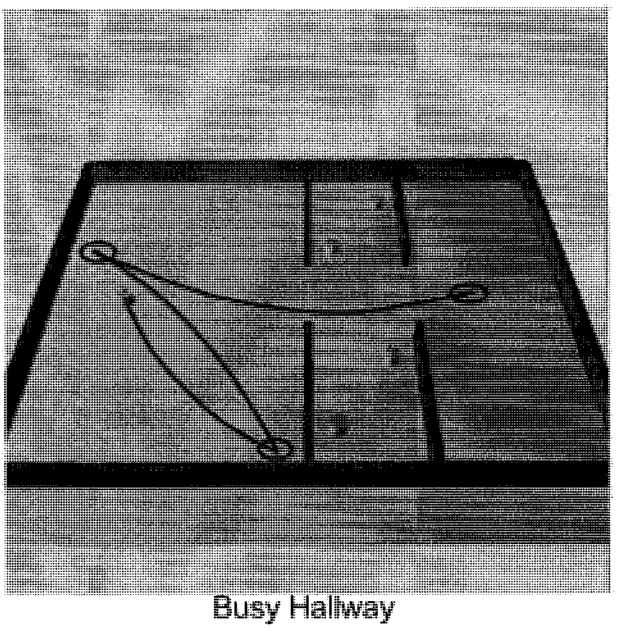


Figure 4.13: Category 5 Solution. This is one of the most interesting solutions in both the NEAT and SDNEAT experiments. It is accurate and may be a precursor to the Category 3 solutions.

4.2.4.6 Category 6:

These neural agents sometimes behave like Category 3 or Category 4 agents, but they slow down to navigate past static obstacles and speed up to push dynamic obstacles out of their paths. This behaviour is akin to "bullying". This was the most advanced solution evolved by the NEAT algorithm.

4.2.4.7 Category 7:

In the maze environment, a Category 7 neural agent spins on its center point or in tight circular movements and follows nearby walls all the way to its goal. In all the experiments, this was the only solution to the maze evolved, and it was only evolved in SDNEAT. In the busy hallway and busy room scenarios it has no wall to follow near its starting point, and it simply spins.

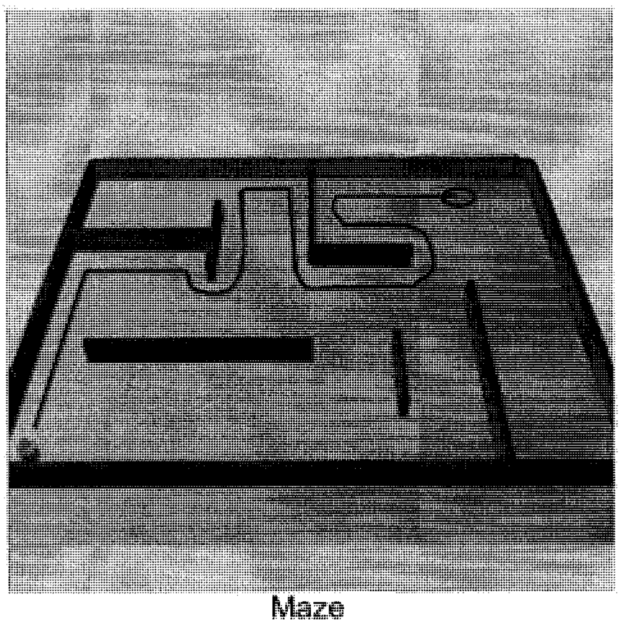


Figure 4.14: Category 7 Solution. This is the only evolved solution to the maze.

4.2.4.8 Category 8:

Similarly to Category 7 solutions, Category 8 neural agents exhibit wall following. However as a Category 8 agent is progressing northward against the wall, it eventually increases its turning radius too much and gets stuck. The agent solves both the busy hallway and busy room environments using a Category 6 approach.

All of the solutions evolved by both NEAT and SDNEAT fall into one of the described categories. NEAT evolved solutions in Categories 1, 2, 3, 4 and 6. Its best solution fell into Category 6. It did not successfully evolve any other solutions. SDNEAT evolved solutions that fit into all the categories. This suggests that SDNEAT's segmental duplication mutation may cause the populations to evolve into a more diverse set of solutions.

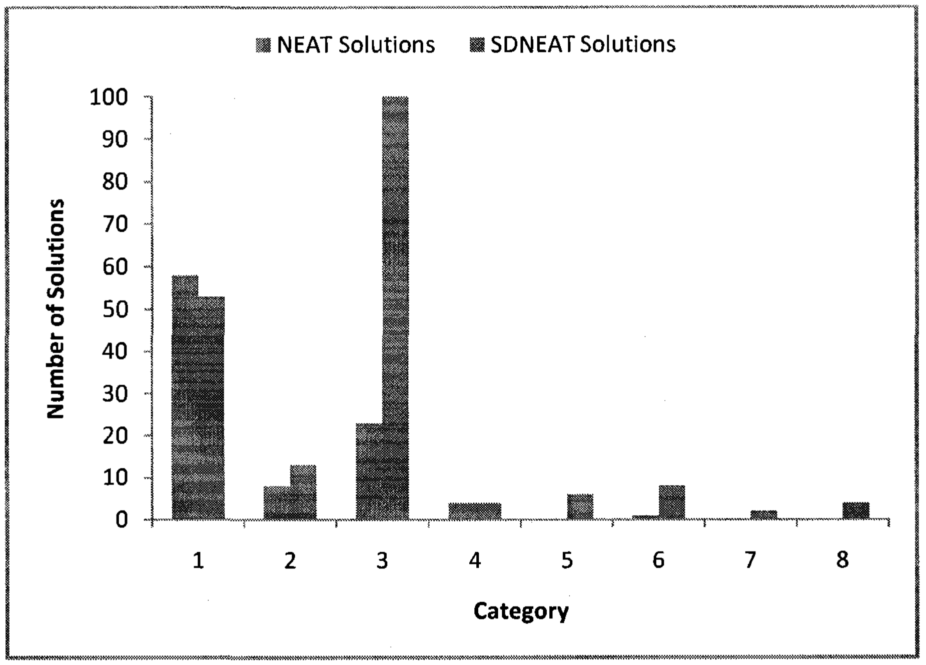


Figure 4.15: Number of Solutions versus Category. This chart shows that SDNEAT outperforms NEAT in evolving complex solutions.

In Figure 4.15 it is clear into which category each algorithm's most advanced solutions fall. NEAT evolved a Category 6 solution that SDNEAT also evolved. SDNEAT evolved more sophisticated solutions, including a Category 7 solution which was a wall-follower that solved the maze problem, as well as a Category 8 solution that integrated wall following behaviour with the Category 3 and 4 solutions that accurately and efficiently found the goal in the busy hallway and busy room scenarios.

4.2.5 The NEAT Solution

The Category 6 NEAT solution was evolved in the two hundredth generation of NEAT Experiment 25. It was composed of 25 neuron genes including its input and output genes, and 60 link genes. Its fitness value was only 47237.3, which suggests that it either did not reach its goal in the busy room or the busy hallway scenario. This also suggests that its goal finding was not as accurate as that of other evolved solutions. This agent did exhibit behaviour that incorporated some elements of wall following; in the maze environment it turned on its center point and moved towards the starting corner until it got stuck. In the busy hallway and busy room environments it proceeded towards its goal as fast as possible. It slowed down to avoid static obstacles and sped up to push dynamic obstacles out of its way.

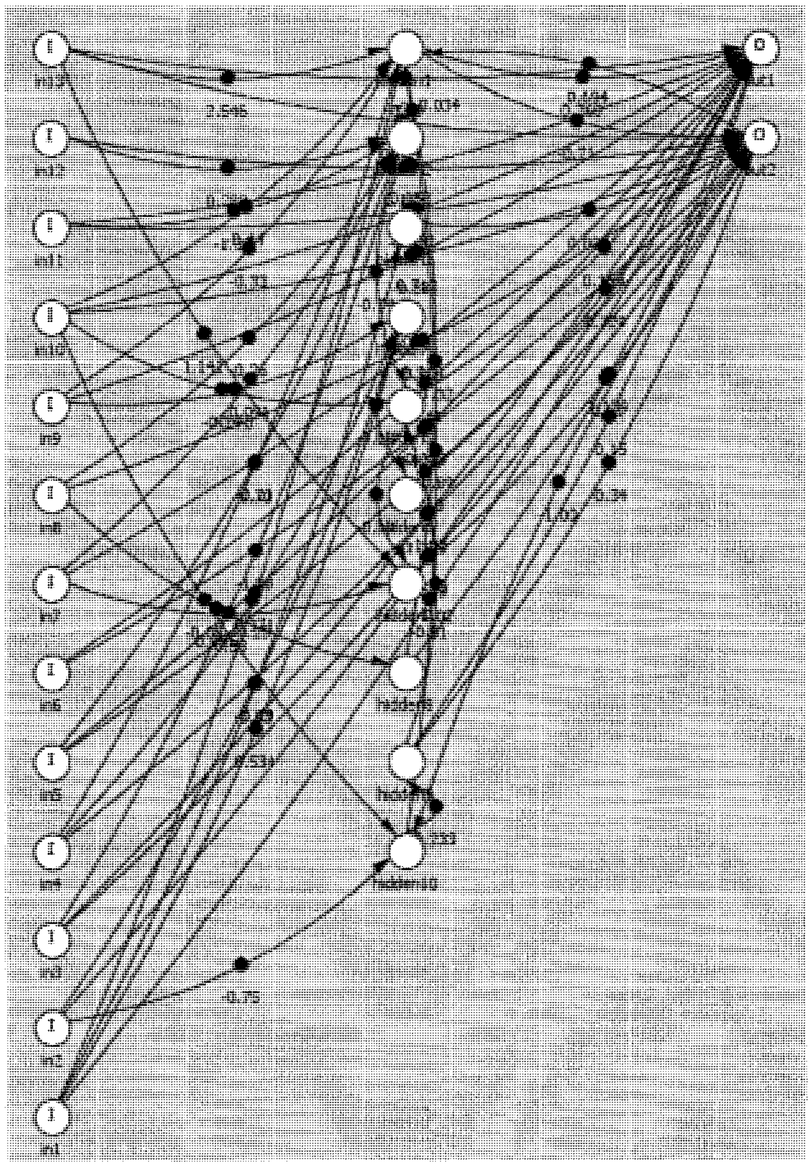


Figure 4.16: NEAT Solution Topology. This is the NEAT algorithm's evolved solution to the dynamic obstacle avoidance problem.

The NEAT solution was a relatively low-scoring population; not many of its agents evolved high fitness values until the last generation.

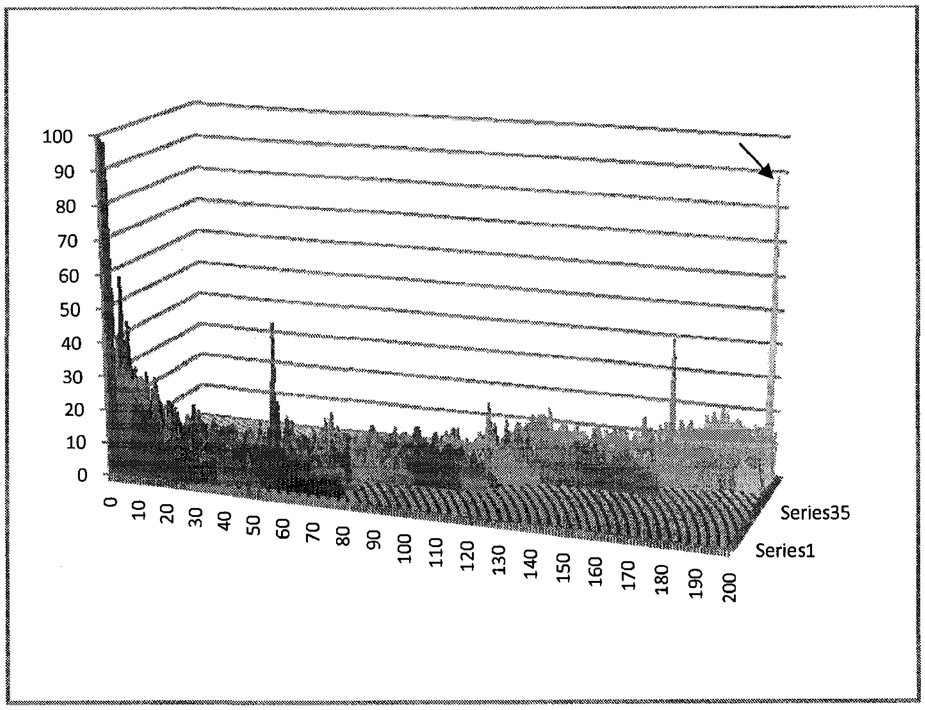


Figure 4.17: NEAT Species History. This image displays the species information for the NEAT solution. The x-axis shows the generation number, the y-axis shows the population size and the z-axis shows the species. There were 47 species over 200 generations. No species achieved significant dominance in the population until Generation 200. The sharp spike marked with an arrow is the population that the NEAT solution evolved in.

4.2.6 The SDNEAT Solution

The category 8 SDNEAT solution evolved in Generation 88 of SDNEAT's fifteenth experiment. It was composed of 19 neuron genes including its input and output genes, and 37 link genes. Its fitness value was 98875.6, which suggests that the agent successfully reached two out of three goals. This agent's goal-finding was quite accurate. It has only 19 neuron genes; its topological structure is substantially less complex than the NEAT solution. Since 15 of its genes were already dedicated to input and output nodes, this solution required only four hidden genes.

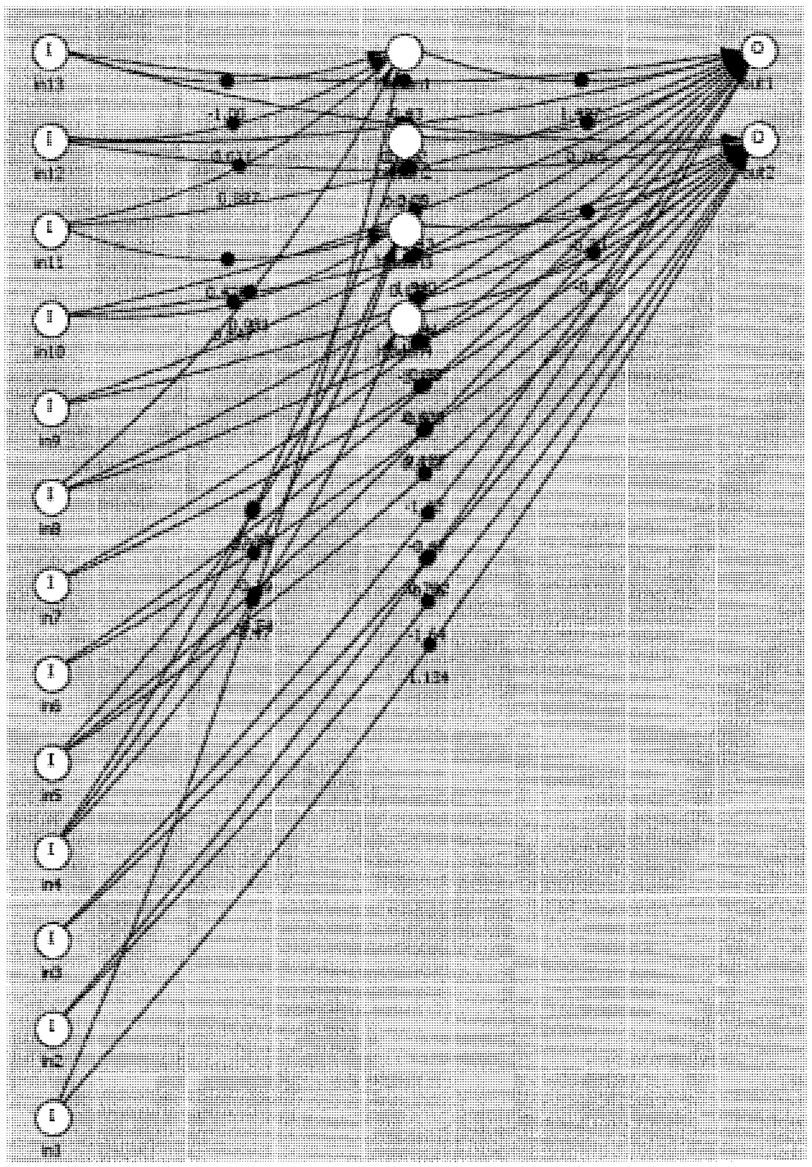


Figure 4.18: SDNEAT Solution Topology. This image shows SDNEAT's solution for the dynamic obstacle avoidance problem. This solution is substantially less complex than the solution evolved by NEAT, which is shown in Figure 4.16.

This neural agent exhibited more advanced wall following than the NEAT solution. Like a Category 7 agent would, in the maze it initially followed the west wall, but eventually its turning radius increased until it got stuck turning into the wall rather than continually avoiding it. In the busy hallway scenario the agent proceeded to the goal so rapidly that it completely avoided the dumb agents. In the busy room environment, the agent did not avoid the east-to-west agents and collided with one of them on the way to its goal. It did not slow down or avoid the dumb agent; it proceeded directly to the goal by pushing the dumb agent out of its path.

The best-performing SDNEAT solution was part of a series of solutions, which continued to evolve after the best-performing solution was attained. After several more generations the same solution correctly evolved wall-following and became a Category 7 solution. Unfortunately the agent lost its ability to solve the busy hallway and busy room scenarios as a result.

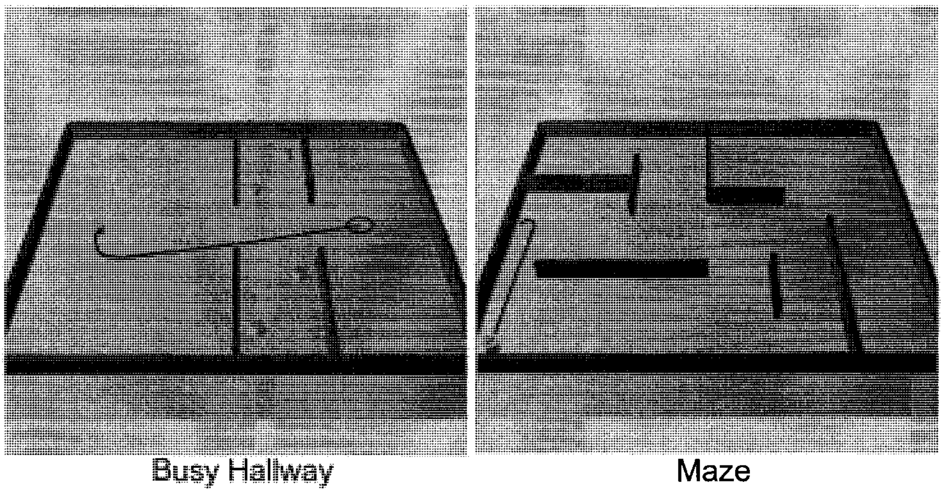


Figure 4.19: SDNEAT Solution. The SDNEAT solution nearly solved all three environments and eventually evolved into a solution that solved the maze environment. Unfortunately in the process it lost its ability to solve the busy hallway and busy room environments. The busy room environment is not shown above as the agent used the same solution there as it did in the busy hallway environment.

Interestingly the best performing SDNEAT solution had no segmental duplications in its structure. However when tracing its genetic origins, it was found that this solution was a direct descendant of its original species champion which was heavily mutated with segmental duplications. NEAT could have evolved this solution, but SDNEAT ultimately caused the solution to surface faster. Even though the final solution actually had no segmental duplications in it, it did have genes descended from a parent that had segmental duplications.

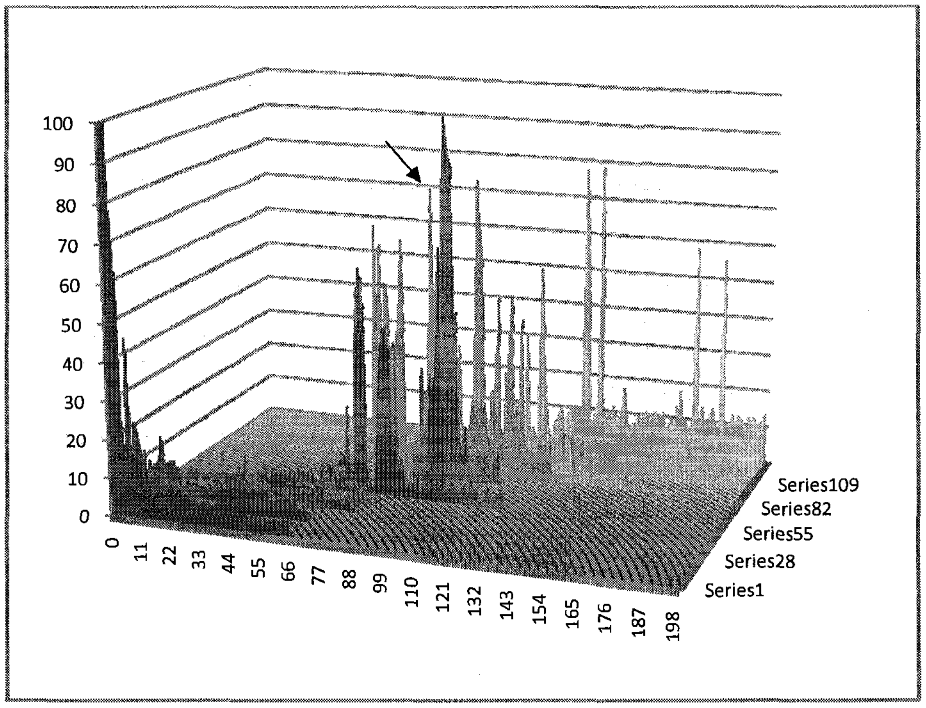


Figure 4.20: SDNEAT Species History. This image displays the species information for the SDNEAT solution. The x-axis shows the generation number, the y-axis shows the population number and the z-axis shows the species. There were 109 species over 200 generations. Several of the species achieved significant dominance in the population numbers due to their solution fitness. The sharp spike marked with an arrow is the species that evolved the best-performing SDNEAT solution.

The SDNEAT species history displayed in Figure 4.20, when compared to the NEAT species history shown in Figure 4.18, clearly shows that SDNEAT evolved significantly more high-fitness solutions.

NEAT is fully capable of evolving solutions to the dynamic obstacle avoidance problem and is capable of evolving high-fitness solutions very quickly. However, SDNEAT evolved highly sophisticated solutions faster than NEAT in this project. The SDNEAT solutions exhibited extremely high fitness and were not necessarily more complex than the NEAT solutions to the problem. While this project did not completely solve the dynamic obstacle avoidance problem, future work with the SDNEAT and NEAT algorithms may complete a solution.

Chapter 5

DISCUSSION AND CONCLUSION

The difficulty of the dynamic obstacle avoidance problem varies greatly depending on the chosen domain of implementation. Solving the problem in a single domain with WEAs and TEAs has been attempted in prior work. Searching for a solution to the problem in multiple training domains seems to be a more difficult problem to solve. Both the NEAT and SDNEAT algorithms are capable EANN systems. The NEAT algorithm can, from a base genome, methodically develop a neural network solution to very complex problems. SDNEAT has all the advantages of the NEAT algorithm and increases its performance by adding segmental duplication. These algorithms, when applied to the dynamic obstacle avoidance problem, came close to achieving an optimal solution.

5.1 Segmental Duplications

The NEAT algorithm introduces complexity to a population of genomes with a basic initial structure. It introduces this complexity gradually through mutation and crossover operators that are made manageable by the addition of historical markings to the NEAT genes. The unique solutions evolved through the gradual addition of complexity are protected by speciation. As speciation occurs, the structurally diverse genomes are broken into separate groups and given time to evolve to their fullest potential. These strengths of the NEAT algorithm are shared by the SDNEAT algorithm.

SDNEAT introduces the concept of segmental duplication within an evolutionary artificial neural network. The idea of segmental duplication is borrowed from the natural genetic processes of life on earth. Segmental duplications are thought to speed the genetic adaption of natural life (Bailey & Eichler, 2006). In SDNEAT, when segmental duplication occurs the mutation function identifies a specific sequence of nodes and links in a neural network and adds an additional segment of similar links, heavily mutated, to the same genome. It is hoped that the segment will speed the genetic adaption of the solutions in the SDNEAT population.

The complexity introduced to genomes through segmental duplication is protected by NEAT speciation. This accelerated addition of complexity has the potential to cause SDNEAT to fail to identify structurally optimal solutions that the NEAT algorithm may identify in a shorter time. This thesis showed that when SDNEAT was applied to the dynamic obstacle avoidance problem, it found higher-fitness solutions more frequently than NEAT. While SDNEAT may introduce complexity faster than NEAT, that added complexity is protected by speciation, increasing the total number of species. Each species contains a proportionally smaller segment of the population but is more dispersed in the problem solution space. The added complexity speeds the search for an optimal solution.

5.1.1 Increasing performance of SDNEAT

While SDNEAT did evolve the most effective solutions to the dynamic obstacle avoidance problem, there is potential to improve the methodology. All of the agent training in this project used unsupervised learning. During the initial simulations, agents were directly punished for colliding with an object; for the time slices during which they were in collision with another object, they received zero fitness. At first this appeared to be a good practice, but it was found that several agents quickly evolved movement toward their goals and consequently collided with a wall, ceased moving and also ceased gaining fitness. This attempt at supervising the learning of the agents resulted in undesirably low fitness values for potential solutions.

However, a more effective type of supervised learning could be implemented. Such supervision might involve observing and recording agent behaviour and modifying the fitness of an agent when poor behaviour is observed, while continuing to reward agents for positive behaviours. This method may be too complex to implement and therefore impractical. Making the training environments more random may limit the overspecialization of the solutions. This might increase the fitness of the overall best solution by making it independent of its environment.

During the dynamic obstacle avoidance experiments, the biases of the agents were set to one as a default and were not allowed to mutate or evolve. Introducing mutation or evolution of biases into the algorithm may offer slight improvements to the agents' overall performance and fitness.

In this project, the agents were allowed to train in every environment, during every time slice and in every generation in which they were evaluated. It is possible that this resulted in the neural networks becoming over-fit, and their performance decreased as a consequence. A smarter version of SDNEAT could halt learning and proceed with evaluation only when the observed level of performance reaches a certain threshold. This threshold would be dependent on the training environment and integrated as a parameter in the SDNEAT algorithm.

5.2 Dynamic obstacle avoidance

While SDNEAT did find a nearly optimal solution, it did not find a perfect solution to the defined problem of dynamic obstacle avoidance. SDNEAT also evolved substantially more high-fitness solutions than the original NEAT algorithm. The best SDNEAT solution was capable of efficiently solving the busy room and busy hallway environments, and it almost solved the maze problem. Several generations after the best solution was evolved, one of its descendants solved the maze scenario; however, its solution was not optimal. An optimal solution would have taken a more direct route to the goal and would have not spun in circles on its way there.

In an effort to optimize the evolved solutions, prior to experimentation the fitness function was carefully honed. The initial version of the fitness function included no modification for the agent's distance from its goal.

Early variation of the function involved subtracting the distance-to-goal from the calculated fitness. This resulted in negative fitness, which did not function properly in the simulator. Since the function should reduce to zero for poor fitness behaviour, using the inverse of the distance worked well. If an agent is far from its goal this inverse is a substantially low number, and if the agent is near the goal the number rises sharply.

After these changes, agents still did not progress effectively toward their goals. Various modifications were made to the fitness function to reduce the importance of speed and turning velocity in the overall fitness. None of these modifications resulted in higher-fitness solutions. After visual inspection of several experiments, an increase in the number of time slices per simulation environment was attempted. This resulted in the current value of thirty thousand time slices per environment. The extra time in each simulation allowed the neural networks to adapt further to their environments and to increase the calculated difference between high- and low-fitness solutions. This improved the ratio of high-fitness solutions to low-fitness solutions.

Further modifying the evaluation methods might result in a better selection of solutions. The fitness function used by the autonomous agent application could be modified further in an attempt to optimize the agents' calculated fitness. The score of five awarded to agents that arrived at their goals, which was used in place of the calculated fitness function in such circumstances, could be reduced to three or two. This would lower the highest-fitness score and might prevent SDNEAT from concentrating nearly all the offspring into that one high-performing species.

As species grew old, their average fitness scores began to decrease. This decrease in fitness may have been due to over-training of the neural network agents. In all the experiments, the scores of the highest-performing species eventually decreased while the scores of originally lower-performing new species increased. This was counterintuitive; it seems that per-species elitism should have prevented the highest-performing agents from being changed between each generation. However, only their topology remained static; their neural networks' knowledge and behaviour did change. The best evolved solution, which later evolved into a solution for the maze environment, lost its ability to solve the busy room and busy hallway environments. This was probably due to over-training of the neural networks. If so, the problem could be corrected using a smart-learning version of SDNEAT, as described.

5.2.1 Improving the simulation

The simulation environment could be changed to increase consistency between the experiments. Random noise was introduced into each agent's laser range-finder signals, but this noise was not normalized before fitness evaluation. As a result the fitness function may have reported a small amount of fitness when there was none. This may have skewed fitness scores slightly. This could be corrected by normalizing the sensor readings before fitness calculation but after neural network training.

The simulation system itself produced a lot of random noise through mathematical inaccuracy. The multiplicative increase of decimal inaccuracy may have led to changes in the dumb agents' behaviour between each experiment. It is unclear if this was a positive or negative influence on the neural agent training; it is conceivable that it might have prevented a good solution from reaching the goal during the early stages of its evolution. However if such a solution were truly promising, it should have avoided the random obstacles and reached the goal anyway.

Many of the simulations demonstrated that the agents preferred to spin even if it decreased their overall fitness. A potential solution for this problem would be to focus the density of sensor input from one direction. Since the agents had a uniform belt of sensors around their circumference they had no one direction that was optimal for detecting dynamic obstacles. Increasing the sensor density in one half of an agent's circumference might bias the agent towards moving in that direction. This bias is exhibited in natural life forms; for example, most human sensory inputs are focused towards one half of their surroundings.

5.2.2 Future improvements

It appeared to be more difficult for the agents to evolve solutions for finding their goals in the maze environment than in the other environments. This may have been due to the bias of having two structurally similar scenarios where agents had to navigate through a dynamic environment towards a goal, versus one static maze environment. Balancing the number of similar environments would remove the bias. Randomizing the environments, including randomizing their start and end points, might also help to evolve more robust solutions. The solutions that were found were nevertheless local maxima as the agents performed well in two of the three environments. A further level of evolution would probably find a solution for all three environments but such a solution might still not be the global maximum solution. A global solution would perform well in an environment it has never encountered. Randomizing the environments and their start points and end points might serve to evolve a robust dynamic obstacle-avoidance agent that can perform well in any environment.

5.3 Future direction and Component SDNEAT

Although NEAT and SDNEAT can evolve efficient solutions to complex problems, they are constrained by the topological limits of a single network. SDNEAT provides a way to increase the complexity of evolved solutions through a new indirect encoding method. Existing biological systems are composed of several highly-connected neural network structures that are genetically related but may have different structures and function completely differently. The nerves in the eye are closely related to the nerve structures in the brain, as they all are realized from the same DNA, but functionally the cells are quite different.

A new version of SDNEAT could implement neural controllers for agents of much greater complexity. This "Component SDNEAT" would describe its phenotype using indirect encoding of component genes and segment genes. Each input or set of inputs of an autonomous agent would be given its own neural network to train and evolve. This network would then be a component of the phenotype and would be encoded as a component gene of the genotype. The outputs of the agent would also be grouped by component in a similar fashion. A final component would be added that would not be connected to the inputs or outputs of the agent but would act as a central processing unit for the agent's input and output components. Such a component would essentially be the agent's brain.

These components would describe the first portion of the genome and would have originally been composed of a base set of segment genes which describe the other portion of the genome. Whenever complexity is added to one of the components' phenotypes through the Component SDNEAT algorithm, the segment would be stored in the list of unique segments and the segment gene would be added to the individual's genotype. If the innovation is not unique it would be treated the same way that NEAT and SDNEAT would treat a non-innovation. Essentially the NEAT algorithm would be further extended to support segments as an innovation. The definition of a segment would be extended to include a single link, making all innovations segments. Through this extension, components would be completely described by this new type of innovation at the highest level of abstraction, which would be segments.

The phenome, which would then be comprised of several highly connected but different specialized neural networks, could be encoded in a genome using only components and segments. Each component could store specialized information about the input it receives from the brain component or its set of inputs from the agent. The brain component could then find new patterns of complex behaviour based on substantially more input knowledge. Breaking up the highly complex single controller neural network into smaller component networks could have the added benefit of allowing the input and output networks to handle far more input and output nodes. Component input networks, for example, could be scaled up to handle optical information from a camera sensor and then feed that information to the brain network with a compressed pattern-matching output, rather than requiring the brain network to optimize optical information as well as laser range-finder information, GPS information and any other sensor input.

Mutation operations could remain the same in Component SDNEAT as in NEAT and SDNEAT, acting only on the segment genes. Crossover could then be defined as an operation on the components' phenomes. Input phenomes could crossover with other input phenomes and similarly, brain phenomes could crossover with other brain phenomes and output phenomes could crossover with other output phenomes. This process would superficially resemble the complex crossover process that occurs between biological cells. Components could even be directly tied to physical aspects of their agent. This could extend the Component SDNEAT algorithm to evolve the structure of its agent as well as the topological structure of its neural network controller.

Using Component SDNEAT, the complexity of the neural network solutions could be increased along with the potential for storing specialized information, without dramatically increasing the size of the genome. Segments could scale to this level of abstraction because they would not break the topological rules of neural networks and would still take full advantage of the historical markings introduced in NEAT. A Component SDNEAT algorithm could potentially scale to solve much more complex real-world problems than NEAT or SDNEAT alone.

5.4 Conclusion

NEAT and SDNEAT were first compared in order to resolve the question of whether or not the topological efficiency of their solutions to the XOR problem would be similar.

Experiments showed that SDNEAT evolved solutions that were as efficient in structure as those evolved by this implementation of NEAT. SDNEAT also found solutions in a shorter average time than NEAT. Further experimentation with the dynamic obstacle avoidance problem showed that SDNEAT evolved more high-fitness solutions than NEAT in the name number of experiments, as well as a higher-efficiency high-performance solution. SDNEAT's solution to the dynamic obstacle avoidance problem was the only solution to exhibit solution behaviour in all three environments. SDNEAT was also the only algorithm to evolve a solution to the maze environment.

NEAT is capable of evolving, from simple initial genomes, complex structures that solve complex problems. SDNEAT empowers NEAT to optimize these solutions much more efficiently through the use of segmental duplication, without losing any of the benefits of the original NEAT algorithm. SDNEAT evolves complex and nearly optimal solutions for the dynamic obstacle avoidance problem described in this thesis. SDNEAT can potentially be upgraded to Component SDNEAT, which could evolve complete environment-independent solutions to complex real-world problems. Future work could result in a complete solution to the dynamic obstacle avoidance problem.

Appendix A

EXPERIMENT PARAMETER VALUES

This appendix reviews the definition of the parameters used to modify the behaviour of the NEAT and SDNEAT algorithms. It also details the values used in each NEAT and SDNEAT experiment for both the XOR and dynamic obstacle avoidance problems.

A.1 Definitions

There are twenty-six neuroevolutionary system parameters for NEAT and twenty-eight for SDNEAT.

1. **Initial Population Size**: The number of individual genomes in the initial population in an experiment.
2. **Generations**: The number of generations the experiment should run for.
3. c_1: Coefficient modifying the importance of excess genes during distance calculation.
4. c_2: Coefficient modifying the importance of disjoint genes during distance calculation.
5. c_3: Coefficient modifying the importance of the average weight difference during distance calculation.
6. **Compatibility Threshold**: The distance required for a genome to be considered structurally different from another genome.
7. **Threshold Increment**: The amount the compatibility threshold is modified when no speciation is occurring. Induces speciation in lower-complexity populations.
8. **Max Number of Species**: Limits speciation to a maximum number of concurrent species.

9. **Young Bonus Threshold**: If a new species is below this number of generations its fitness is boosted by the young fitness bonus.
10. **Young Fitness Bonus**: The amount by which a new species' overall fitness is boosted when it is below the young bonus threshold.
11. **Old Age Threshold**: If a species is over this age and is not improving its fitness is penalized by the old age penalty.
12. **Old Age Penalty**: If a species is over the old age threshold, its fitness is punished by this amount.
13. **Survival Rate**: The percentage of the population to survive each generation.
14. **Probability Rate Replaced**: The probability a link weight is completely replaced by a new random weight.
15. **Max Weight Perturbation**: The maximum amount by which a weight will be mutated.
16. **Activation Mutation Rate**: The probability an activation function will be mutated.
17. **Max Activation Perturbation**: The maximum amount by which an activation function will be mutated.
18. **Genome Inputs**: The number of inputs in a genome.
19. **Genome Outputs**: The number of outputs in a genome.
20. **Number of Generations Allowed with No Improvement**: After a species reaches this number of generations, if it has not improved and it is not the species containing the genome with the population's highest fitness, the species is killed off.
21. **Crossover Rate**: The probability of crossover occurring.
22. **Max Number of Neurons**: The maximum number of neurons a genome is allowed to evolve.
23. **Mutation Rate**: The probability of mutation occurring.
24. **Chance to Add Node**: The probability of a node mutation occurring.

25. **Chance to Add Link**: The probability of a link mutation occurring.
26. **Chance of Looped Link**: The probability of a recurrent link mutation occurring.

These parameters are specific to the SDNEAT algorithm:

1. **SD Mutation Rate**: The probability of a segmental duplication mutation occurring.
2. **SD Sub-Mutation Rate**: The probability that a link or node mutation will occur in a segmental duplication.

A.2 Common Parameters

Several parameters were not changed between experiments in both NEAT and SDNEAT they are outlined in table A.1.

Parameter	XOR	Dynamic Obstacle Avoidance
c_1	1	1
c_2	1	1
c_3	0.4	0.4
Threshold Increment	0.05	0.05
Young Bonus Age Threshold	10	10
Young Fitness Bonus	1.3	1.3
Old Age Threshold	50	50
Old Age Penalty	0.7	0.7
Initial Genome Inputs	13	13
Initial Genome Outputs	2	2

Table A.1: Common parameter settings. These parameter values were used in every experiment.

A.3 Variable Parameters

Most parameters were varied between experiments. In XOR the experiments all used the low range value from Table A.2.

Parameter	Base	Low	High	Increment
Initial Population Size	100	50	200	50
Number of Generations	200	100	300	100
Initial Compatibility Threshold	0.5	0.2	0.5	0.1
Maximum Number of Species	30	20	30	5
Survival Rate	0.2	0.2	0.5	0.1
Probability Rate Replaced	0.1	0.1	0.3	0.05
Max Weight Perturbation	0.5	0.5	0.8	0.1
Activation Mutation Rate	0.1	0.1	0.3	0.1
Max Activation Perturbation	0.1	0.1	0.3	0.1
Number of Generations no improvement	15	15	30	5
Crossover Rate	0.07	0.1	0.3	0.1
Maximum Number of Neurons	25	25	50	5
Mutation Rate	0.3	0.2	0.4	0.1
Chance of Adding Node	0.04	0.01	0.1	0.01
Chance of Adding Link	0.07	0.04	0.14	0.2
Chance of Looped Link	0.05	0.03	0.11	0.2

Table A.2: Variable parameter settings. These parameter values were varied in individual experiments. The base value is the default when there is no variance, the low value is the bottom of the range of values while there is variance, high is the top of the range being varied and increment is the amount by which each is varied.

BIBLIOGRAPHY

Abbass, H. A. (2003). Speeding Up Backpropagation Using Multiobjective Evolutionary Algorithms. *Neural Computation* (15), 2705-2726.

Aguilar, J. M., & Jose, L. C.-V. (1994). Navite: A Neural Network System for Sensory Based Robot Navigation. *Proceedings of the World Congress in Neural Networks* .

Aitkenhead, M. J., & McDonald, A. J. (2002). A neural network based obstacle navigation animat in a virtual environment. *Engineering Applications of Artificial Intelligence* (15), 229-239.

Aliev, R. A., Fazlollahi, B., & Vahidov, R. M. (2001). Genetic Algorithm based learning of fuzzy neural networks. Part 1: feed forward fuzzy neural networks. *Fuzzy Sets and Systems* (118), 351-358.

Alsultanny, Y. A., & Aqel, M. M. (2003). Pattern recognition using multilayer neural genetic algorithm. *Neurocomputing* (51), 237-247.

Arifovic, J., & Gencay, R. (2001). Using genetic algorithms to select architecture of a feedforward artificial neural network. *Physica A* (289), 574-594.

Bailey, J. A., & Eichler, E. E. (2006). Primate segmental duplications: crucibles of evolution, diversity and disease. *Nature Reviews Genetics* (7), 552-564.

Blanco, A., Delgado, M., & Pegalajar, M. C. (2000). A genetic algorithm to obtain the optimal recurrent neural network. *International Journal of Approximate Reasoning* (23), 67-83.

Boozarjomehry, R. B., & Svrcek, W. Y. (2001). Automatic Design of Neural Network Structures. *Computers and Chemical Engineering* (25), 1075-1088.

Buckland, M., & Collins, M. (2002). NEAT. In M. Buckland, *AI Techniques for game programming.*

Capi, G., & Doya, K. (2005). Evolution of recurrent neural controllers using an extended parallel genetic algorithm. *Robotics and Autonomous Systems* , 148-159.

Castillo, P. A., Merelo, J. J., Prieto, A., Rivas, V., & Romero, G. (2000). G-Prop Global optimization of multilayer perceptrons using GAs. *Neurocomputing* (35), 149-163.

Cybenko, G. (1989). Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals and Systems*, 303-314.

Fernandez Leon, J. A., Tosini, M., Acosta, G. G., & Acosta, H. N. (2005). An experimental study on evolutionary reactive behaviors for mobile robots navigation. *Journal of Computer Science and Technology*, 183-188.

Floreano, D., & Mondada, F. (1994). Automatic Creation of Autonomous Agent: Genetic Evolution of a Neural-Network Driven Robot. *Proceedings of the Conference on Simulation of Adaptive Behavior*.

Floreano, D., & Mondada, F. (1996). Evolution of Homing Navigation in a Real Mobile Robot. *IEEE Transactions on Systems, Man, and Cybernetics-Part B* (26), 396-407.

Floreano, D., & Mondada, F. (1998). Evolutionary neurocontrollers for autonomous mobile robots. *Neural Networks* (11), 1461-1478.

Garcia-Pedrajas, N., Ortiz-Boyer, D., & Hervas-Martinez, C. (2006). An alternative approach for neural network evolution with a genetic algorithm: Crossover by combinatorial optimization. *Neural Networks*, 514-528.

Golubski, W., & Feuring, T. (1999). Evolving Neural Network Structures by Means of Genetic Programming. *Genetic Programming, Proceedings of EuorGP* (1598), 211-220.

Hughes, L., & Bredeche, N. (2007). *Simbad Project Home.* Retrieved from http://simbad.sourceforge.net/

Ilakovac, T. (1995). Adaptation of Neural Networks Using Genetic Algorithms. *Croatica Chemica Acta*, 29-38.

Janson, D. J., & Frenzel, J. F. (1993). Training Product Unit Neural Networks with Genetic Algorithms. *IEEE Expert*, 26-33.

Kassahun, Y., & Sommer, G. (2005). Automatic Neural Robot Controller Design using Evolutionary Acquisition of Neural Topologies. *Autonome Mobile Systeme*, 315-321.

Kitano, H. (1994). Neurogenetic learning: an integrated method of designing and training neural networks using genetic algorithms. *Physica D*, 225-238.

Kluge, B., Bank, D., & Prassler, E. (2002). Motion Coordination in Dynamic Environments: Reaching a Moving Goal while Avoiding Moving Obstacles. *IEEE Int. Workshop on Robot and Human Interactive Communication* .

Kluge, B., Illmann, J., & Prassler, E. (2001). Situation Assessment in Crowded Public Environments. *Proceedings of International Conference on Field and Service Robotics* .

Kluge, B., Kohler, C., & Prassler, E. (2001). Fast and Robust Tracking of Multiple Moving Objects with a Laser Range Finder. *Proceedings of IEEE International Conference on Robotics and Automation* .

Knoblock, C. (Ed.). (1996). Neural Networks in real-world applications. *IEEE Expert* , 4-12.

Koza, J. R. (1998). Genetic Programming. *Encyclopedia of Computer Science and Technology* .

Lee, M. (2003). Evolution of behaviors in autonomous robot using artificial neural network and genetic algorithm. *Information Sciences* (155), 43-60.

Miglino, O., Lund, H. H., & Nolfi, S. (1995). Evolving Mobile Robots in Simulated and Real Environments. *Artificial Life* (2), 417-434.

Mondada, F., & Floreano, D. (1995). Evolution of neural control structures: some experiments on mobile robots. *Robotics and Autonomous Systems* (16), 183-195.

Nelson, A. L., Grant, E., & Henderson, T. C. (2004). Evolution of neural controllers for competitive game playing with teams of mobile robots. *Robotics and Autonomous Systems* , 135-150.

Nelson, A. L., Grant, E., Galeotti, J. M., & Rhody, S. (2004). Maze exploration behaviors using an integrated evolutionary robotics environment. *Robotics and Autonomous Systems* (46), 159-173.

Neruda, R. (2007). Evolving neural network which control a robotic agent. *IEEE Congress on Evolutionary Computation* , 1517-1522.

Nissinen, A. S., Koivo, H. N., & Koivisto, H. (1999). Optimization of Neural Network Topologies Using Genetic Algorithm. *Intelligent Automation and Soft Computing* , 211-224.

Sato, Y., & Furuya, T. (1996). Coevolution in Recurrent Neural Networks Using Genetic Algorithms. *Systems and Computers in Japan* (27), 64-73.

Scheutz, M., Cserey, G., & McRaven, J. (2004). Fast, Reliable, Adaptive, Bimodal People Tracking for Indoor Environments. *IEEE International Conference on Intelligent Robots and Systems.*

Sexton, R. S., & Gupta, J. N. (2000). Comparative evaluation of genetic algorithm and backpropagation for training neural networks. *Information Sciences* (129), 45-59.

Sexton, R. S., Dorsey, R. E., & Sikander, N. A. (2004). Simultaneous optimization of neural network function and architecture algorithm. *Decision Support Systems* (36), 283-296.

Sharkey, N. E. (1997). The New Wave in Robot Leaning. *Robotics and Autonomous Systems* (22), 179-186.

Siebel, N. T., Krause, J., & Sommer, G. (2007). Efficient Learning of Neural Networks with Evolutionary Algorithms. *Lecture Notes in Computer Science Pattern Recognition* , 466-475.

Srinivas, M., & Patnaik, L. M. (1994). Genetic Algorithms: A Survey. *IEEE Transactions* , 17-26.

Stanley, K. O. (2004). *Efficient Evolution of Neural Networks through Complexification.* Austin: Department of Computer Sciences: The University of Texas at Austin.

Stanley, K. O., & Miikkulainen, R. (2002). Efficient Evolution of Neural Network Topologies. *Proceedings of the 2002 Congress on Evolutionary Computing* .

Stanley, K. O., & Miikkulainen, R. (2002). Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation* (10), 99-127.

Stanley, K. O., Bryant, B. D., & Miikkulainen, R. (2003). Evolving Adaptive Neural Networks with and without Adaptive Synapses. *Proceedings of the IEEE Congress on Evolutionary Computation* .

Tsukimoto, H., & Hatano, H. (2003). The functional localization of neural networks using genetic algorithms. *Neural Networks* (16), 55-67.

Tzafestas, S. G., Tzamtzi, M. P., & Rigatos, G. G. (2002). Robust motion planning and control of mobile robots for collision avoidance in terrains with moving objects. *Mathematics and Computers in Simulation* (59), 279-292.

Ward, K., & Zelinsky, A. (1997). Learning Mobile Robot Behaviours by Discovering Associations Between Input Vectors and Trajectory Velocities. *Tenth Australian Joint Conference on Artificial Intelligence* , 138-143.

Ward, K., Zelinsky, A., & McKerrow, P. (1999). Learning to Avoid Objects and Dock with a Mobile Robot. *Proceedings of the Australian Conference on Robotics and Automation* , 132-137.

Xu, F., Van Brussel, H., Nuttin, M., & Moreas, R. (2003). Concepts for dynamic obstacle avoidance and their extended application in underground navigation. *Robotics and Autonomous Systems* , 1-15.

Yao, X. (1999). Evolving Artificial Neural Networks. *Proceedings of the IEEE* , 1423-1447.

Zamparbelli, M. (1997). Genetically Trained Cellular Neural Networks. *Neural Networks* (10), 1143-1151.