Abstract

This thesis presents a further development of Neuroevolution of Augmenting topologies(NEAT)[21]. The author augments NEAT by parallelizing the fitness evaluation of the phenotypes enabling the method to be utilized on highly complex fitness evaluations by running it on a cluster. This augmented version of NEAT is then applied to the inherently complex problem of the Go board game, by using the Gnugo (**See** `www.gnu.org/software/gnugo/.`) software package as a fitness evaluator. The performance increase also enables the author to follow up on the predictions of Kenneth Stanley's previous discussions that co-evolution will help evolve a more general Go player, rather than the predicted evolved behaviour of specializing in beating Gnugo.

Preface

This thesis was done over the academic year of 2006/2007. The thesis is advised by Dr. Keith L. Downing. The last semester while writing the thesis I was also visiting the Evolutionary Complexity Research Group(EPLEX) at the University of Central Florida, where Dr. Kenneth O. Stanley and the students under him also contributed to advising me during my work.

Acknowledgements

I would like to thank my adviser Dr. Keith L. Downing for his wise words, his bravery in believing i could reach my goals, as well as his support and help with regards to my stay at the EPLEX group at UCF. I also owe great thanks to Dr. Kenneth O. Stanley for his research group hosting me, his help and the uncountable number of tricky questions he answered with great patience. I would also like to thank Boye A. Hoverstad for his excellent framework for running software tasks on a cluster, and his efforts to support my use of it. I would also like to mention my appreciation of the help I got, and all the inspiring discussions I had with my co lab workers; Anders Storsveen, Pavel Petrovic, David Lambrosi, Jeremiah Folsom-Kovarik, Jason J. Gauci , Phillip Verbancsics and Gautham Anil.

Contents

List of Figures **ix**

1 Introduction **3**
1.1 GO . . . 3
1.1.1 Rules of Go . . . 3
1.1.2 Example Game . . . 5
1.2 Previous Work . . . 8
1.2.1 SANE . . . 8
1.2.2 NEAT . . . 9
1.3 Motivation . . . 9
1.4 Working Hypothesis . . . 9
1.5 Overview of Thesis . . . 10

2 Background **11**
2.1 Artificial Neural Networks . . . 11
2.2 Genetic Algorithms . . . 12
2.3 Evolving Artificial Neural Networks . . . 15
2.3.1 Montana and Davis . . . 16
2.3.2 Kitano . . . 17
2.3.3 SANE . . . 18
2.3.4 ESP . . . 20
2.3.5 NEAT . . . 21
2.4 Computer Go Systems . . . 21
2.4.1 GNU Go . . . 22
2.4.2 The Many Faces of Go . . . 22
2.4.3 Mogo . . . 22

3 Problem **23**
3.1 Tractability . . . 23
3.2 Simultaneous Tactics . . . 24
3.3 Scaling With Board Size . . . 25

3.4 General Player 25

4 Method **27**
4.1 NEAT 27
4.1.1 NEAT Genotype 27
4.1.2 Speciation 30
4.1.3 Genetic Selection 33
4.1.4 Complexification 36
4.2 Roving Eye 36
4.3 GNU Go 40
4.3.1 Fitness Function 40
4.4 Co-evolution 41
4.5 Cluster Architecture 42

5 Results **45**
5.1 Performance Evaluation 46
5.1.1 Functional Performance 46
5.1.2 Run-time Performance 47
5.2 5x5 Board 48
5.2.1 Experimental Setup 48
5.2.2 Experimental Result 48
5.3 7x7 Board 49
5.3.1 Experimental Setup 50
5.3.2 Experimental Result 51

6 Analysis of Results **55**
6.1 Performance 55
6.2 Parallel Architecture 55
6.3 5x5 Board 56
6.4 7x7 Board 56
6.5 Co-evolution 57

7 Discussion **59**
7.1 Evaluation of Working Hypothesis 62
7.2 Future Work 63

Bibliography **65**

A Appendix **69**
A.1 Parameters 69
A.1.1 Fixed Parameters 74
A.1.2 Experiment Parameters 74

A.2 Source code 74

List of Figures

1.1 This figure shows a captured white stone 4
1.2 This figure shows one eye group that will get captured if white places piece in center. 5
1.3 This figure shows two eye group that will survive if someone puts a piece in one of the eyes, which makes that move illegal. 5
1.4 This figure exemplifies the Ko rule and the reasons for having it. In (a) black captures (2) the white piece (1). In (b) white has to move at another position (3) to change the board state to capture the black stone set in the second move in (a). If white didn't move to (3) before, move (4) would be illegal per the Ko rule. 6
1.5 This figure shows the early moves of the example game(source britgo.org). 6
1.6 This figure shows the moves of mid-game during the example game(source britgo.org). 7
1.7 This figure shows the end state of the example game, here you can observe that white has lost the large group it had inside the black territory. Even though the group is lost white ends up having more territory. This is a good example of how material losses is but a temporary consideration(source britgo.org). . . 7
1.8 This figure shows the board ready for territory counting(source britgo.org). 8

2.1 This figure shows a sigmoidal transfer-function with $k = -4.9$, X-axis represents the input to the node, and the Y-axis represents the output of the node. 12
2.2 This figure shows a typical feed-forward network. 13
2.3 This figure shows a network with a recurrent link from the output node to the middle input node. 13
2.4 This figure shows a SANE genome, and the network generated from the genome. Based on a figure from [7] 19

4.1 This figure depicts a NEAT genotype containing the node list and the genes representing the connections between them. The node genes are depicted for readability, but in essence they could be derived from the connection genes. Based on a figure from [21]. 28

4.2 This figure depicts a NEAT genotype and the results when one mutates it by adding a connection and a node respectively. Based on a figure from [21]. 29

4.3 This figure depicts a NEAT genotype and the results when one mutates it by adding a node and keeping the link. Based in a figure from [21]. 30

4.4 This figure depicts two NEAT genotypes and their networks, and then how they are aligned for crossover, and finally the resulting genotype. Based on a figure from [21]. 32

4.5 This figure depicts a speciation. Here you can see at generation 0 the number of species is 1. In generation 4(A) you can see the advent of a new species. This species grows and can be seen more clearly in generation 5(in green). The same species can be seen almost extinct in generation 8(B) however. The number of species also converge towards the set target number of species, which in this case is 20. 34

4.6 This figure depicts an example where a roving eye would have an advantage of being able to turn around to change its orientation. This figure shows two go shapes, shape A and shape B. When the roving eye, depicted by R, can turn around, representing the two as is easier for a ANN. 37

4.7 This figure depicts the typical input of a roving eye. Based on a figure from [21]. The white circles depict the direct inputs to the roving eye for each of the NxN intersections(in this case 3x3), while the circles marked with a L depicts the long range inputs. 38

4.8 This figure depicts the typical starting network of a roving eye. Where the circles in the inner 3x3 represents a typical 3x3 roving eye input area, while the outer four circles labeled L is the long range sensors. Based on a figure from [21]. 39

4.9 This figure depicts the distributed architecture. 43

5.1 In this figure, the green nodes are input nodes with a linear transfer-function(in this case f(x) = x), the red node is a bias node that always outputs 1 to the node it is connected to and the yellow node is regular node with a sigmoid transfer-function. (a) Shows the starting point, or seed, of the evolution. (b) Shows the minimal solution for XOR. 47

5.2 This figure shows the fitness of NEAT evolving against GNU Go over 500 generations with a population of 400 with fitness function given in equation (4.6). 49

5.3 Here is a game played against GNU Go in mid-evolution, at generation 150. The network starts out well but spreads out too much with move 3. It responds well with moves 7 and 5, but it stops short as it has not learned to play the game to its end. This was evolved using the fitness function given in equation (4.6). 49

5.4 This figure shows the fitness of NEAT evolving against GNU Go over 700 generations with a population of 400 with fitness function given in equation (4.6). 50

5.5 Here is a game played against GNU Go by the resulting champion of a ended evolution. The ANN starts to build a line with move 1 and 3, but is interrupted with white's move 2 and 4. This was evolved using fitness function given in equation (4.6). 50

5.6 This figure shows the fitness of NEAT evolving against GNU Go over 500 generations with a population of 400 with the alternative the fitness function given in equation (4.7). In this fitness function the roving eye is awarded for every stone it puts on the board. 51

5.7 This figure shows a game played against GNU Go in mid-evolution, at generation500. This was evolved using the alternative fitness function given in equation (4.7). The network evolved here performs better than the game shown as a result of evolution with fitness function given in equation (4.6). The move marked A represents both move 1 and move number 13. 51

5.8 This graph shows an evolution where co-evolution was started at generation 500 and continued until generation 1000. 52

5.9 In this graph the evolution was started from scratch on a 7x7 board space. This was evolved using the fitness function given in equation (4.6). The game played by the resulting champion can be seen in figure 5.10. 52

5.10 In this figure the evolution was started from scratch on a 7x7 board. This was evolved using fitness function given in equation (4.6). 53
5.11 This figure shows a game from a champion evolved on a 7x7 board seeded with a 5x5 champion as seen in the 5x5 board simulation reported in figure 5.5. 53
5.12 This graph shows an evolution on a 7x7 board using the alternative fitness function given in equation (4.7). The game played by the champion of generation 500 can be seen in figure 5.13. 54
5.13 This figure shows a game of the champion evolved using the alternative fitness function given in equation (4.7). The roving eye plays well but seems to stop playing after move 11. It later plays move 18 to little effect on the outcome of the game. . . . 54

7.1 These two games depicted in this figure both show strong symmetrical properties as to the series of moves made by the roving eye. In figure both figures (a) and (b) the symmetry seems to be around the central vertical axis from top to bottom. Both patterns are highly symmetrical with the one exception of move 9 in figure (b). 60
7.2 This figure show a simple winning strategy against GNU Go when playing with no handicap and a komi of 0.5. 61

Chapter 1

Introduction

This paper will try to outline a new approach to the problem of Go, and the challenge Go poses to current AI methods and game solving approaches. To understand this challenge and the inherent complexity of this board game one needs a overview of the rules and the complex game patterns that follow.

1.1 GO

The ancient board game originating in ancient China[1] named Go is a two player strategic board game. It is usually played on a 19x19 rectangular grid, with black stones for the first player and white stones for the second player.

Go also differs from many other strategic board games in that it is exceptionally hard to make a computer excel in the game. This is mainly because of the combination of a large default board(19x19 compared to 8x8 in chess), and the simple rules(only one type of piece), together make an alpha-beta search as applied in modern chess engines intractable.

1.1.1 Rules of Go

The object of the game is to capture territory with your stones. You place the stones on the intersections, and the stone has liberties, one for each

[1]First written mention of it is 548 BC, but thought to be 4000 year old.

unoccupied intersection directly[2] adjacent to the stone. These liberties can be removed by the opposing player placing stones at the adjacent intersections. If a stone looses all its liberties, as seen in figure 1.1 it is captured. A stone

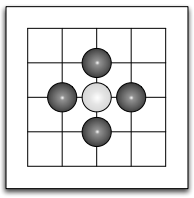


Figure 1.1: This figure shows a captured white stone

can also can also connect to other stones of the same color if it placed in a intersection directly adjacent, the stones now form a group and they share their liberties. Such that a two stone group with all their liberties free in the middle of a board will have six liberties as opposed to the maximum of four liberties of a single stone. Grouping makes it harder to capture your stones and territory formed by these stones. Regardless, most groups can be captured if the opposing player just plays his stones well enough but there are groups of stones that can't be captured. In addition, a stone is not allowed to be placed in a intersection with no liberties, except when that move would remove the last liberty of the group surrounding the intersection[3].

Ko is another important rule which states that one cannot play a move that would recreate the exact same board state that the last move the same player did resulted in. Ko situations are central to the problem of parallel tactics in a go game, and thus central to one of the biggest problem facing computer go. An example of this rule can be seen in figure 1.4.

A game can also have a handicap if the two players are not at the same level of play, the handicap is a predetermined number of stones placed out by the weaker player.

The game ends when both players pass. This usually happens when both players see no other moves they can make that will increase their score.

[2]Diagonally adjacent intersections does not constrain liberties as they are not directly connected to the mentioned intersection by a line.

[3]one such example can be seen in figure 1.2.

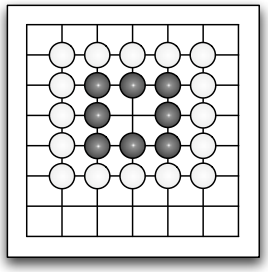


Figure 1.2: This figure shows one eye group that will get captured if white places piece in center.

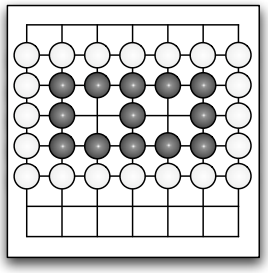
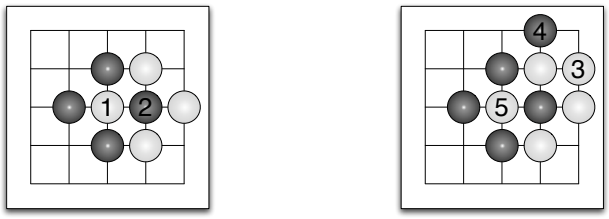


Figure 1.3: This figure shows two eye group that will survive if someone puts a piece in one of the eyes, which makes that move illegal.

The game can also be won by resignation if one of the players resign when evaluating the current board as lost. When the game is over the number of captured opponent stones and the number of enclosing empty intersections enclosed by the players stones are counted. If the game is even and no handicap is used, a pre-decided amount of points called "Komi" is added to the score of the white player. This value is variable to the board size, and is supposed to balance back the fact that black has an advantage by starting.

1.1.2 Example Game

To give a short example of how the game is played, four snapshots of board are given, one of the pregame and one of the mid game and a final board from the final board. This game is played on a 9x9 board to ease the understanding



(a) Ko pattern start (b) Ko result.

Figure 1.4: This figure exemplifies the Ko rule and the reasons for having it. In (a) black captures (2) the white piece (1). In (b) white has to move at another position (3) to change the board state to capture the black stone set in the second move in (a). If white didn't move to (3) before, move (4) would be illegal per the Ko rule.

of the game tactics(they grow more complex as the board size increases). In the preliminary moves seen in figure 1.5 the two players try each other out and aim to "mark their territories". As seen in the figure the black player maneuvers to take the lower right corner. White responds by surrounding his position, and seems to end up with a less firm grip on a larger territory.

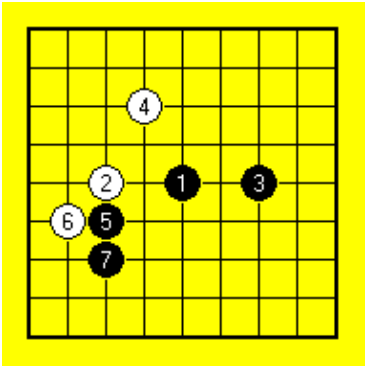


Figure 1.5: This figure shows the early moves of the example game(source britgo.org).

In figure 1.6, the white player moves in on the weakly defended upper right quadrant of the board by placing move number 40. Move 40 strengthens the two white pieces by making it impossible for black to "cut" the structure by placing a black piece at the same spot. Black responds by intercepting in

move 41 and 43, and strengthens his wall with move 45.

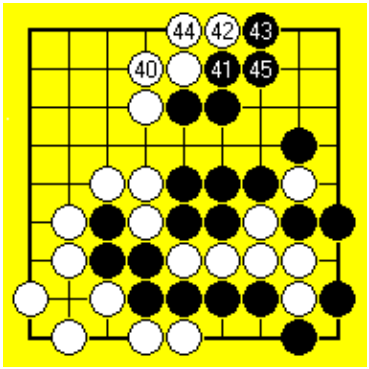


Figure 1.6: This figure shows the moves of mid-game during the example game(source britgo.org).

At this point the game is almost over, with a few points to win or lose. Figure 1.7 shows the final board. At this point every neutral position that can be filled gets filled, and the players connected any unconnected group. The pieces are then rearranged so that is easier to count the territories, usually so that you have a set of rectangles to count as seen in figure 1.8.

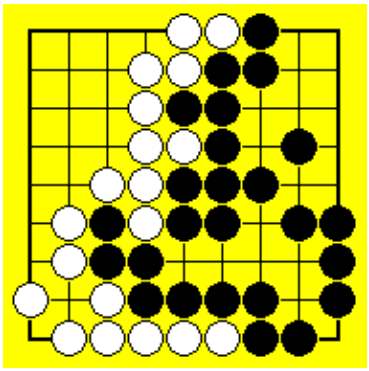


Figure 1.7: This figure shows the end state of the example game, here you can observe that white has lost the large group it had inside the black territory. Even though the group is lost white ends up having more territory. This is a good example of how material losses is but a temporary consideration(source britgo.org).

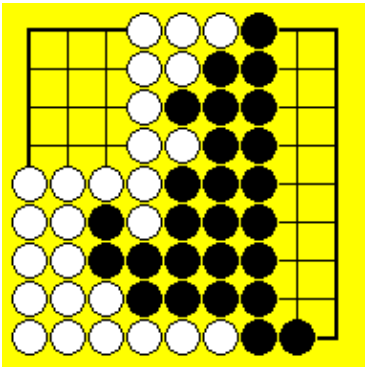


Figure 1.8: This figure shows the board ready for territory counting(source britgo.org).

1.2 Previous Work

There has been done previous work in the direction of using biologically inspired methods to attack the problem of playing go with computers. This section will give a short introduction to some of the earlier approaches which have similar characteristics, namely that it employs artificial neural networks and evolution for searching the solution space.

1.2.1 SANE

SANE is a system developed by Risto Mikkulainen, which he later applied to the problem of GO[18]. SANE was matched up against Wally, a simple go engine, generally regarded as weaker than GNU Go. SANE did beat wally quite fast on small boards. Although it only took SANE 260 generations to beat wally at 9x9, it took 5 days in CPU time. This result originates from 1998, Moore's law suggests that Mikkulainen had 2^9 less computing power the current computers[4], but since 1998 the GO engines has grown stronger, and way more power hungry. And there is little detail as to how good the wally engine was in relation to humans, so it is hard to say what SANE could have done with todays computing power.

[4]Per date, 1st of May 2007

1.2.2 NEAT

Kenneth O. Stanley has developed the NEAT[21] system for evolving neural networks, both weights and topology. In chapter seven of the dissertation[21] outlying this system, he describes how he uses NEAT to play go. He achieves pretty good results on 5x5 and 7x7 size boards, and suggests for further work that co-evolution may give the ANNs(Artificial Neural Networks) a more general sense of the play, which may improve their game even further. This neuro-evolutionary system will be described in more detail in section 2.3.5 and in depth in chapter 4.

1.3 Motivation

As Go stands as one of the final board games to be conquered by artificial intelligence or computer science, it is a tempting target to aim for, but it must be approached incrementally. So far computers have shown promise in small board sizes up to 9x9. The motivation for this thesis is to test whether increasing evolutionary parameter's as population size and number of generations, will show results when attacking bigger boards than 7x7. Finally the biggest motivation is to try out the effect of co-evolution on the play style of the neural networks.

1.4 Working Hypothesis

Hypothesis I:*Neural networks evolved in NEAT against GNU Go, and with* ***co-evolution*** *to play GO, will achieve a higher level of generality, and will adapt more general playing patterns than a network evolved against GNU Go alone.*

1.5 Overview of Thesis

This thesis is divided into 7 chapters, in chapter 1 I start by introducing the problem of computer go and the previous work towards solving this. Chapter 2 describes the background to my approach, the main methods underlying the NEAT method and the contributions made by this thesis. In chapter 3 the thesis gives more insight to the complex problem of computer go. The main efforts to confirm the working hypothesis of the thesis is described in chapter 4. The results of the experiments derived from chapter 4 is presented in chapter 5, with analysis and discussion in chapter 6 and 7 respectively.

Chapter 2

Background

This chapter will further introduce the concepts and methods used in this thesis, and give a broader foundation on which to understand the methods later introduced.

2.1 Artificial Neural Networks

Artificial Neural Networks(ANNs) are an AI abstraction capable of simplistically mimicking the neural structures in our brain. The networks consists of nodes and the connection between them. The nodes operate by reading all their inputs from their incoming connections, enumerating the inputs, and inputting this sum into the activation function of the node. The node then sends the output of the activation function to all its outgoing connections. The connections between the nodes simply propagate the value from one node to the other, but it also multiplies the propagating value with a weight, given as:

$$n_i = \sigma\left(\sum_i w_{ij} * x_i\right), x_i \in X$$

For node n_i , the activation function σ and the set X of nodes with connections leading into n_i.

Changing one of more of these weights can change the output of a network. This is the main source of learning in many ANN systems, mainly

feed-forward networks with the back-propagation gradient descent learning rule. Feed-forward networks generally employ a static network topology. Other ANN learning systems also change the topology of the network during learning, but this also opens up a seemingly unlimited number of new search parameters, making the search harder. This problem will be addressed in chapter 4. The activation function as most often a variation of a sigmoid function, as it mimics the activation firing pattern seen in neurons. Input nodes, nodes taking its inputs directly from sensors or data, typically use a straight transfer function:

$$f(input) = input$$

This helps it not to deteriorate the signal before it enters the network. A much more widely used activation function, namely the sigmoidal activation can be described as:

$$sgm(input) = \frac{1}{1 + e^{(k*input)}}$$

Where k is a constant deciding how steep the sigmoidal curve is. A typical curve can be shown below in figure 2.1.

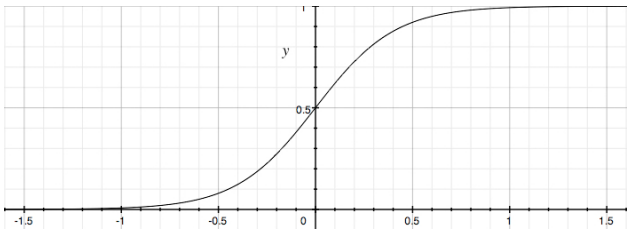


Figure 2.1: This figure shows a sigmoidal transfer-function with $k = -4.9$, X-axis represents the input to the node, and the Y-axis represents the output of the node.

2.2 Genetic Algorithms

Genetic algorithms[17, 11, 13, 12] provide a functional analogy to Darwinian evolution, albeit usually evolution against a set specific target. As nature

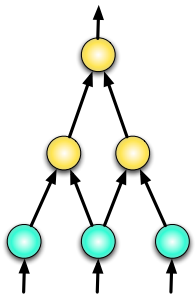


Figure 2.2: This figure shows a typical feed-forward network.

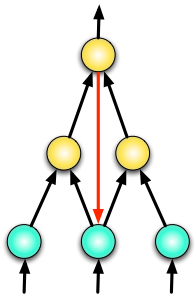


Figure 2.3: This figure shows a network with a recurrent link from the output node to the middle input node.

restricts a genome from procreating if it less fit than the others in its generation, an evolutionary algorithm will restrict bad genotypes if they are unfit in the eyes of the given fitness function. The development from genotype to phenotype, which can be tested by the fitness function, can be as small as taking the numbers from a numeric genome to do one calculation and return the inverse of the error offset, to using interpretation rules to rewrite the genome thousands of times over time to finally be interpreted by the fitness function. One big advantage with genetic algorithms is that they require very little domain knowledge of the task at hand, although one would need domain knowledge to understand the semantics of the genetic operators mentioned below, this is not the case for many genetic encodings.

Genetic encoding is an important aspect of an genetic algorithm as it represents the syntax of the genetic algorithm and thus effectively also the

boundaries of what the genetic algorithm can express through it. You can also discriminate the genetic encodings on how much development there is between the genotype and the phenotype, in the case of a long developmental process it can be hard to judge the expressability of the genetic encoding by the syntax itself but one has to take into account the semantics in relation to the syntax that the developmental process represents.

Mutation is the genetic operator that changes the genotypes through generations, in analogy with nature, the algorithms employ mutations. These mutations are locally dependent on the structure of the genotype. The structure of the genotype is highly dependent on the task to be solved, e.g a fitting genotype to solve the traveling sales person problem could simply be a ordered list of the cities, whereas a genotype to solve XOR could be a evolutionary programming tree or a list of genomes describing the weights in a fixed topology feed forward artificial neural network. As the structure depends on the task, so does the implementation of the mutations, in the case of neural networks it could be to shift one of more of the genomes describing the weights in the network by a randomly picked value. In the case of the traveling sales person problem it could simply be to switch two of the cities in the list. The number of parameters available to the evolutionary algorithm to mutate is in addition to the structure itself also dependent on the mutations themselves as they can make the genome bigger or smaller, thus respectively increasing or decreasing the search space the genetic algorithm searches. An example of this could be a genetic algorithm evolving a neural network that gradually added structure to the network, thus increasing the number of weights that the weight mutation could mutate. In that way mutation can in some cases be thought of as a semantic operator that changes its own syntax.

Crossover is the functional analogy to nature's meiosis. Not every genetic algorithm employs crossover, although most do, as it strengthens the biological analogy to Darwinian evolution, however it could be argued that a simple genetic clone with mutation could represent asexual procreation which still is supported by evolution in nature. Crossover is usually implemented in relation to the theories in biology, but often with some functional twists to pair the crossover with the semantics of the task. One usual way to im-

plement it is to take two parent genomes, align them horizontally and pick a single point along the genomes, then cut them in half and chose the "first" part from the first parent and the "second" from the second parent. One variation to this it to do the same thing only with N cut points, a popular sub-variation of this is when N equals the number of genes in the genomes; you simply choose each gene randomly from the parent's genomes. There are also other variations such as averaging the gene values between the parents. Crossover is a genetic operator which again is highly dependent on the genetic encoding being used, and both the syntax and the semantic of the genetic encoding should be considered when implementing. An example of this would be the highly problematic area of using crossover on two genomes representing two neural networks with two different topologies. If you don't consider the semantics of this operation you will most likely end up with a dysfunctional neural network as a end product, thus to make your crossover operator work in a less destructive manner in this case, you need to consider the semantic aspects of the operation.

Co-evolution is the version of a genetic algorithm where the fitness of the phenotypes are not only dependent on a deterministic fitness evaluator, but each other. There are several versions of co-evolution but the most known would be the direct co-evolution, where phenotypes are matched up against each other to compete for fitness in an almost tournament like style. Another version evolves two or more populations which use each other to solve tasks together in a symbiotic fashion. An example of th latter will be presented in section 2.3.4.

2.3 Evolving Artificial Neural Networks

Artificial neural networks have been popular targets for genetic algorithms, quite naturally as it does combine two good natural analogies which both have shown good results. It is also an enticing approach to Artificial Intelligence as it appears as a natural analogy to the same processes that resulted in human intelligence. This approach does however also harbour some roadblocks on our path to AI.

ANNs can mainly be evolved in two ways. The first is to evolve the weights of a fixed topology network, giving you a fitness landscape in a fixed amount of dimensions. This has the drawback that you would have to guess which topology would fit the task at hand before starting evolution, thus removing the advantage of not involving domain knowledge in the genetic algorithm. In return, fixed topology evolution is much faster, but could in essence be replaced by back-propagation in most cases.

The second way to evolve ANNs to evolving the topology in addition to the weights expands the fitness landscape exponentially, as mentioned above in section 2.2, it also lets you attack problems with yet unknown solutions. The problem with evolving the topology is applying a useful and non-restrictive heuristic to the evolution of topology to optimize the tractability of the method.

Next the paper will try to introduce some of the concepts and ideas in neuroevolution through examples of previous systems.

2.3.1 Montana and Davis

Montana and Davis was one of the first to apply genetic algorithms to ANNs[3]. Their task was a supervised pattern matching task, their system had to classify sonar data. The topology of the networks used was the same across the population and fixed before runs. The genome of this system consisted of a list of real numbers, representing the weights of this feed forward ANN. The evolutionary operators applied consisted of one mutation operator and one crossover operator. The mutational operator simply shifted the real numbers by small increments in negative or positive direction. The crossover operator operated on a set of atomic units within the genome, each unit consisted of the weights of all the incoming links to one node. In this way the crossover operator had a bigger chance of being constructors and the chances of it having a destructive influence on the offspring decreases.

At the time their system performed beyond standard back-propagation. The system was outperformed by newer optimized versions of back-propagation. They concluded that their system did perform well in supervised situations

but had its niche in unsupervised learning situations.

2.3.2 Kitano

As an early contributor to the neuro-evolution field Kitano[14] was one of the first to try a indirect encoding of the genome representing the network. In the previous year there had been introduced several systems that evolved neural networks with genetic algorithms, but using direct encoding[20, 9]. Kitano recognized and demonstrated the scalability problems associated with the direct approach when you want to represent bigger networks.

Kitano suggested a new approach to genetic encoding, this new method uses grammar to create rules that produce a connectivity matrix for the networks. He applies a modified version of the L-system[16, 15], which he augments to be context free and deterministic for experimental reasons. A rule is defined by the left hand symbol and its corresponding matrix e.g graph generation rules used for generation of the 2-2-1 XOR network[14]:

$$S \rightarrow \begin{bmatrix} A & B \\ C & D \end{bmatrix} A \rightarrow \begin{bmatrix} c & p \\ a & c \end{bmatrix} B \rightarrow \begin{bmatrix} a & a \\ a & e \end{bmatrix} C \rightarrow \begin{bmatrix} a & a \\ a & a \end{bmatrix} D \rightarrow \begin{bmatrix} a & a \\ a & d \end{bmatrix}$$

$$a \rightarrow \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} b \rightarrow \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} c \rightarrow \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} e \rightarrow \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} p \rightarrow \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

This would then be used on the initial state S generating the following:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \rightarrow \begin{bmatrix} c & p & a & a \\ a & c & a & e \\ a & a & a & a \\ a & a & a & b \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

This final matrix can then be translated to a network. The chromosome or genotype in this system consists of a variable and a constant part. The constant part contains the starting rule symbol and the left hand symbol between "a" and "p" and the corresponding binary pattern for the right hand side. This system is then tested on the 4x4 and 8x8 encoder/decoder problem with population sizes 10-100. The results clearly showed faster GA convergence and better ANN performance than direct encoding. The results also showed that the indirect approach was less sensitive to network size.

There is however several assumptions and shortcomings in the article mentioned. One assumption is that ANNs with more regular patterns are better, as this regularity is derived from the method. The article also only did tests on one type of problem and using a extremely small population over a small number of generations.

2.3.3 SANE

SANE(Symbiotic, Adaptive Neuro-Evolution)[7] is Neuro-Evolution algorithm that evolves neurons. The genome of the neuron consists of a series of connection definitions. A connection definition is defined by a 8-bit label and a 16-bit weight. The labels define to which output neuron the connection is, or the input neuron which the connections comes from. To achieve a more compact encoding the author used a modulo encoding of the labels. Thus the interpretation of a label is defined in the following way: For a label D, if $D > 127$ the connection made to output node $N = D \bmod O$ where O is the total number of output nodes. If $D \leq 127$, connection made to the input node $N = D \bmod I$ where I is the total number of input nodes.

A generational epoch of the SANE system is defined in the following way:

1. Pick a random subset of size ς and combined to form a neural network.

2. Evaluate the network against the task at hand.

3. Assign the score of the network to the neurons that participated.

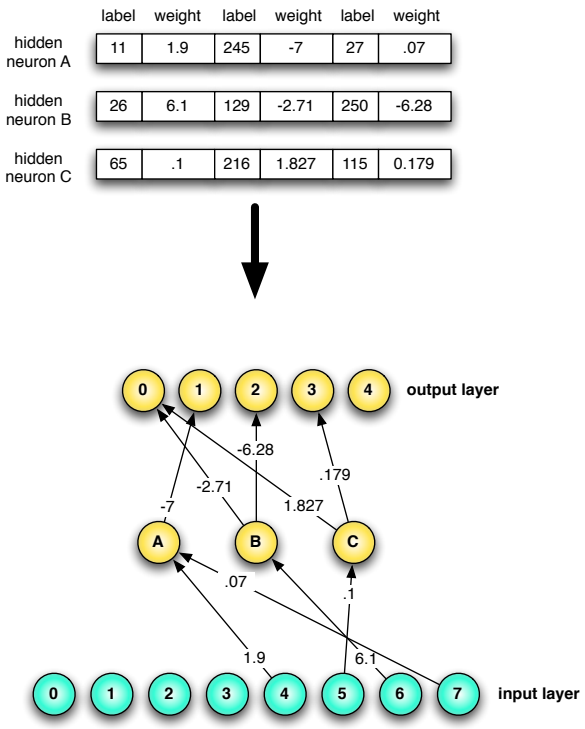


Figure 2.4: This figure shows a SANE genome, and the network generated from the genome. Based on a figure from [7]

4. Repeat previous steps sufficient[1] times

5. Average the fitness of the neurons by dividing their fitness by the number of networks they have participated in.

6. Perform crossover on the neurons based on the fitness assign the previous step.

SANE performs reasonably well, and is compared favourably to the earlier systems among others GENITOR[5, 4]. However this implementation of SANE also suffers from the fact that it can't represent recurrent connections. SANE is also outperformed later by NEAT[21] and it's own successor ESP.

In a later revision of SANE the author augments the method by introducing a separate population of ANN blueprints. The ANN blueprints is a

[1] A parameter, depends on the task at hand.

structure containing pointers to the neurons in the neuron population. The blueprints using the neurons describes a functioning network, which is evaluated, thus the neurons can contribute to different blueprints. The performance of the blueprints affects the neurons fitness in the same way that the randomly constructed networks in the original way did. In the augmented version the performance is also assigned to the blueprints, which is mutated and recombined in a similar way to the neurons. This effectively gives a symbiotic co-evolution between the blueprints and the neurons.

2.3.4 ESP

ESP(Enforced Sub-Populations)[8] is a modification of SANE. As in SANE the population consists of neurons and not full networks, and the networks is built from choosing from there neurons. However ESP uses a separate population of neurons for each of the P positions in the network that can be evolved. In ESP a neuron can only be recombined with a member of its own sub-population. This allows for specialization within each sub-population for each of the positions in the network. This had a tendency to happen[8] in SANE, only it was indirect and the sub-population had to self-organize within the larger population. These sub-populations avoids the many cases of redundancy often observed with SANE, as one neuron needs only specialize in one "role" and not all the possible roles. This also makes the evaluation of the neurons more consistent, since the neurons, regardless of sub-population is bound to end up collaborating with neurons that also specializes in the position they are in. This could also happen in SANE, but in the cases they did happen, it was mere coincidence. This also enables ESP to support recurrent connections, which is regarded as one of ESP's main contributions. SANE could not support neurons with recurrent connections as these neurons were highly dependent on the neurons around it that it connected to, and in SANE this was to inconsistent for reliable performance.

ESP outperforms it's predecessor SANE and performs on par[2] with NEAT in several tests. ESP and SANE both suffers from the fact that the systems

[2]The difference between NEAT and ESP is insignificant $p<0.001$.[21]

are fixed topology and depends on a expert setting up the topology in advance of the evolution.

2.3.5 NEAT

Neuro Evolution of augmenting topologies(NEAT)[21], developed by Kenneth O. Stanley is one of the more successful methods for evolving weights and topology. NEAT starts out with a minimal structure with one of two starting points;

- A given genome seed which spawns evolution(all offspring it mutated off of this genome)
- A spawn, where you specify the number of input, output and hidden nodes and the ratio of which there should be generated connections between them, these connections will have random weights. This will then spawn a diverse starting population.

With NEAT starting out at a minimal structure and slowly expanding will bias the system towards smaller solutions which makes the system more tractable. NEAT also speciates its networks, so that similar networks share fitness. NEAT also penalizes species collectively if it has not gotten a better result in a given amount of generations. Both these features helps NEAT progress to bigger structures, as newly added structure often tends to lower the fitness the fitness will be shared in the species it evolved from. If the structural mutation made the network create its own species, it will have a small grace period for tuning the weight(s) of the new structure, as NEAT penalizes old species.

2.4 Computer Go Systems

This section aims to provide a short overview of other important Go systems and their impact and methods.

2.4.1 GNU Go

GNU Go[3] is the standard among free and open source Go playing computer programs. Version 1.1 was first posted to comp.sources.games on March 13th 1989. The current stable version 3.6 was released 17th of march 2004, and the development branch, version 3.7 is enjoying development. Both of the two newest releases is used in several front-ends and several bots can be seen playing on Go playing networks such as KGS. GNU Go is built upon several methods of analyzing the board and groups of stones symbolically. GNU Go is also used in a shell program that implements global search in GNU Go called "Slug Go"[4] which is run a cluster of Apple X-serve computers.

2.4.2 The Many Faces of Go

The many faces of go is a program developed by David Fotland [5]. It employs many of the same methods analyzing the board used in GNU Go. It is the 2002 and 1998 "World Computer Go Champion", and ten time US "Computer Go Champion". It has many features often missing from other Go programs, such as Joseki dictionary[6]. It also contains a large base of Fuseki[7] moves.

2.4.3 Mogo

MoGo[23] uses version of Monte Carlo simulation to calculate the utility of each possible move from a given position, it is fast and highly parametrized as to how deep it should search, in addition it uses several heuristics to augment the search on bigger boards(19x19). It has however been showed to be weak against certain strategies, but is currently regarded as one of the most promising approaches to playing computer GO on bigger boards than 9x9. It is widely regarded as a top level go program at 9x9 and 13x13.

[3]See `http://www.gnu.org/software/GNUGo/`

[4]See `http://sluggo.dforge.cse.ucsc.edu/`

[5]See `http://www.smart-games.com/manyfaces.html`

[6]Joseki means a sequence of moves which results in a fair outcome for both black and white player.

[7]Go opening moves.

Chapter 3

Problem

The problem of making a computer play Go is one composed of several important sub-problems; the large search space, the way local go tactics interact in a larger go strategy and a more technical question of how you make a computer scale well against an increased board size. This chapter will try to describe these problems and introduce them for the next chapter in which the methods for attacking them will be detailed.

3.1 Tractability

Go is one of the bigger board games around, both in shear board size and in the number of moves made in a game. There are many reasons for this game being complex:

- The board is large, usually a board with 19x19 lines, 361 intersections and legal positions.
- The number of legal moves is on average 220 and rarely goes below 50, compared to chess where the average is 37
- The pieces don't disappear in the way that pieces can be killed of in most other strategic games like chess. The game grows more complex as new pieces appear with every move made.

- Material gain, e.g. a player having more officers in chess, does not mean a sure victory, but rather a short term lead.
- The degree of pattern matching in the game, and the ability of humans to excel in this is a big factor in making it easier for humans to play the game. In turn this is a disadvantage for most computer methods which seldom excel in pattern matching involving such complex patterns involved in a go game.

The game is normally very long with around 200 moves in a average game. This is a game that require patience for humans. The simplistic rules, and the fact that the game only has two types of pieces yours and your opponents, gives the game an almost unmatched branching factor when considering the moves as a game tree . This branching factor combined with the previously mentioned dimensions of the game, creates a unimaginable number of possible games. A common comparison points out that the number of possible games in Go is greater than our current estimates of number of particles in the known universe. This fact suggests that a brute force attack using a tree search would be impossible in the foreseeable future given that Moore's law still holds. So other methods needs to be addressed, heavy pruning of a game tree with domain knowledge based heuristics. It is however quite clear that humans, without having every possible game or move searched, manage to play the game at a reasonable level. Thus the idea of applying biologically inspired methods, more specific neuroevolution, is appealing to researchers trying to build a method for playing go.

3.2 Simultaneous Tactics

Another prominent feature of go is the presence of several small battles going on in parallel on the board, often in the early and middle stages of the game. This is hardly a problem for a Go player, but for a computer trying to concentrate on analysing one battle while keeping the other close in memory, may serve to be quite a challenge. This is necessary, as your choices in the current battle may very well affect the other battle, and thus the current

battle cant be analyzed by itself. The Ko rule described in section 1.1 is a good example of this, as one battle played on one part of the board while having a Ko situation on some other part of the board may very well mean the life or death of a group depending on how the Ko situation will be resolved.

3.3 Scaling With Board Size

There is also a question as to how make the computer best handle an increase of board size. As we can hardly claim that a human being see a whole 19 x 19 board at one glance, but more likely we sense some central focal point and a more diffused view of the peripheral parts of the board outside the focal point. To compensate for our less than perfect vision we move our eyes to scan over the board.

Many of the current go-playing systems using ANNs faces a problem relating this scaling property. ANNs usually receive their input from the whole board, and train on a certain board size. Increasing the board size in that case means one of two things. The first being that the ANN can't see the whole board as the input area is smaller than the actual board. This would result in the ANN taking decisions based in incomplete information. And even worse the ANN could possibly not place pieces on the whole board, somewhat depending on the implementation. The other outcome of increasing the board size would be to increase the number of inputs to the network. In turn this would also result in the network being retrained as the weights of the smaller network would make little or no sense on a bigger network. This would mean effectively more than doubling the time of training(as a larger input base usually means more weights to train).

3.4 General Player

One aim of computer go is to make general player, which cannot easily be fooled by exotic strategies, or strategies simply designed to the computer program in question. This happened to early chess playing engines where top-

players learned to play “anti-computer” chess. Developers of chess engines in turn responded by incorporating “anti-anti-computer” chess, which countered the erratic moves of the anti-computer tactics.

This is also a problem facing AI methods that need supervised training, as they often develop systems that train against current go-engines. This makes the supervised methods specialize in beating the go-engine it trains against. This in turn seldom pays off in the long run as you would like to develop a method for playing go not only against the training engine but also other engines or even human opponents.

Chapter 4

Method

This chapter will try to detail the methods underlying the system developed to explore the questions outlined in chapter 3.

4.1 NEAT

As previously explained in section 1.2.2 and 2.3.5, NEAT is a neuroevolution method that takes advantage of a genetic marker it marks the structural mutations with. I chose to use this method as it has shown good performance against standard benchmarks, it has also been tried applied against Go before.

4.1.1 NEAT Genotype

One of the goals behind NEAT was to develop a system that could ease the complex process of applying crossover on two genomes that represented two ANNs with different topologies while still retaining common denominator of the functionality retained in the networks. The solution used was to mark the topological mutations with a genetic marker, so that each mutation had its own marker, but if the same mutation happened twice, it would get the marker of the previous mutation. In this way one could easily discern between newer mutations and older, and align the genes for crossover. Figure 4.1 shows a example genome and the markers of the connection genes.

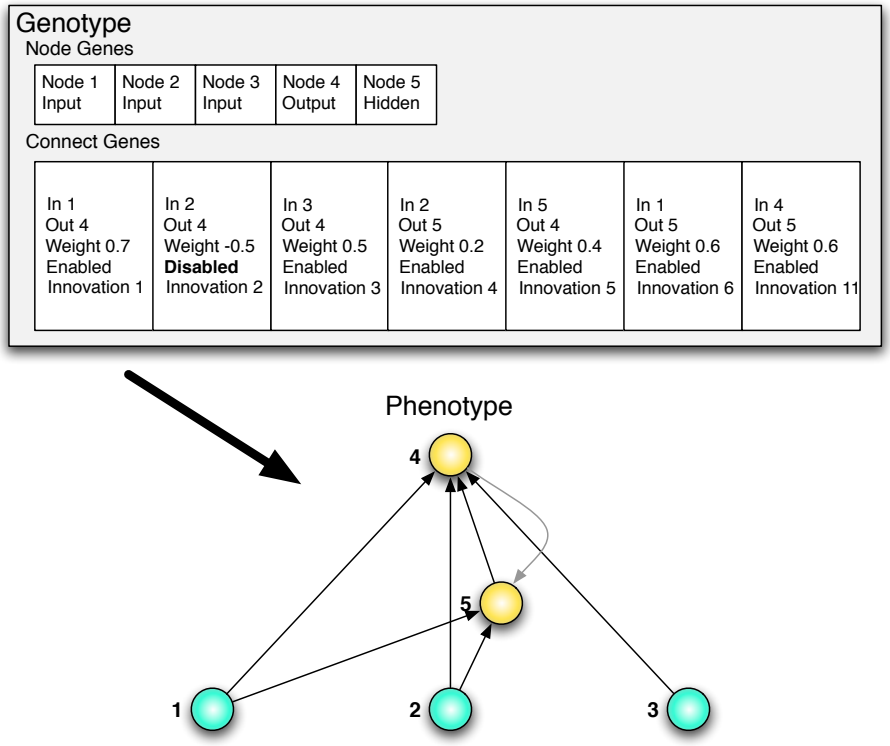


Figure 4.1: This figure depicts a NEAT genotype containing the node list and the genes representing the connections between them. The node genes are depicted for readability, but in essence they could be derived from the connection genes. Based on a figure from [21].

Mutation

There is five mutation methods in NEAT these are used based on a random draw given the ratios for each of the methods[1]. The three most central are adding a link between two unconnected nodes, adding a new node and mutating weights. The first two are illustrated in figure 4.2, as they are the most central to the contributions made by NEAT.

Adding new links is done by locating two nodes that are previously unconnected, and give this new link a new genetic marker(incremented from the previously highest marker number). This new connection is given a random

[1]See appendix A.1 for details about how the ratios are set

weight by the formula:

$$nw = randsign() * randdouble() * R$$

Where R is the maximum negative or positive range.

Adding new nodes(illustrated in figure 4.3 and 4.2) is compromised of two sub-tasks, finding a suitable connection, and then splitting that connection in two and inserting the new node in place. As to finding a suitable connection to split up, the current implementation implements a heuristic where given that the genome is less than 15 genes it favours to split up older links over newer ones, so to avoid a "chaining" effect. In this implementation I chose to add a slight augmentation to the original implementation in that you can chose to keep the old link you are splitting up, so as to minimize the impact the mutation has on the phenotypes performance.

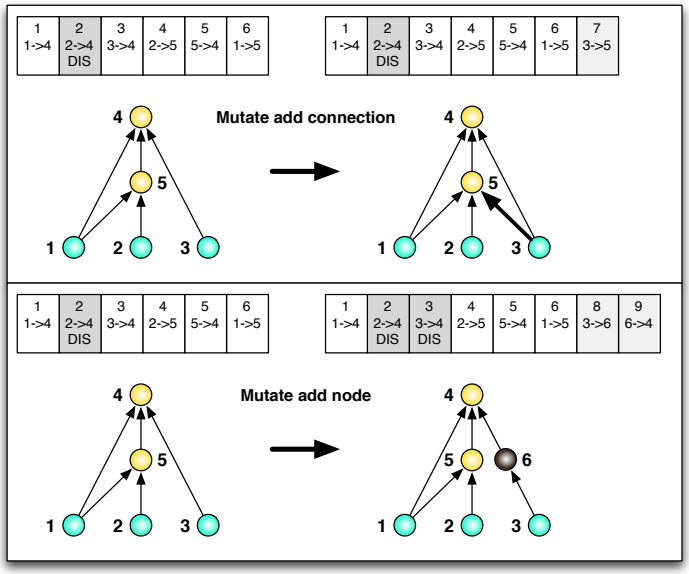


Figure 4.2: This figure depicts a NEAT genotype and the results when one mutates it by adding a connection and a node respectively. Based on a figure from [21].

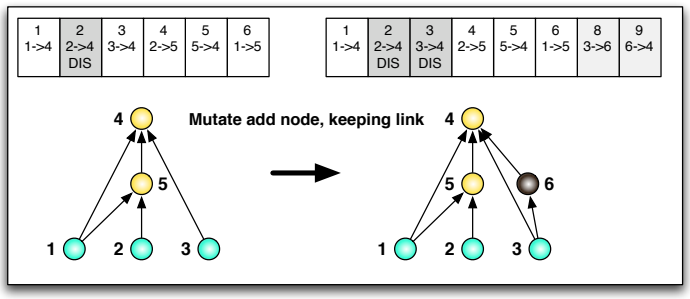


Figure 4.3: This figure depicts a NEAT genotype and the results when one mutates it by adding a node and keeping the link. Based in a figure from [21].

Crossover

Crossover(illustrated in figure 4.4) is one of the most central contributions of NEAT to neuroevolution. It uses the genetic markers to align the genes after the markers, enabling the crossover algorithm to identify disjoint or excess genes. The genes with matching genetic markers are lined up and crossed over by one of two methods; either random choice, or the weights are averaged over the two genes. The ratio of which it chooses to average and when it doesn't is given at run-time[2]. When it comes to excess and disjoint genes the crossover algorithm takes into account the fitness of the two parents taking part in the crossover, so the offspring only receives disjoint or excess genes if those genes comes from parent p_1 who's fitness is higher than parent p_2. If in this case parent p_1 has higher fitness than parent p_2, then any excess or disjoint genes from p_2 is sure to be absent in the offspring.

4.1.2 Speciation

Outlined earlier in section 2.3.5 NEAT uses speciation to preserve new innovations in structure, so that it can be fine tuned to achieve a good fitness. NEAT groups genotypes in species based on a genetic distance:

The genotypes within the species share their fitness, in that way speciation

[2]See appendix A.1

Algorithm 1 Speciation algorithm

1: **procedure** ADDGENOTYPE(g) ▷ Add genotype g to a species
2: **while**
3: $s_{try} \leftarrow getNexSpecies(S)$ **do**
4: $g_{original} \leftarrow getOriginalMember(S)$ ▷ This could also be getBestMember, depending on where you want the species to drift against.
5: **if** $C(g, g_{original}) < C_t$ **then**
6: $S \leftarrow addToSpecies(s_{try}, g)$
7: **end if**
8: **end while**
9: **if** $noSpecies(g)$ **then** ▷ No species was genetically close enough for this genotype.
10: $S \leftarrow addToSpecies(s_{new}, g)$
11: $S \leftarrow addSpecies(S, s_{new})$
12: **end if**
13: **end procedure**
14: **procedure** SPECIATEPOPULATION(P) ▷ speciate population P
15: $g_{first} \leftarrow randomMember(P)$
16: $S \leftarrow addToSpecies(s_{first}, g_{first})$
17: $S \leftarrow addSpecies(S, s_{first})$
18: **while**
19: $g \leftarrow getNextGenotype(P)$ **do** $addGenotype(g)$
20: **end while**▷ Make the first species out of a randomly picked genome
21: **end procedure**

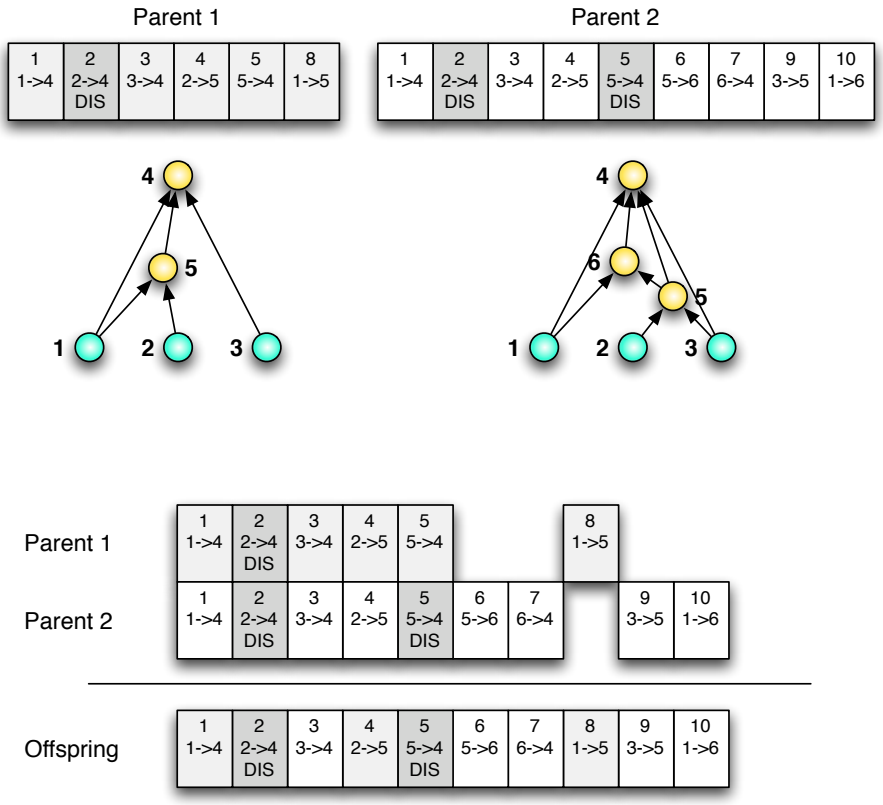


Figure 4.4: This figure depicts two NEAT genotypes and their networks, and then how they are aligned for crossover, and finally the resulting genotype. Based on a figure from [21].

protects innovation within the species. This can also be described in this way:

$$f_{aug}(g_i) = \frac{f(g_i)}{\sum_{j \neq i}^{n} sh(C(g_i, g_j))}$$

Where G is the set of genotypes, $g_i, g_j \in G$, $n = |G|$ and

$$sh(C_v) = \begin{cases} 1, C_v < C_t \\ 0, C_v \geq C_t \end{cases} \tag{4.1}$$

This is achieved by calculating the genetic distance $C(g_i, g_j)$ between any two genomes and then comparing that to a threshold value C_t. $C(g_i, g_j)$ is

defined by the following formula[21]:

$$C(g_i, g_j) = \frac{c_1 * E(g_i, g_j)}{N} + \frac{c_2 * D(g_i, g_j)}{N} + c_3 * \overline{W}(g_i, g_j) \tag{4.2}$$

Where given genotypes g_i and g_j, $E(g_i, g_j)$ is the number of excess genes, $D(g_i, g_j)$ is the number of disjoint genes and $\overline{W}(g_i, g_j)$ is the average weight difference between the two genes. c_1, c_2 and c_3 are coefficients to guide the speciation to signify disjoint genes, excess genes or average weight difference respectively when calculating the genetic distance between two genomes.

After the fitness is assigned to every phenotype the number of offspring allotted to each species is assigned as follows. Let P be the set of every phenotype in the population, N be the set of every species and $n = |N|$, $\overline{F}_a(N_i)$ be the average fitness of species N_i,

$$\overline{F}_{tot} = \sum_{j}^{n} \frac{\overline{F}_a(N_j)}{n}, N_j \in N \tag{4.3}$$

The number of allotted offspring for species N_k can then be defined as

$$A(N_k) = \frac{\overline{F}_a(N_k)}{\overline{F}_{tot}} * |P| \tag{4.4}$$

4.1.3 Genetic Selection

During the generational epoch the phenotypes are evaluated and given a fitness, that fitness is later adjusted according to its species as shown in equation (4.2). The initial description of NEAT does not specify how to allot offspring within the species, the original implementation uses[21] truncation selection[2]. In this paper we chose to implement SUS(stochastic universal sampling) [1] with a pluggable selection mechanism. This method tries to minimize the difference between the expected and the actual value of offspring allotted to the phenotype. This difference can be significant in standard roulette wheel methods if an unlikely spin on the "wheel" occurs. To use an analogy from [17]: Given a task of allotting N offspring, instead of spinning a roulette wheel N times with 1 pointer, you spin the wheel 1 time with N

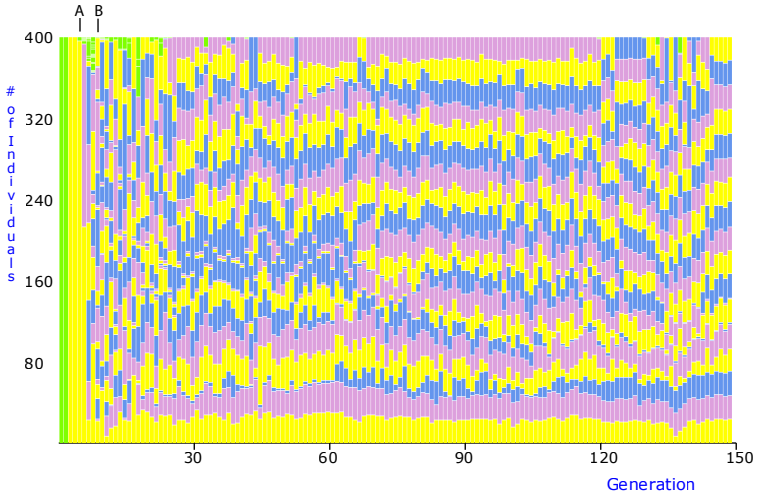


Figure 4.5: This figure depicts a speciation. Here you can see at generation 0 the number of species is 1. In generation 4(A) you can see the advent of a new species. This species grows and can be seen more clearly in generation 5(in green). The same species can be seen almost extinct in generation 8(B) however. The number of species also converge towards the set target number of species, which in this case is 20.

equally spaced pointers. SUS can be described in the following way [17]:

Where i is the index of phenotype P_i in the set of phenotypes in the population P, $ExpVal(i,t)$ is the expected value of offspring to be allotted

Algorithm 2 SUS algorithm

```
r ← rand                                   {A random float between 0 and 1}
sum ← 0
for i = 0 to N do
    begin
    for sum+ = ExpVal(i, t) to r do
        begin
        S ← Select(i)
        r ← r + 1
        end
    i ← i + 1
    end
```

to phenotype i at time t. Now you can implement $ExpVal(i,t)$ for every selection method you prefer. An example would be sigma scaling, where the expected value of the phenotype depends on both the average fitness of the population and the standard deviation of the population:

$$ExpVal(i,t) = \begin{cases} 1 + \frac{f(i) - \overline{f}(t)}{2\sigma(t)} & if\ \sigma(t) \neq 0 \\ 1.0 & if\ \sigma(t) = 0 \end{cases}$$

Where $f(i)$ is the fitness of individual i, $\overline{f}(t)$ is the average fitness of the population at time t and $\sigma(t)$ is the the standard deviation of fitness values in the population at time t.

In the case of NEAT this could be translated to the following:

$$ExpVal(p,N_i) = \begin{cases} 1 + \frac{f(p) - \overline{F}_a(N_i)}{2\sigma(N_i)} & if\ \sigma(N_i) \neq 0 \\ 1.0 & if\ \sigma(N_i) = 0 \end{cases}$$

Where t is omitted since $ExpVal()$ is used every generational epoch in the NEAT implementation, $p \in P, p \in N_i$ given set of phenotypes in population P, the set of species N, and $\sigma(N_i)$ which is the standard deviation of fitness within species N_i.

4.1.4 Complexification

NEAT will gradually add on structural mutations as the generations progress, but will only keep the ones that proves to be beneficial within a few generations[3]. This means that NEAT will start out with small networks and incrementally, depending on the task the network is solving, grow larger network to fit the task[4]. In a more detailed sense this means that the algorithm searching through the fitness space of all possible genomes increases the number of search parameters and in effect the number of dimensions in the fitness search space as it goes along, unless it hits a acceptable performance in the current search space.

4.2 Roving Eye

A problem with using neural networks to play computer Go is how to scale the network to the input size, as the Go board often vary in size, from 5x5(unusually small, for training only) to 19x19(the standard match board). Varying the network input size will in most[5] cases degrade the performance of a neuro-evolutionary system, as it has trained for a different size network and has no knowledge of how to assign weights to the new connections.

A roving eye doesn't need to have the whole board as input, as it can move around on the "input surface". Thus gets different inputs depending on the position it is in. Accordingly the roving eye also has a way of changing it's position by outputting certain values or firing on a specific output node. The way the eye moves is often of great importance to the results as it relates to how many concepts the network has to represent internally. With the example of go(see figure 4.6), if the eye can turn around to change its orientation(from looking north to looking west[6]), it can see two shapes as the same shape even though their orientation seems superficially different. The roving eye can then represent this shape internally in a single representation

[3]This depends on the parameter drop-off_age, see Appendix A.1
[4]See section 5.1 for an example of this.
[5]See [22, 10, 6] for an example of a system which in some respects solves this problem.
[6]It is essentially a matrix operation on the input matrix.

in contrast to the worst case scenario of representing four different shapes. Whether this is a good thing is however domain dependent as this is a good thing in go as the function of these shapes are largely disconnected from the orientation of the shapes. In other domains it may be more useful for a roving eye to be able to differentiate more clearly between two similar shapes with different orientations in the input.

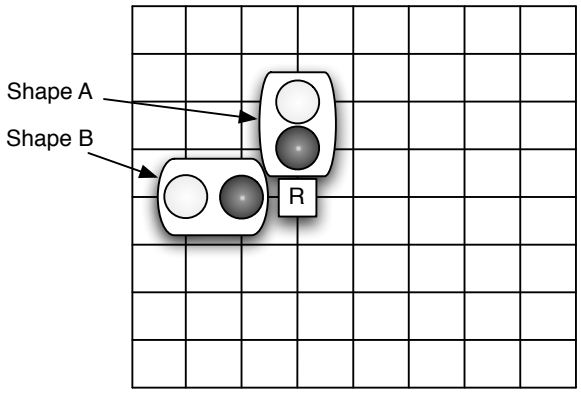


Figure 4.6: This figure depicts an example where a roving eye would have an advantage of being able to turn around to change its orientation. This figure shows two go shapes, shape A and shape B. When the roving eye, depicted by R, can turn around, representing the two as is easier for a ANN.

In this case the roving eye sees a 3x3 matrix of the board at a time, the roving eye also has a long range input, which is a count of how many white and how many black stones is present in the section of the board beyond the currently viewed section. The long range inputs are given for each of the four relative orientations. Thus the roving eye has a general idea of what is beyond its scope of view to the left, right, front and back of its current position. It is given two nodes for each orientation, one is given the count of black stones the other the count for white stones. This can be useful information for deciding where to move the eye next. It also receives as two inputs the absolute position it has on the board, so it knows where on the board it is. As useful domain knowledge the ANN also gets a boolean input of 0 or 1 to let it know if it is legal to place a stone at the current center of its

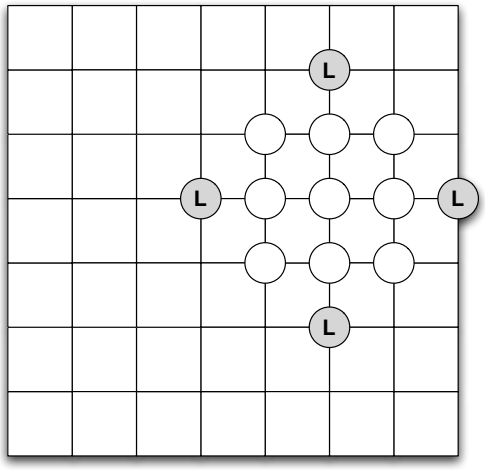


Figure 4.7: This figure depicts the typical input of a roving eye. Based on a figure from [21]. The white circles depict the direct inputs to the roving eye for each of the NxN intersections(in this case 3x3), while the circles marked with a L depicts the long range inputs.

view. To let the network discriminate between white and black stones easier each of the 9 intersections are assigned two input nodes One is activated if the intersection is occupied by a white piece, the other if the stone is black, neither is activated if the intersection is unoccupied.

The input for the roving eye is also transcribed when the eye is playing as the white player. In this way the roving eye doesn't get confused by an entirely different set of inputs(the opposing player's pieces are black instead of white as the roving eye was trained on). So the opposing players pieces will always look like "white stones" to the roving eye.

As described in [21] and depicted in figure 4.7 the roving eye has five output nodes. Each node is marked as active if it has an output above 0.5. The five nodes represent "go forward", "turn left", "turn right", "put piece" and "pass" respectively. If none of the outputs are above 0.5 the roving eye stands still in a "pause" and receives the same input again. If more than one node is above 0.5 it is interpreted in the following matter:

- If more than one movement node is active, the eye turns and moves forward at the same time. If both turning nodes are active the eye

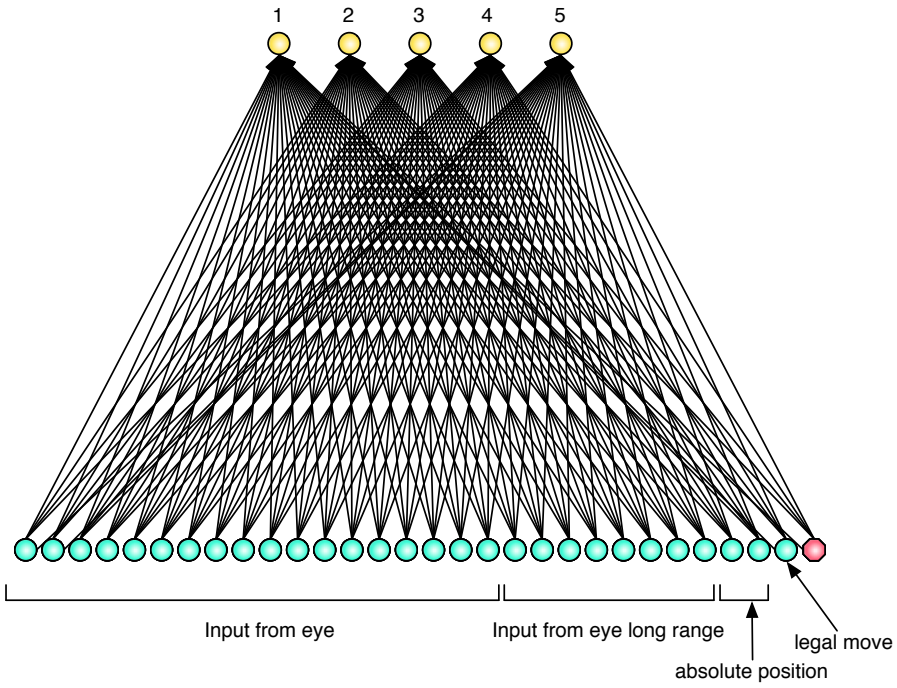


Figure 4.8: This figure depicts the typical starting network of a roving eye. Where the circles in the inner 3x3 represents a typical 3x3 roving eye input area, while the outer four circles labeled L is the long range sensors. Based on a figure from [21].

turns left.

- If none of the first three nodes(the movement nodes) are active, the first active node in the list from node three to node five are chosen.

The network is then given N(in this case $N = 100$) time steps to move around or sit still doing nothing for each move it has. If the network doesn't put a piece during its move it is interpreted as a pass. As a starting point the roving eye always starts out in the center of the board for each move. This helps the eye to build a single representation of its orientation on the board.

4.3 GNU Go

In this paper GNU Go was used as a go playing engine which NEAT trained against. In the experiments in this paper NEAT ran for M generations and trained against GNU Go for N generations to train. In the case where $M - N > 0$ NEAT progressed into running co-evolution between the phenotypes but kept playing against GNU Go at a given ratio to keep the domain knowledge from getting lost. In every type of evolution the GNU Go engine was used as a fitness evaluator[7].

GNU Go was directly linked into the evaluator part of the system, to provide more speed than the alternative GTP[8] connection.

4.3.1 Fitness Function

GNU Go also provided the author with a opportunity to do a more fine grained fitness evaluation of the network than a simple win/loss ratio, as it has a function for estimating the score that returns a estimate for the white player. This means when white plays well it returns a positive score, and if black is in the lead the function will return a negative score. The function returns an estimate of the Go score for the white player. This essentially means it tries to estimate how many captures white will do during play in addition to how much territory the white player will end up with. This is of course a really hard thing to estimate, especially early in games on big boards. In this case however, the boards will be below mid-size and thus poses a lesser challenge to evaluate early in the game. The score estimates will return values in the range of $\pm(boardsize^2 + komi)$[9] which is hardly a good range to use as fitness scoring for a genetic algorithm. The following function was used for fitness, given the *estimatescore* from GNU Go:

$$e_{board}(p) = \begin{cases} (-estimatescore + S_{max})/(2 * S_{max}), isblack(p) \\ (estimatescore + S_{max})/(2 * S_{max}), iswhite(p) \end{cases}$$

[7]The co-evolution games between roving eyes the boards where evaluated by GNU Go, in contrast to a pure win/loose ratio.

[8]See **See** `http://senseis.xmp.net/?GTP`.

[9]Where komi is the handicap point mentioned in section 1.1.1.

Where $S_{max} = S^2 + K$, $S = boardsize$ and $K = komi$. The final fitness function that assigns a network fitness after a game is modeled after the original paper[19] from Stanley, it takes into consideration all the moves leading up to the final board, and weights them up in relation to the final board as will rewards networks that play the whole game through and makes good moves early. This is thought to encourage the networks to adapt to the game rules better before using the more usual win/loss ratio used in many other evolutionary approaches to board games and in co-evolution. The fitness function for single player evaluation is defined as followed:

$$F_{game}(p) = \frac{(2 * \sum_i^n e_i(p)) + e_{finalboard}(p)}{(2 * n) + 1} \tag{4.6}$$

Where n is the number of moves in the game. Later as a test to see whether the previous fitness function awarded the networks too much for moves played by GNU Go, the following fitness function was constructed:

$$NF_{game}(p) = F_{game}(p) + (M_{game}(p) * 0.1) \tag{4.7}$$

Where F_{game} is given in equation (4.6) and $M_{game}(p)$ is the number of stones put by the network.

The fitness function described in equation (4.7) awards the networks for the number of stones it puts on the board, which is usually a good thing in a go game.

4.4 Co-evolution

To implement co-evolution the "Hall of Fame" method was chosen. This method of co-evolution keeps a list of duplicates of the top ranked phenotypes from each generation. Every phenotype then needs to be evaluated against each one of these. In this implementation the list length is variable but set to 12 in the experiments presented in this paper. GNU Go is represented in the list by a given fraction of the slots. This is done to not lose any of the go playing ability learned earlier in the evolution, and to ensure that no

phenotype gets away "easy" and only plays against low-ranked phenotypes.

4.5 Cluster Architecture

To simplify the implementation of the parallel architecture the author chose to employ a parallel architecture made for clusters using pvm or mpi. This architecture was developed at NTNU(Norwegian University of Science and Technology) by Boye A. Hoverstad. The architecture enables a master cluster node to send a batch of genomes out to the slaves nodes. The system distributes them out to the evaluation slaves which later returns the fitness. In the case of co-evolution the master also sends a list of genomes which the evaluation slaves evaluates the genome against. To use more available cpu time the master also evaluates a given percentage of what the evaluation slaves does. This is given at run-time[10].

[10] A typical value would be 97%.

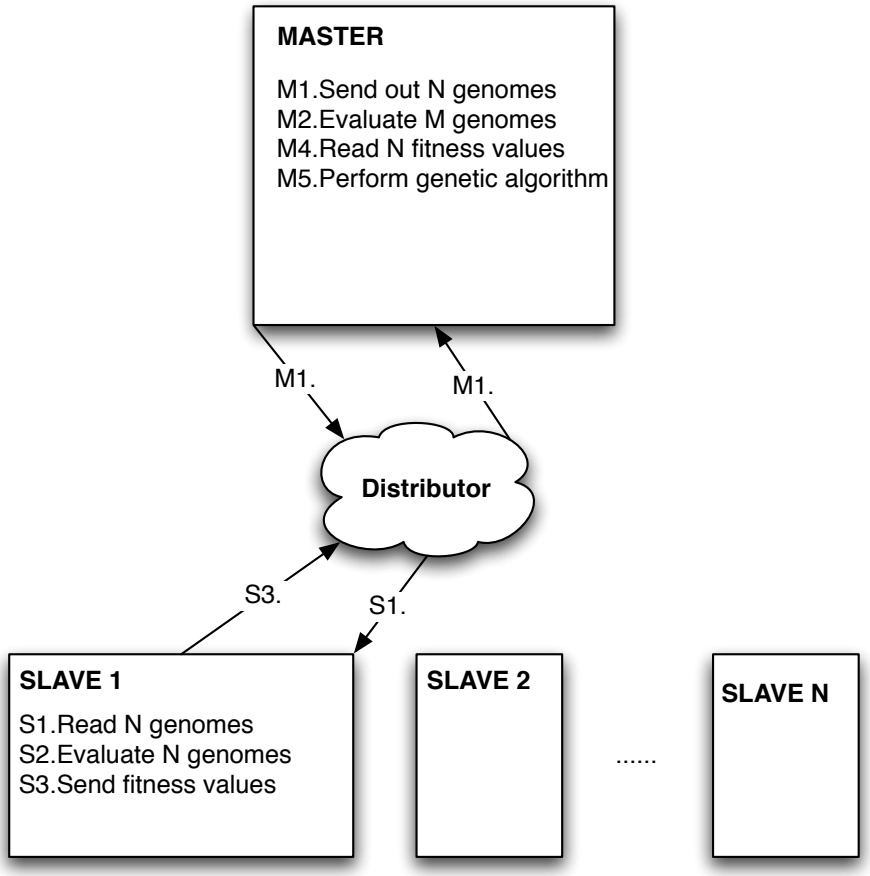


Figure 4.9: This figure depicts the distributed architecture.

Chapter 5

Results

In this chapter the results of the experiment will be presented. The general functional performance of the system will be evaluated through a standard XOR solving problem. The run-time performance of the cluster system will be demonstrated through a series of runs on different sized subsets of a cluster. Following the performance results the results of the evolution against GNU Go will be shown. The paper will present results from evolving against GNU Go in two categories. The first is results from evolving on a 5x5 board. This will involve standard evolving against GNU Go where GNU Go is the fitness evaluator from start to end. This includes experiments for each of the two fitness functions described in equation (4.6) and (4.7). The results presented from playing on the 5x5 board also include more generational runs, as well as runs using co-evolution. The second category is where the games are played on a 7x7 board. This category will include runs where the evolution starts from scratch(again with experiments for each of the two fitness functions). The category will also include results from experiments where evolutions seeded[1] with the champions from runs on 5x5 boards and runs using co-evolution on 5x5 boards.

[1]In the cases where the evolution is seeded with a genome, the evolution does not start with a generic starting point, but with mutations of the seeding genome.

5.1 Performance Evaluation

This section aims to establish that the implementation of the NEAT algorithm is operating within acceptable performance criteria. This includes both functional and run-time performance. It is important to establish this in advance, as it forms the foundation of the later runs targeting harder problems.

5.1.1 Functional Performance

To evaluate the functional performance of the implementation the XOR problem was chosen as a benchmark. XOR(Exclusive OR) is a binary operator that returns true if and only if the two inputs are unequal. This requires the two inputs to be joined at a hidden node in the ANN as the classifications are not linearly separable given the two inputs.

This is a widely used benchmark for ANN implementations as it is not a linearly separable function, thus the ANN requires a hidden node to represent the function. This requires the neuro evolutionary algorithm to add a hidden node and search for the correct weights for the two new[2] connections added. Solving this task shows that the implementation can grow the topology required to solve the task.

Simulation Setup

In this simulation the networks are given the four possible combinations of true or false(represented as 0 and 1) as inputs. The fitness assigned to the networks is a sum of the four euclidean distances between the output of the network and the target values.

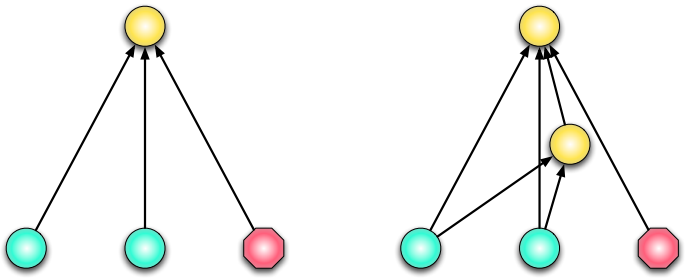
Simulation Results

In order to evaluate the complexifying capabilities of this NEAT implementation[3] the number of hidden nodes and number of connections in the solution

[2] In the case of NEAT where the algorithm splits up a connection to add a new node, see section 4.1.1

[3] see section 4.1.4

found in a run was recorded. Over 100 runs, average the implementation used 27 generations to solve XOR, and the solutions had 9.5 connections and 2.2 hidden nodes. The implementation found the minimal structure in 23 out of 100 runs.



(a) Starting genome (b) Minimal XOR

Figure 5.1: In this figure, the green nodes are input nodes with a linear transfer-function(in this case f(x) = x), the red node is a bias node that always outputs 1 to the node it is connected to and the yellow node is regular node with a sigmoid transfer-function. (a) Shows the starting point, or seed, of the evolution. (b) Shows the minimal solution for XOR.

Figure 5.1.1 shows the starting genome and minimal solution for the XOR problem.

5.1.2 Run-time Performance

This experiment is performed to see how well the evaluation of genomes against GNU Go scales as the number of evaluation slaves is increased.

Experiment Setup

This experiment used evaluation against GNU Go with 1000 genomes and was run with 5, 10 and 15 cluster nodes as evaluation slaves. The time is measured from start of evaluation until the last genome is received from the slaves.

Experiment Results

Evaluating 1000 genomes took 325 seconds on 5 nodes, 164 seconds on 10 nodes and 80 seconds on 20 nodes. Thus respectively 0.325, 0.164 and .080 seconds per genome.

5.2 5x5 Board

A natural staring point for experiments is the smallest board used in go. This serves as a learning board for beginners, and is often used in computer go as it provides a easy start. Here we use it to demonstrate is demonstrated that NEAT learns to play against GNU Go at a reasonable level.

5.2.1 Experimental Setup

In this experiment the run-time parameters used are the same as those presented in [21]. Population size used is 400. The roving eye is set to the standard 3x3 size and has 100 time steps for each move. For more detailed parameters see A.1.2.

5.2.2 Experimental Result

The results of the experiments performed on the 5x5 board are shown in figures 5.2, 5.3, 5.4, 5.5, 5.6 and 5.7. All graphs show the fitness of the population champion at the given generation. The results will be more thoroughly commented and analysed in chapter 6.

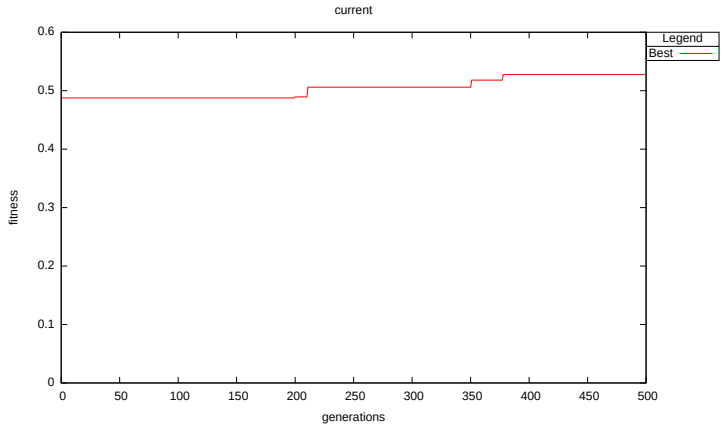


Figure 5.2: This figure shows the fitness of NEAT evolving against GNU Go over 500 generations with a population of 400 with fitness function given in equation (4.6).

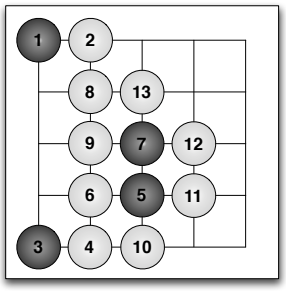


Figure 5.3: Here is a game played against GNU Go in mid-evolution, at generation 150. The network starts out well but spreads out too much with move 3. It responds well with moves 7 and 5, but it stops short as it has not learned to play the game to its end. This was evolved using the fitness function given in equation (4.6).

5.3 7x7 Board

In these experiments the board size was 7x7, a considerably harder problem for a roving eye to solve than the smaller 5x5 board. Containing almost twice as many intersections and thus many more possible board states, this is a bigger challenge for both classical and neuroevolutionary approaches.

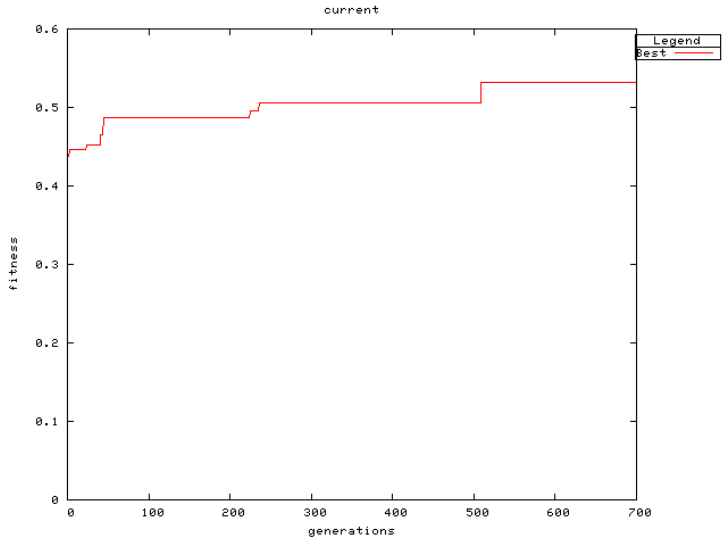


Figure 5.4: This figure shows the fitness of NEAT evolving against GNU Go over 700 generations with a population of 400 with fitness function given in equation (4.6).

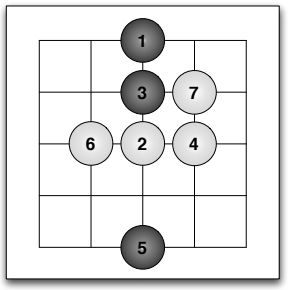


Figure 5.5: Here is a game played against GNU Go by the resulting champion of a ended evolution. The ANN starts to build a line with move 1 and 3, but is interrupted with white's move 2 and 4. This was evolved using fitness function given in equation (4.6).

5.3.1 Experimental Setup

In this experiment the run-time parameters used are the same as used in the 5x5 board experiments. Population size used is 400 and the number of generations is set to 500. The roving eye is set to the standard 3x3 size and

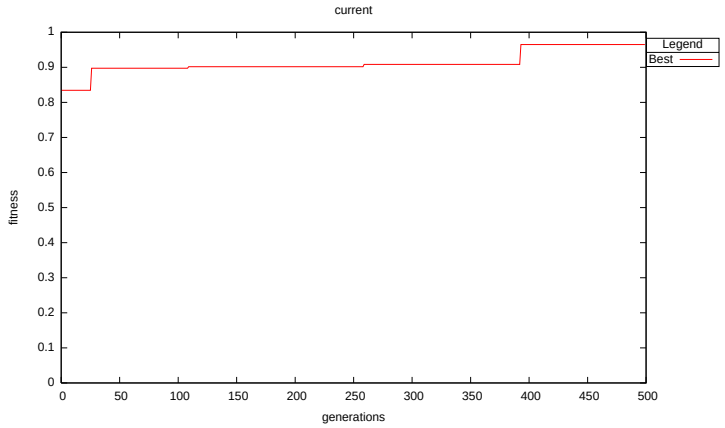


Figure 5.6: This figure shows the fitness of NEAT evolving against GNU Go over 500 generations with a population of 400 with the alternative the fitness function given in equation (4.7). In this fitness function the roving eye is awarded for every stone it puts on the board.

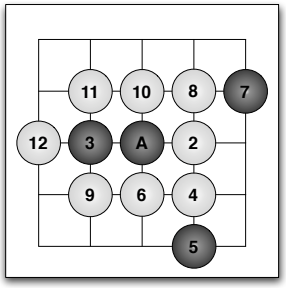


Figure 5.7: This figure shows a game played against GNU Go in mid-evolution, at generation500. This was evolved using the alternative fitness function given in equation (4.7). The network evolved here performs better than the game shown as a result of evolution with fitness function given in equation (4.6). The move marked A represents both move 1 and move number 13.

has 100 time steps for each move.

5.3.2 Experimental Result

The results of the experiments performed on the 7x7 board are shown in figures 5.9, 5.10, 5.11, 5.12 and 5.13. All graphs show the fitness of the pop-

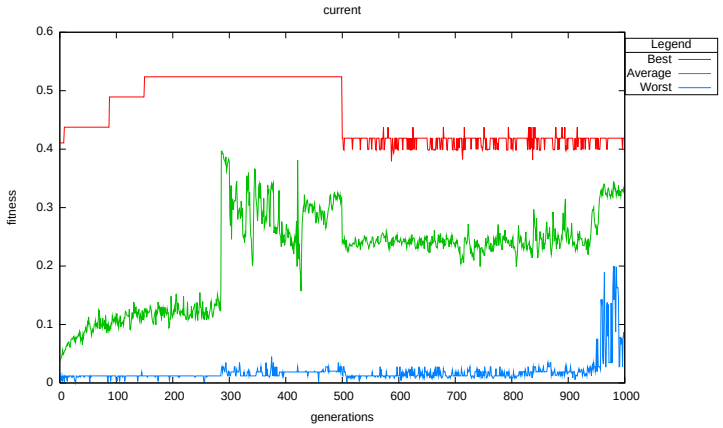


Figure 5.8: This graph shows an evolution where co-evolution was started at generation 500 and continued until generation 1000.

ulation champ at the given generation. The results will be more thoroughly commented and analysed in chapter 6.

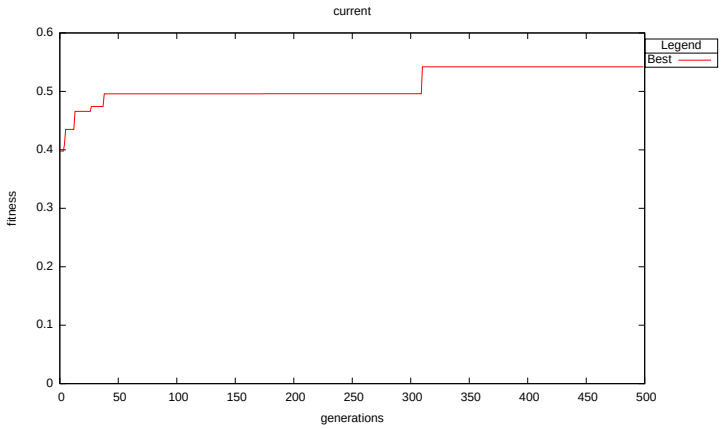


Figure 5.9: In this graph the evolution was started from scratch on a 7x7 board space. This was evolved using the fitness function given in equation (4.6). The game played by the resulting champion can be seen in figure 5.10.

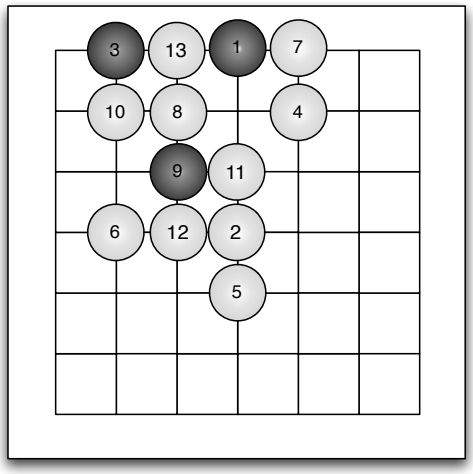


Figure 5.10: In this figure the evolution was started from scratch on a 7x7 board. This was evolved using fitness function given in equation (4.6).

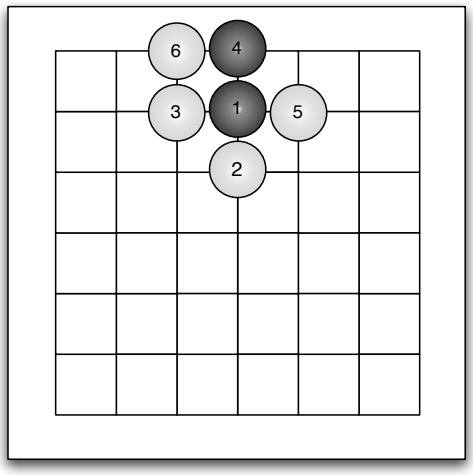


Figure 5.11: This figure shows a game from a champion evolved on a 7x7 board seeded with a 5x5 champion as seen in the 5x5 board simulation reported in figure 5.5.

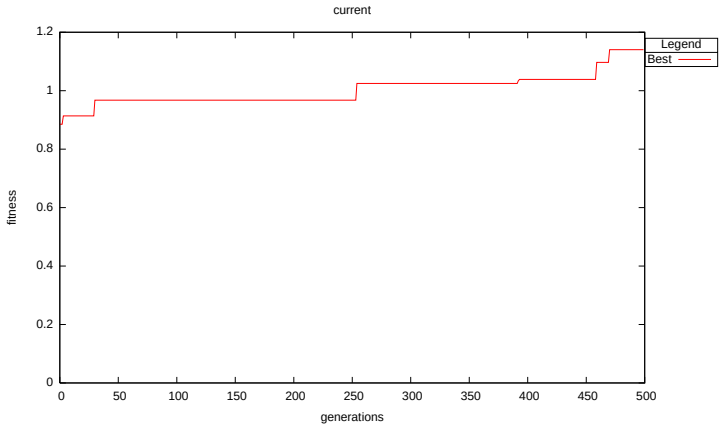


Figure 5.12: This graph shows an evolution on a 7x7 board using the alternative fitness function given in equation (4.7). The game played by the champion of generation 500 can be seen in figure 5.13.

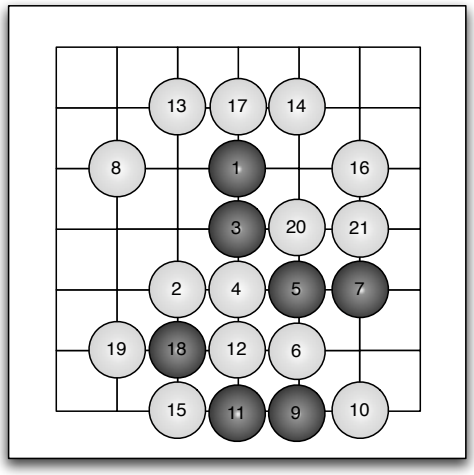


Figure 5.13: This figure shows a game of the champion evolved using the alternative fitness function given in equation (4.7). The roving eye plays well but seems to stop playing after move 11. It later plays move 18 to little effect on the outcome of the game.

Chapter 6

Analysis of Results

In this section I will present comments and analysis related to the results presented in the previous chapter.

6.1 Performance

The results as seen in section 5.1 clearly show that the functional performance of the implementation is on par with the original NEAT implementation. Thus given the same parameters and equal evaluation results(a genome should receive the very similar fitness in this method as in the original implementation) the evolution against GNU Go will yield similar results.

6.2 Parallel Architecture

In section 5.1.2 the parallel architecture shows to be pretty scalable as the evaluation time of 1000 genomes halves from 5 nodes to 10 nodes. This result is of course highly dependent on the percentage of computational resources required on evaluation of the genome in the evolutionary process. An evaluation of a short data-set classification will spend almost as much time on applying genetic operators such as recombination and mutation, as on evaluation itself. This will reduce the advantage of applying a cluster of computers as a computational resource, as the slaves do not contribute to the

genetic operations in this implementation. This could however be considered for future augmentations of the implementation, as attacking more complex problems will require more computational resources dedicated to the genetic operators when the genomes grow bigger in an effort to represent the problem better.

6.3 5x5 Board

The networks seem to reach a fitness of 0.53, which suggests that the algorithm finds a partial solution. However unlike the results presented in [21] it does not find a solution that beats GNU Go. This result can be related to many things, mainly that the implementation used a different selection method[2] which may have proven to be a better choice for this domain. Sigma scaling, seen in equation (4.4), which is used in this experiment, may have proven to be less effective. The same difference in selection method can also have offset the choice of parameters for the NEAT algorithm and in effect decrease the performance of the algorithm.

As seen in figure 5.6,5.7,5.12 and5.13 the fitness function detailed in equation (4.7) shows better results. This could be because the original fitness function (4.6) will award a little fitness for every move made by GNU Go. This is because even though the roving eye is at a disadvantage in the board being evaluated the GNU Go evaluation algorithm returns a small reward for the early moves made by the roving eye. This is again amplified by the fact that the early and mid game moves are weighted up to encourage stronger play earlier in the game.

6.4 7x7 Board

The roving eye's performance is as expected lower on the bigger 7x7 board than the 5x5 board as seen in the game played 5.10. The reason it is harder for a roving eye to play well in this evolution is mainly because of the bigger board, which requires more moves for a success-full strategy.

The fitness stays somewhat the same since the fitness is highly dependent on the number of moves made. Given the fitness functions detailed by equation (4.6) and (4.7), a bad move that creates a lot of moves by GNU Go in response is better than a good move that only create a few moves by GNU Go in response. So even though the same moves was better on a 5x5 board it still gets a good fitness function at this 7x7 board.

One can also note how the roving eye kept trying the same strategy on a 7x7 board that it used in the 5x5 strategy as seen in figure 5.5 when a champion from the 5x5 evolution was used as a seed. This suggests that the inherent strategy that was represented in the roving eye champion network is applicable across board sizes. This also confirms the findings made in [19].

6.5 Co-evolution

As presented in section 4.4 the algorithm applies hall of fame co-evolution to enable a steady co-evolution. It also keeps playing the phenotypes against the GNU Go engine in order to keep the phenotype from "forgetting" the go rules. Several evolutions where run with co-evolution, an example of this can be seen in figure 5.8. Using champions from 5x5 experiments applying co-evolution such as the one shown in figure 5.8 as seeds in experiments on 7x7 boards did not result in any improvements in comparison to the other methods[1] used in 7x7 experiments. The fitness also clearly oscillates as the generation champion is paired up with different phenotype opponents. The mean of the population does however seem to be on the rise from the point in the evolution the co-evolution starts. It is hard to explain the almost paradoxical results of a rising mean in co-evolution that still does not give any improvements in strategy over normal evolution. It could simply stem from the fact that to few experiments were done on co-evolution to see the results due to lack of time to fine tune the co-evolutionary parameters.

[1] Such as seeding with a normal 5x5 champion, or normal evolution.

Chapter 7

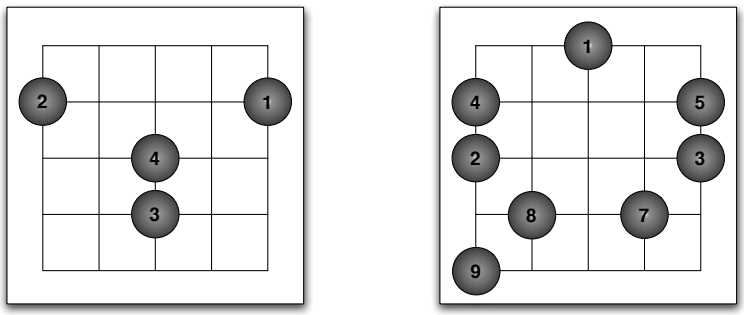
Discussion

This chapter will present the findings from the experiments in the perspective of the motivation and working hypothesis presented in sections 1.3 and 1.4.

The results presented in chapter 5 and in [21, 19] show that the roving eye has an advantage over more traditional techniques applied when using ANNs to play go. There is still a long way to go in terms of performance as my implementation only nearly beats GNU Go. The experiments also only ranges up to 7x7 which is far below the standard board size of 19. As stated in the motivation in section 1.3 the goal was 9x9 but given the results in 7x7 the author chose to try to tune the parameters on the smaller board sizes. Still at 5x5 both this paper and earlier attempts[19] only show the game-play of a weak amateur or beginner. The roving eye technique only seems to amend some of the problems associated with making computers play go at a higher level. There seems to be significant requirements connected to the structure of the network of the roving eye to enable a roving eye to represent the patterns associated with playing go. Just making the roving eye represent the sequence of playing a series of moves seems to pose a serious challenge to neuro-evolutionary systems. This could be caused by the need for the network to represent how to move across the board in addition to the domain knowledge needed to play the game itself.

During the experiments the author observed symmetrical features in the patterns of the way the eye chooses to put its stones when playing games

with more than 2-3 moves made from the roving eye. This illustrated in snapshots taken from evolutions in figure 7.1.



(a) First symmetrical example (b) Second symmetrical example

Figure 7.1: These two games depicted in this figure both show strong symmetrical properties as to the series of moves made by the roving eye. In figure both figures (a) and (b) the symmetry seems to be around the central vertical axis from top to bottom. Both patterns are highly symmetrical with the one exception of move 9 in figure (b).

This could be caused by the fact that it requires less structure in the network of the roving eye to represent a symmetrical series of moves rather than a unsymmetrical which would require a more specific type of information to be stored in the roving eye.

The roving eye did show that it could represent a strategy and continue to use that on a bigger board, as shown in figures 5.5 and 5.10. This confirms the findings in [19] that the roving eye can apply strategies learned on smaller board on a bigger board. This confirms that the strategies evolved on a smaller board can be used by a roving eye on a bigger board. This is highly useful when trying to incrementally learn to play a highly complex game which may be impossible to learn in a evolution starting straight on a 19x19 board.

However the implementation did not reach the same level of play as earlier systems [19], even though some of the games played clearly shows(e.g figure 5.7) something very close to a winning strategy.

There are many possible explanations of the differences between the results presented in this paper and those presented in [19]. The roving eye's

requirement of representing movement and analysis of the inputs to the net following new moves on a board does put a rigid requirement on the neurevolutionary algorithm to grow the correct structure. The speciation parameters may require adjustments to allow for the newly grown topological structures to be tuned to a greater degree[1].

The roving eye needs to be able to represent a non-symmetrical sequence of movements across the board associated with the placements of at least 8 stones. This is a minimum for a winning strategy against GNU Go on a 5x5 board, as seen in figure 7.2 where the black player ends up with more area of territory at the end of the game.

All these reasons adds up to the fact that the experiments presented in this paper are highly sensitive to the neuroevolutionary parameters used in NEAT[2]. And the small differences in implementation and selection algorithm, may require additional tuning to the parameters.

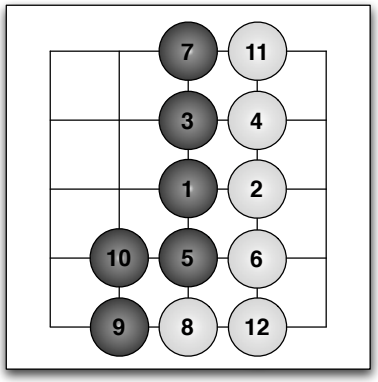


Figure 7.2: This figure show a simple winning strategy against GNU Go when playing with no handicap and a komi of 0.5.

For this kind of sequence to be represented in a ANN, even without any noise[3] or randomness involved in the input, the ANN would need to have several recurrent structures. This represents a major challenge for a neuroevolutionary algorithm to develop.

[1]To allow for the weights of the new connections to be adjusted for higher fitness.
[2]As presented in section 4.1 and A.1
[3]Resulting from the responses by GNU go on the board.

Although not impossible it may prove highly difficult to grow an ANN big enough to represent the patterns necessary to play go at a high level on even 9x9 or 13x13 boards. Even though an ANN in theory can represent the patterns involved, it may very well be way beyond the scope of anything yet grown with neuroevolutionary algorithms[4].

A genetic algorithm can be seen as a search through the fitness landscape. In comparison other more traditional approaches to computer go often uses search methods, as described in section 2.4.3. So while traditional approaches searches during the game, a approach employing a neuroevolutionary algorithm will in most cases search before any real game is played, for example a computer go tournament. With that in mind the a priori search does not suffer from the same time constraints that a real-time search algorithm would be constrained by. The evolutionary search is however often more demanding, as in the case of NEAT where the complexifying feature gradually adds new dimensions to the fitness search space by adding new connections and nodes. It would be interesting to compare the two searches in terms of efficiency in relation to the results found. At this time the neuroevolutionary search is the underdog often running for days to create a network that plays below the level of many other real-time search methods. Although this last fact is made up by the fact that it is much faster run-time.

7.1 Evaluation of Working Hypothesis

To reiterate the working hypothesis:

Hypothesis I:*Neural networks evolved in NEAT against GNU Go, and with* ***co-evolution*** *to play GO, will achieve a higher level of generality, and will adapt more general playing patterns than a network evolved against GNU Go alone.*

Hypothesis I can not be said to have been confirmed nor weakened by the experiments in this paper. The impact of the co-evolution can be seen to

[4]See [10] for an example of one of the biggest ANN grown by a neuroevolutionary algorithm yet.

be negligible on the roving eye's performance on bigger board in comparison to a roving eye that was simply evolved on a smaller board and seeded into the evolution of the bigger board. The author however feels that investigations into other co-evolutionary methods such as pareto co-evolution may be well founded. This is because some signs, such as rising mean fitness in co-evolution population, may indicate that tuning of evolutionary parameters can result in better results in the co-evolutionary experiments. An additional reason for the the result may be the fact that the roving eyes only evolved a single strategy for playing go, and playing two of these against each other may act as destructive interference. One suggestion to counteract this would be to somehow ensure that there was more than one working strategy of playing go against GNU Go before proceeding to co-evolution.

7.2 Future Work

The goal of making computers play go is a very large and ambitious goal. Within the neuroevolutionary approaches to this problem, this paper only scratches the surface. The paper shows that a roving eye scales well from one board size to another, while still retaining the patterns learned in the smaller board sizes. Even though co-evolution did not show any direct impact on the level of play when progressing from a smaller board to a bigger board, the co-evolutionary method implemented in this paper is rather crude and outdated by many standards. Further work could include a implementation of pareto optimal co-evolution. Another option, that could work in combination with the previous suggestion could be to measure the roving eye's performance in co-evolution by a pure win/loss ratio. This would reduce the dependence on the estimation metrics of GNU Go when progressing into co-evolution. This metric is inherently flawed, as the GNU Go engine builds on it evaluate the possible moves, and the engine is empirically proven to be on level of a weak amateur.

Further experiments with different fitness functions in the "learning period" of the roving eyes also seems to be well founded as seen in the small adjustments made to the original fitness function in equation (4.7). Even

though this may seem as domain knowledge based heuristic, it may very well be required for neuroevolutionary methods to reach a higher level of play. It may also prove to be even more domain knowledge in the fitness function to guide the evolution. This is in many cases unavoidable for a genetic algorithm, as current computer experiments within genetic algorithms needs to specialize as it does not have the benefit of time, in comparison to nature in the evolution of humans as go-players. A way to do this could be to utilise the metrics already found in the GNU Go code and split them up in the metrics found most useful. These can then be used in a weighted fashion during evolution, changing emphasis later in evolution as the roving eyes learn the playing concepts emphasised earlier in the evolution. It could also be beneficial to use an even easier opponent than GNU Go earlier in evolution, such as a player that only uses a simple strategy of placing stones adjacent to the opponent.

Bibliography

[1] Cambridge, Massachusetts, United States. *Reducing bias and inefficiency in the selection algorithm.* Lawrence Erlbaum Associates, Inc. Mahwah, NJ, USA, 1987.

[2] J. F. Crow and M. Kimura. Efficiency of Truncation Selection. *Proceedings of the National Academy of Science*, 76:396–399, January 1979.

[3] L. D. Davis D. J. Montana. Training feedforward networks using genetic algorithms. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 762–767, 1989.

[4] J Kauth D. Whitley. Genitor: A different genetic algorithm. In *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, pages 118–130, 1988.

[5] R. Das D. Whitley, S. Dominic and C.W Anderson. Genetic reinforcement learning for neurocontrol problems. *Machine Learning*, 13:259–284, 1993.

[6] David B. D'Ambrosio and Kenneth O. Stanley. A novel generative encoding for exploiting neural network sensor and output geometry. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*. New York, NY: ACM, 2007. Nominated for Best Paper Award in Generative and Developmental Systems.

[7] Risto Miikulainen David E. Moriarty. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22(22):11–33, 1996.

[8] Risto Miikulainen Faustino Gomez. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5(5):317–342, 1997.

[9] S. Hedge G. Miller, P. Todd. Designing neural networks using genetic algorithms. In *Proceedings of the International Conference on Genetic Algorithms*, 1989.

[10] Jason J. Gauci and Kenneth O. Stanley. Generating large-scale neural networks through discovering geometric regularities. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*. New York, NY: ACM, 2007.

[11] D. E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Reading: Addison-Wesley, 1989, 1989.

[12] J. H. Holland. *Adaptation in natural and artificial systems. an introductory analysis with applications to biology, control and artificial intelligence*. Ann Arbor: University of Michigan Press, 1975, 1975.

[13] Kenneth Alan De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, University of Michigan, 1975.

[14] Hiroaki Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4(4):461–476, 1990.

[15] A. Lindenmayer. Mathematical models for cellular interactions in development. *J. Theor. Biol.*, 18(18):280–299, 1968.

[16] A. Lindenmayer. Developmental systems without cellular interactions, their languages and grammars. *J. Theor. Biol.*, 30(30):455–484, 1971.

[17] Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, Cambridge, MA, USA, 1996.

[18] Norman Richards Risto Miikkulainen, David E. Moriarty. Evolving neural networks to play go. *Applied Intelligence*, 8:85–96, 1998.

[19] *Evolving a Roving Eye for Go*. Springer-Verlag, 2004.

[20] A Guha S. Harp, T. Samad. Towards the genetic synthesis of neural networks. In *Proceedings of the International Conference on Genetic Algorithms*, 1989.

[21] Kenneth O. Stanley. *Efficient Evolution of Neural Networks Through Complexification.* PhD thesis, The University of Texas at Austin, 2004.

[22] Kenneth O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines Special Issue on Developmental Systems*, 2007.

[23] Rémi Munos Sylvain Gelly, Yizao Wang and Olivier Teytaud. Modification of uct with patterns in monte-carlo go. Technical report, INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE, 2006.

Appendix A

Appendix

A.1 Parameters

These parameters are largely derived from the parameters given in [21] and the names used in the settings file are also derived from the actual release of the code from the same dissertation, in this case NEAT v1.1, available from Dr. Kenneth Stanley's web page. Some of the parameters may change in future releases (see section A.2).

The actual parameter word used in the settings file is typefaced in italic. Each of the settings uses a real number as its value, unless its a boolean setting, in which case it is marked as such and the values 0 and 1 apply.

- **Disjoint genes coefficient**
  disjoint_coeff
  This parameter lets you decide how much disjoint genes is considered when calculating the genetic distance between two genomes.

- **Excessive genes coefficient**
  excess_coeff
  This parameter lets you decide how much excessive genes is considered when calculating the genetic distance between two genomes.

- **Mutational difference coefficient**
  mutdiff_coeff

This parameter lets you decide how much the mutational difference, that is the difference in weights, (for each of the aligned genes) is considered when calculating the genetic distance between two genomes.

- **Weight mutation rate**
  weightmutation_rate
  The rate of genes(weights) in the genome which should be mutated, given that the genome already is selected for weight mutation.

- **Severe mutation rate**
  severe_mutation_prob
  A rate for each genome, which greatly increases weight mutation rate, this is fixed to 0.5 in the original version of NEAT, which means that half of the time a genome is mutated, it is a severe mutation.

- **Compatibility threshold**
  compat_threshold
  This threshold lets you decide how genetically close two genomes need to be given the formulae in (4.1) [1]

- **Drop off age**
  dropoff_age
  When the number of generations in which the species has not improved increases beyond this number, the overlap is used as a "agedebt", to adjust the fitness of the species down.

- **Age significance**
  age_significance
  This is a coefficient to the fitness of each phenotype in a species who's age is above 10.

- **Survival threshold**
  survival_thresh

[1] This parameter should be considered as a starting point if you also specify species target, in which case the evolutionary algorithm gradually shifts this value in either direction to try to achieve the right number of species.

This is the percentage of phenotypes to survive within a species, if set to 0.2, the top 20% phenotypes will survive and possibly be selected to have offspring.

- **Species target**
  species_target
  This tells the evolutionary algorithm to adjust the compatibility threshold so that the number of species will converge to this target number. As this algorithm just increases and decreases the threshold by 0.3, the number of species will oscillate around the target, but the average number of species with regards to generations will be quite close to this target number.

- **Species target size**
  species_target_size
  This does the same as species target, in a more consistent way to the experimenter, as the size of the species is what affect the other processes dependent on the species. This parameter is only used if the above parameter "Species target" is not set.

- **Mutate only probability**
  mutate_only_prob
  This is the probability that the offspring is a straight clone rather than a result of a crossover.

- **Mutate weights probability**
  mutate_link_weights_prob
  This is the probability that the weights of the offspring will be mutated.

- **Mutate toggle enable probability**
  mutate_toggle_enable_prob
  This is the probability that one mutates the genome via picking a gene and toggling whether its enabled or not.

- **Mutate gene re-enable probability**
  mutate_gene_reenable_prob

This is the probability that one of the disabled genes of the offspring will be re-enabled.

- **Mutate add node probability**
  mutate_add_node_prob
  The probability that the mutation picks a link, splits it and ads a new node in between.[2]

- **Mutate add link probability**
  mutate_add_link_prob
  The probability that the mutation adds a link between two nodes that doesn't have a link[3].

- **Interspecies mating rate**
  interspecies_mate_rate
  The rate of which a crossover operation is between two phenotypes from different species.

- **Mate multipoint probability**
  mate_multipoint_prob
  The probability that a crossover should be a multipoint crossover, deciding for each gene which parent it should be from.

- **Mate multipoint average probability**
  mate_multipoint_avg_prob
  The probability that a crossover should be like a mulitpoint crossover only that the gene's(or link weights) are averaged between the two parents(unless the gene in question is disjoint or excess).

- **Mate singlepoint probability**
  mate_singlepoint_prob
  The probability that the crossover chooses one point in the genes to use for crossover point

[2]See keeplink option, as this alters the effect of this parameter.
[3]See newlink tries as this alters the behaviour of this algorithm.

- **Mate only probability**
  mate_only_prob
  The probability that the resulting phenotype from a crossover is not mutated.

- **Newlink tries**
  newlink_tries
  This number determines how many times the algorithm for adding a link should try to find two nodes that doesn't have a link between them, this is necessary as there could be the case that this isn't any two nodes without a link between them.

- **Spawn recurrence probability**
  spawn_recur_prob
  The probability that during the random spawning of the initial generation, a given recurrent link will be created.

- **Cold gaussian weight mutate**(boolean)
  coldgaussian_weight_mutate
  This parameter turns on or off cold gaussian weight mutation[4].

- **Recurrent connection probability**
  recurrent_conn_prob
  This is the probability that the link will be added if the link in question a recurrent link.

- **Weight mutation power**
  mutate_link_weights_power
  This sets the maximum weight mutation.

- **Number of innovations**
  number_of_innovations
  This number sets how many innovations the NEAT algorithm should remember.

[4]Cold gaussian implies that the mutation doesn't use the original weight as a starting point, see the code for more detail

- **Keep link while adding node**(boolean)
  keeplink
  If set, this parameter will change the algorithm for adding nodes so that it keeps the original link enabled after creating the two new links and adding the node in between.

A.1.1 Fixed Parameters

Some of the parameters were fixed across experiments as they wasn't affected by the difference in task domain.

A.1.2 Experiment Parameters

This subsection will list important parameters that differ between the XOR and the GO experiment

Setting	**GO**	**XOR**
mutational diff coefficient	0.4	2.0
compatibility threshold	2.0	8.0
drop off age	15	130
species target size	N/A[5]	20
Add node probability	0.08	0.005
Add link probability	0.12	0.1
Inter-species mating probability	0.001	0.015
Recurrence	Enabled	Disabled
Weight mutation power	2.5	1.5

A.2 Source code

The source code containing the implementation is released under the GPLv2(see `http://www.fsf.org/licensing/licenses/info/GPLv2.html`) license from the Free software foundation. This source-code can be found at the following location: `http://neatzsche.generation.no`. The source-code may change

[5]No species target set, the compatibility threshold is used.

in the future, but the results in this paper are based on the releases tagged as 1.0 and 1.1(for fitness equation (4.6) and (4.7) respectively).