$Revision: 1.60 $

THE UNIVERSITY OF QUEENSLAND

School of Computer Science
and Electrical Engineering

Bachelor of Engineering Thesis

Neural Networks in RTS AI

Author: Timothy Adam Smith
Supervisor: Peta Wyeth

Submitted for the degree of Software Engineering
October 2001

Abstract

This thesis presents an environment for studying RTS computer controled players, a package for using neural networks and a package for coevolution. It then uses all these packages to create a computer player that employes a neural network.

Acknowledgements

I would like to thank several people that had an influence on this document:

- My supervisor, Peta Wyeth for everything, most especially for the encouragement.
- Janet Wiles for the document creation aspects of COGS4031
- Paul Darwen for discussions and the use of his library.
- The occupants of room 78-209 for their tolerance of me using more than my fair share of the computers.
- My fellow gamers from the UQNGA, for being real opponents, and for the discussions on RTS AI.
- My family, for their support.

Contents

Part I

Background

Chapter 1

Introduction

1.1 Problem overview

Real Time Strategy games (RTS) are third person games where the player controls troops and often, needs to collect resources[1] Playing an RTS against a conventional computer opponent is substantially less satisfying than playing against a human opponent. This is because a computer plays the game fundamentally differently from a human. The thesis aims to explore the possibility of evolving neural networks as a component of strategy AI, in the hope that neural network opponents will be more satisfying.

1.2 Human opponent-Computer opponent differences

There are many flaws with conventional rule based CPs [2] (computer players). The major flaw being: an experienced player has learnt the CP's blind spots, and can exploit them, time and time again. For example, in Warcraft II, the AI would not attack walls, so by building a wall around a catapult, you could make it immune to melee units (a human player would attack the wall, then attack the catapult).

Another is the tendency of CP programmers to cheat - in order to make a sufficiently formidable opponent, the AI is often given access to more information that a human player is (e.g. Darkreign's AI always knows just

[1] Readers unfamiliar with the RTS genre should look over Appendix A before continuing, as it contains explanations of the terms used here.

[2] In this document, a computer player is referred to as a 'CP', rather than the more traditional (in computer gaming circles) 'AI', because 'AI' has a different meaning in Cognitive Science circles.

where the resources are while a human player has to send out scouts to find resources).

A human is able to access only a small amount of the information about the game state at a time, that is, the specific unit placements in the area they are currently looking at and rough details from the minimap. A CP however can access about the entire map instantly. There is no point, for example, in attacking an computer player's forces on one side of the map, in order to distract it from troop movements on the other side. This technique is effective against a human opponent.

1.3 Aims

The purpose of this project is to produce an environment for studying RTS CPs, and to study the feasibility and usefulness of evolving neural network based CPs.

1.4 Structure of this document

This document is broken up into 4 parts. Part I introduces the problem and provides information about the techniques considered for the production of the CP. Part II explains the design and implementation of the CPs and the RTS and the various software packages used to evolve and implement the CP. The effectiveness and potential effectiveness of evolving CPs is discussed in Part III.

The text occasionally refers to areas of itself. The terms used and the scopes to which they refer are hierarchical. Starting at the largest and progressing down the terms are: part, chapter, section, subsection. This sentence is in Part I, Chapter 1, Section 1.4. This section does not have subsections.

Chapter 2

Theory

2.1 Introduction

In order to make the Literature review (Chapter 3) more concise, the theory that it requires is presented in a separate chapter, here. The following sections explaining neural nets in general, some specific types of neural nets, coevoltion and some coevoltion techniques.

2.2 Neural Networks

Neural Networks (or NN or neural net or net), or more pedantically, Artificial Neural Networks [1] take their name from the biological neural networks that they model to a greater or lesser extent.

Some notes general notes on neural networks ('general' because some of the notes are generalisations that do not apply universally):

- A neural network is a collection of weights and units [2] that maps input states onto output states.
- Some units are input, some are output and some are both.
- A unit that is neither an input unit nor an output unit is called a hidden unit.
- Weights are used to connect a units to another unit.

[1] outside this section Neural Network and NN may safely be assumed to refer to software based Artificial Neural Networks

[2] the term unit is used in this document in two quite distinct ways - to mean a component of a neural net, as here and in reference to the entities a player controls in a RTS. The context must be used to distinguish between to two usages

- Weights have direction. That is, 'unit i_1 connects to unit o_1' is not the same as 'unit o_1' connects to 'unit i_1'.
- Input units' activations are (initially at least) set from outside the network.
- Other units' activations are functions of:
  - The activations of the units that connect to them
  - the weights connecting to them
  - a bias term associated with the unit
  - a scaling function
- one popular activation function (and the one used in the CP's implementation) is:

$$a_x = tanh(\sum_{i=1}^{n}(a_i * w_i) - b_x)$$

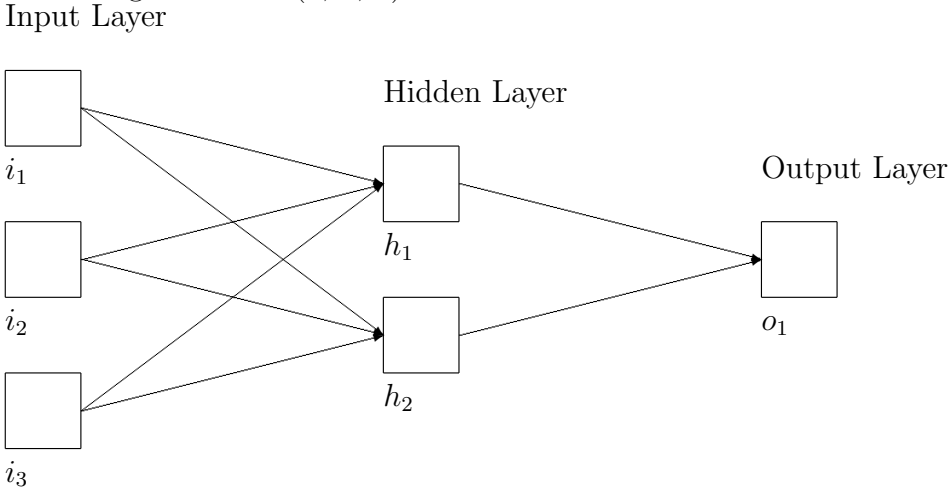
  where
  - a_x is the activation of unit x
  - a_1 to a_n is the activations of the units connecting to x
  - w_j is the weight from unit j to unit x
  - b_x is the bias term associated with unit x
- a layer is a collection of units
- A layer, A is fully connected to another layer, B if every unit in A has a weights connecting to every unit in B.

Three different types of neural networks are covered in the following subsections, the Feed Forward, one of the simpler types, Time Delay and Recurrent.

2.2.1 Feedforward

Feedforward neural networks are a specific type of neural network – one that does not contain any loops, that is, where no unit's inputs depend on the activation of that unit. Feedforward neural networks are often take the form of a several layers of units, with each of an input layer, zero or more hidden layer and an output layer, with each layer fully connected to the previous

Figure 2.1: A (3, 2, 1) feed forward neural network



layer. The structure of feedforward networks of this form can be recored thus:

$$(|I|, |H_1|, |H_2|, \ldots, |H_n|, |P|)$$

where

- $|I|$ is the number of units in the input layer
- $|P|$ is the number of units in the output layer
- $|H_x|$ is the number of units in the xth hidden layer
- n is the number of hidden layers

An example of a (3, 2, 1) feed forward net can be found in Figure 2.1.

If a set of inputs and the correct outputs are available, the weights a feedforward network uses can be tuned to produce the desired output using a method called backprop. Backprop (short for backpropogation) is an error correction method that works by propagating an error signal (generated by comparing the desired and actual output) back through the network, and changing weights based on this signal.

2.2.2 Time Delay Neural Networks

Time Delay neural networks are similar to normal feed forward neural networks. The temporal aspect comes from the additional units that each layer after the first has access too. At time t, layer n is connected to layer $n-1$ at time t and layer $n-1$ at times before t. TD it is covered in depth in [7].

Temporal Difference (TD) learning is accomplished using a method similar to backprop, but modified to use sequences of inputs. The algorithm is written TD(λ). The λ is a parameter specifying how much an error signal is propagated backwards in time, $\lambda = 0$ means the error applies only to the current timestep, while $\lambda = 1$ means the error is not decayed at all. Values between zero and one are also used.

2.2.3 Recurrent

Recurrent neural networks do have loops in them. This makes it impossible to evaluate the network in one pass. Instead an iterative process is used gradually updating all the values. Consequently, recurrent networks are also capable of storing temporal information and unlike feedforward TD nets, they are not limited in the length of them that a particular input can have an effect for.

2.3 Coevolution

2.3.1 Evolutionary Algorithms

Evolutionary algorithms are a class of algorithms that work by iteratively modifying individuals in a manner that favours the production of individuals with greater fitness, where the fitness is some function to be maximised.

2.3.2 Coevolution

In coevoltion a group of individuals (in this case, neural networks) are evolved to a heuristic based on the individual's performance when playing against other individuals (potentially, but not necessarily in another population).

A major advantage of this method over error correction based methods (such as backprop) is that it does not need to be provided with the correct answers, it only requires a way of comparing performance.

Another advantage of coevoltion is the ability to generate a "panel of experts", rather than a single solution. Several solutions may be presented

the problem at once and the another algorithm (called a gating algorithm) is used to decide which solutions' advice to take.

Of course these advantages come at a price. It can be difficult to accurately compare the performance of individuals over generations. This is discussed in the Establishing Progress subsection below.

Another difficulty with unmodified coevoltion is brittle solutions. That is, individuals that perform well against each other, but poorly against unseen individuals. This difficulty is overcome by speciation.

2.3.3 Speciation

Speciation is defined as [5]:

> Speciation refers to any of the numerous modifications to the original GA that overcome this effect [inappropriately clustering around one solution]

Fitness sharing is a speciation method. Essentially, in fitness sharing, fitness is divided between individuals that are near each other.

One way of fitness sharing is implicit fitness sharing (IFS). The implicit fitness sharing method is[5]:

For each individual i:

1. select n random individuals from the entire population
2. find the strategy that scores best against i
3. that strategy receives the payoff

2.3.4 Establishing Progress

Rating individuals based on the population of that generation makes fitness difficult to plot, as only relative fitness is calculated[4]. This difficulty makes it difficult to determine whether the population is improving.

Ancestral Opponent Contests[4] are one possible way of checking for progress. This process simply involves rating current individuals against old individuals, the theory being that if progress has been made, the current individuals will outperform the old individuals. The method will also highlight situations where lack of diversity allows has allowed defences to certain strategies to be lost, after the defences lowered the old strategies fitness.

CIAO[4] (Current Individual Ancestral Opponents) diagrams are useful for displaying these contests. A CIAO diagram represents the an individuals scores against all its ancestorys using squares varying from black to white.

Chapter 3

Literature Review

3.1 Introduction

No existing literature about using a neural network in a real time strategy game was found. This makes the problem more interesting, but also more difficult.

Fortunately, there is a wealth of information about related fields that can be applied. There are four basic components to the system: the type of the neural network, how it receives information, how it outputs information and how the weights and biases are obtained. Each of these sections is addressed in turn.

3.2 Input Methods

The manner in which data is provided to the network has far reaching implications. The number of units used affects the size of the network. The signal to noise ratio (how much useless information is provided) and the size of the network affects the performance of different learning methods.

3.2.1 Topographic

The simplest solution is to simply take each position on the board and represent it with one unit. Different piece types can the be represented as different activations of that unit.

This works well for Tic-Tac-Toe with only nine board positions and two piece types (X and O)[2]; checkers with 32 board positions and four piece types (black checker, black king, red checker red king)[3]; 3×3 Dots and Boxes with 18 board positions and one piece types (edge)[13].

This method is infeasible for games where the range of piece types cannot be represented as a single real number, such as Chess. A Castle and a Bishop may have the same average worth , but a network that can not distinguish them is at a disadvantage. Alternately, multiple inputs can be provided for each board position, each input representing a different unit type, but this rapidly results in a very large number of inputs.

3.2.2 Feature Vector

The other alternative is to preprocess the board before presenting it to the network[12]. This (usually) has the advantage of reducing the number of inputs to the network which in turn speeds learning and operation of the network. The disadvantage is the network is constrained to use the features provided to to it, so a hole in the domain knowledge can be propagated onto the network.

3.3 Output Methods

There are two possible methods for expressing the networks decision. Rating based, or decision based.

3.3.1 Decision Based

In decision based output, the network explicitly outputs the next move. For games that have only one degree of freedom for a move - Tic-Tac-Toe[2] and Go[9] and Dots-and-Boxes[13] for example - this method can be used reasonably easily, one output unit for each board position and the most active legal move is chosen.

Then there are games where there are more degrees of freedom, where for example the piece to move and the location to move it to must be determined. This method is difficult to use with this type of game, because of the complexity involved in representing the networks decision in a manner that does not involve an unusable large number of units.

3.3.2 Rating based

Rating based output uses a single output unit to rate how advantageous a board state is for a player. If the games branching factor is low enough the rating can then be combined with a search such as min-max to determine

the next move[3]. Alternately, a search depth of one may be used, just rating every board state that is a child of the current state.

This method is only unusable where the game's branching factor makes it unreasonable to make the search deep enough for the network to profitably distinguish states.

3.4 Learning Method

The two covered methods of training neural networks for games are: Temporal backpropogation and Evolution.

3.4.1 Temporal Difference Learning

TD learning has been used successfully for Chess[12] and Backgammon[11]. It has the advantage of being more efficient than evolution. Two major disadvantages of TD learning are its complexity and the more more limited search of the search space.

3.4.2 Evolution

EA have been used successfully with Checkers[3], Dots-and-Boxes[13], Tic-Tac-Toe[2] and backgammon[8]. A big advantage of evolutionary algorithms (EAs) rather than error correcting algorithms is that EAs can be applied to any type of network.

3.5 Neural Network Type

Each type has advantages and disadvantages. Recurrent is the most flexible and the most complex. TD has least some temporal context and is less complex. Lastly, standard feedforward is simplest, but such nets can do utilize historic input.

3.6 Conclusion

In the end evolution of standard feedforward networks was chosen for the CP. It seemed simplicity was the best approach when breaking new ground.

Part II

The System

Chapter 4

The System

Quite a lot of code (more than 4900 lines of Java and over 700 lines of test scripts) was written for this project, and this part covers that code. This chapter acts as a brief introduction to the system and this part. It talks about how to read this part, then briefly explains the high level structure of the code.

Some Java terminology is needed to clearly explain the structure of the system and its parts. A reader unfamiliar with Java and Object oriented Programming may find reading this part easier after reading Appendix B which briefly explains the terms 'package', 'class' and 'method' as well as some naming conventions. A reader may also choose to skip the implementation sections, as these sections do not contain information necessary for understanding other sections of the document.

The code for this system was written in Java, by the author, expressly for this project. All the packages included in the product were written by the author or by the distriutors of Java[1]. This was not the original intent, but it was necessary (details of the original plan, and why it was not followed, can be found in 5.2 and 7.2). The code can is divided into three packages:

- `fourgh.surts` - The package that provides the RTS, and RTS CPs.
- `fourgh.nn` - The package used by the RTS CPs for Neural Network tasks.
- `fourgh.evolve.coevolve` - The package used to evolve the RTS CPs.

While these packages are all required by the CPs, they also have general use independent of the CPs. The following three chapters cover then general

[1]The test cases used the roast package, but they do no constitute part of the product

use, implementation and direction of the three packages. The last chapter in this part covers the CP specific classes a in `fourgh.surts`.

The `fourgh` in the package names is used to make the package names unique to the author. This uniqueness prevents potential ambiguities if the packages are used in conjunction with packages from other sources. For example, without the unique package name, two packages that define a `Unit` object could not be used together because Java can not have different classes with the same full name.

Chapter 5

RTS

5.1 Introduction

One of the prerequisites for writing an RTS CP is a suitable RTS. Originally, the plan was to modify a RTS called meadow (lowercase spelling is intentional) for use on this project. When this plan proved unusable, a new RTS, SURTS, was written from the ground up. This chapter begins by explaining meadow and the problems with it. Subsequent sections cover information about SURTS as a game, information about SURTS the package and possible future features for SURTS.

5.2 meadow

Meadow is a reasonably standard RTS. It supports 2 to 8 players in a single game. The playing field is a large area of land, with the occasional water obstacle. The player starts with a group of tanks, which can be used to attack enemy tanks, or to pick up crates. Picking up a crate gives the player an additional tank. There are various different types of tanks capabilities. This lead to the first, and key, problem.

The problem was one of complexity. The game was too complex at the tactical level, as it had units with different ranges, speeds and damage settings. At the same time, the game was insufficiently complex at the strategic level. The only strategic choice was 'stand and fight' or 'collect crates'. A good computer player would have to focus much more on the tactical level, which was not the objective here.

The most obvious problem was that games took a long time to run. Long running times makes evolution of a computer player infeasible, as each generation requires a sizable number of games (usually well in excess of one game

per individual in the population). This problem could probably have been fixed with some work optimising the code, and possibly simplifying some aspects of the game to improve speed but it was decided that there was a better way.

Individually these difficulties may have been addressable. Taken together it was decided writing a new RTS would be a more efficient way of obtaining a suitable game.

5.3 Introduction to SURTS

SURTS stands for Simple *mumble*[1] Real Time Strategy game. SURTS's major design goal was to be as simple as possible while still providing sufficient strategic elements. While the computer player is technically part of SURTS, it is big enough and important enough to have in its own chapter, Chapter 8.

5.3.1 Units

The 'simple' aspect was achieved by having only two unit types, Tank and Factory. Tanks and Factories are identical except in one important respect. Factories produce units, while Tanks damage and destroy units. Tanks and Factories move at the same speed (yes, the Factories move, and can produce units while moving) and can sustain the same amount of damage before being destroyed.

5.3.2 Winning state

The goal of the standard SURTS game is to be the only team with units on the map. This victory condition can be easily modified. For example, the objective could be set to 'destroy all enemy factories'. However, all games for this project used the classic destroy all enemy units objective.

5.3.3 Production

Every time a *unit* (factory or tank) is destroyed, a factory selected at random from all the factories remaining on the map (regardless of team) builds a new *tank* at the factory's location. The newly created unit is on the same team as the factory that built it. The team of the destroyed unit is not necessarily

[1]The 'U' doesn't stand for anything, it was just added so the acronym could be pronounced

the same as or different to the team of the factory that builds the unit, so the more factories a player has, the more likely that player is to get a new Tank when a Tank is destroyed.

5.3.4 Map

An RTS's 'Map' is equivalent to a board game's 'Board'. SURTS's map is simple, it is just a large rectangle with units on it. The Map is not complicated by obstacle and different terrain types found in most RTSs. SURTS is very flexible. It supports an unlimited number of teams and an unlimited[2] sized play field. The experiments in this thesis

all use 2 teams, on a 33×33 sized map, with 3 randomly placed factories per team and 6 tanks around each factory. This is referred to

as the 'standard rating map', as it was used for rating CPs against each other. Figure 5.1 shows a newly initialised standard rating map.

5.3.5 Strategy

SURTS has three strategic areas that resources (in this case, units) have to be divided between:

1. Protecting friendly factories
2. Removing enemy factories
3. Removing enemy tanks

5.4 Gameplay

While the aim of a RTS is to be a continuous (i.e. not turn based) game, digital computers are incapable of being truly analogue. A non-turn-based effect is achieved by having many turns per second. Each of these rapid turns is called a game tick, or just tick. In games involving humans, SURTS runs at approximately ten ticks per second. In games where both players are computer controlled, such as those used during evolution, ticks progress as fast as computing power allows. It takes well under $100ms$ to produce orders for 40 units and to effect one ticks worth of moving and shooting.

Each tick, each unit has the choice of moving, shooting, or doing nothing. The procedure for a tick is detailed in Figure 5.2. A game is simply tick after

[2]unlimited, that is, except by memory limitations

Figure 5.1: Colour is based on team. A filled circle is a Factory, a hollow circle is a Tank. Fractions of a circle represent damaged units – the percentage of the circle drawn is based on the unit's health divided by the maximum possible health for the unit (there are no damaged units in this figure). The number (1) in the top left corner indicates the age of the map in ticks

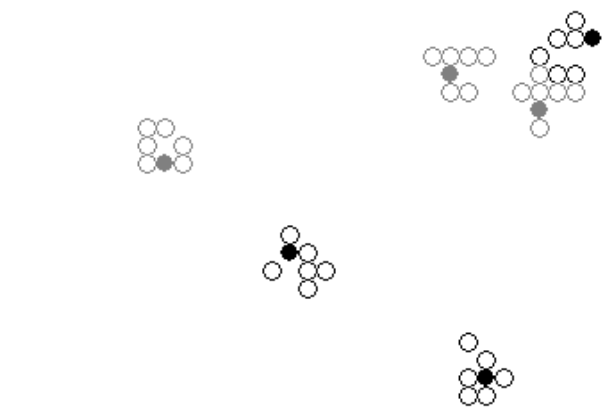


Figure 5.2: The procedure for a game tick

1. for each player, have the player set their unit's orders
2. Starting with the oldest unit and moving to the newest, allow each unit that to move.
3. Allow each tank that did not move to attack, again starting with the oldest and moving to the newest. A tank that is destroyed in a tick before it shoots is still allowed to shoot in that tick.
4. Sleep for $100ms$ if a human is playing/watching

tick after tick, until only one player has units left, or until a draw is declared (for this project a draw was declared if neither side had won after 1000 ticks).

5.5 Implementation Details

SURTS is written in Java, and uses only packages written expressly for this project. The SURTS specific classes explained here are all from the `fourgh.surts` package.

The following list explains the purpose of each class in the `fourgh.surts` package (excluding those classes that are specific to CPs, which are covered in 8):

Control responsible for creating and maintain maps. Includes the implementation of tick (as explained in Fig. 5.2).

Map responsible for storing information about the game. Specifically, it holds its own age and it maintains several collections of units to allow efficient access to important subsets. Multiple references to single units are maintained, for efficient access via various methods. The collections are:

- `Unit map[][]` - an array representing the board. Allows finding a unit at a position to be $O(1)$.
- `Vector units;` - a set of all the live units in the game. Allows easy enumeration over all units in the game.
- `Vector tunits[];` - an array of sets of units, indexed by team.
- `Vector factories;` - a set of all the live factories in the game

– `Vector tfact[];` - an array of sets of factories, indexed by team

Team for storing information about a team. It records a team's name, colour and number. A team's number is the index reserved for it in arrays that are indexed by team. Users of the class are required to ensure that teams' numbers are continuous and start at 0.

Order records one of three possible types of instructions to a `Unit` – move, fire or do nothing. It also stores a location to move to for move orders and an attack preference for fire orders. The attack preference specifies whether, if given the choice, the unit would prefer to attack an enemy factory or an enemy tank. The `Order` objects are immutable. That is, the class implements no set methods so a user of the class can safely assume that any get method on the same object will always return the same value.

Unit the basic unit type: stores the unit's location, health and orders. Also provides initial health and move rate for the unit and a method for drawing units.

Tank extends `Unit` to provide attack capabilities by providing `attack(Unit u)` and `getDamageRate()` methods.

Factory extends `Unit` to provide construction capabilities (actually Factory just overrides the paint method so that factories are drawn differently, other classes are responsible for detecting the Factory subtype and giving it construction).

Place specifies a location on the map, that is, it contains two integers x and y. The class also provides methods for measuring distance between two `Place` objects and methods for creating new place objects a random locations near or specified relative to other `Place` objects. `Place` objects are also immutable.

Target All objects that have a location associated with them implement `Target`, that is, they have a `getPos()` method that returns the `Place` associated with the object.

Display interface implemented by any object that can display a Map. Also requires that objects can state the delays they want between ticks and games.

DisplayApplet an applet that implements the `Display` interface, used when a human wants to watch CPs fighting.

TEST A class that runs automated tests on the `fourgh.surts` package. It is automatically generated from `TEST.roast`.

Chapter 6

Evolution Framework

6.1 Purpose

To evolve a CP, software for evolving CPs is required. In the interests of reuse, a generic framework for coevoltion was written, capable of evolving any entity provided certain forms were followed. Note that the framework currently only handles one population coevoltion (where an individuals fitness is relative to the individuals in its population), it does not support multi-population coevoltion (where an individual fitness is relative to another population, e.g. predator and prey populations). The framework also supplies a method based on CIAO (see 2.3.4) for displaying the results of evolution.

To evolve a type of entity, the framework uses several things that need to be supplied to it:

- A way of mutating the entity to be evolved
- A rating method for comparing pairs of individuals
- A source of entities for generating the initial population
- A choice of prewritten selection method, or a new selection method.
- (optionally) A source of entities to be introduced during evolution, to be added to the population in addition to the entities created via mutation.
- (optionally) A tracker to use for recording the entities evolved

6.1.1 Rating Method

Each entity type needs its own rating method. A rating method takes two individuals and returns a score for each individual based on a comparison. For example, if the individuals play simulated basket ball, the two ratings returned can be the individual's respective scores.

The individuals are provided to individuals in a set order and the rating method can treat the first and second individuals differently, at least when being used with the selection methods detailed in 6.1.2. A rating method does not need to be commutative as long as any bias is consistent. That is, to use the basketball example again, if the first individual's team can be given a wind advantage, as long as the advantage is always given to that team. This in-commutativity is possible because the selection methods rate individuals in both directions.

6.1.2 Selection Methods

The framework allows for a rating method to be selected from several prewritten methods, or for a specialised method to be used. The prewritten methods are:

RR Round Robin involves rating every individual in a generation against every individual in the generation (this includes rating an individual against itself). Each individual's fitness is increased by that individuals score from the rating.

RR2 Round Robin 2 runs the same set of ratings as RR. The difference in RR2 is for each rating the individual that score highest gets one additional point of fitness, regardless of the difference in scores.

IFS reproduced from 2.3.3: For each individual i:

1. select n random individuals from the entire population
2. find the strategy that scores best against i
3. that strategy receives the payoff

IFS ignores i's score, comparing only the random individuals' scores. This fact should be considered when writing rating methods to be used with IFS. n defaults to 6, but the value can be changed.

6.1.3 Process

When it is started, the framework forms a population using the individual source provided for that purpose. It then repeats a simple series of steps until it is killed:

1. duplicate all individuals in the population, the mutate the duplicates
2. add any additional individuals for other sources
3. rate the population
4. send the population and ratings to the tracker, if one exists
5. cull the population - remove the least fit members of the population

6.1.4 Analysing Populations

Also provided is a means for analysing and displaying the analysis of series of generations. IGC, a variant of CIAO, Inter-Generation Contests diagram diagram. The diagrams used in [4] only pitted individuals against older individuals. Due to the possible asymmetry in the rating method, competitions with descendants of your opponents can convey additional information. Additionally, IGC diagrams are normalised across the entire diagram, rather than by row. This allows them to be read both down and across. The top row of an IGC diagram shows ratings of the most fit individual from the first generation. Each row then uses the elite individual from a generation after the row before it. The individual used for a column is the same individual used for the corresponding row, starting with the first individual in the left-most column. The individuals in the columns are the individuals that the individuals in the rows are being rated against.

6.2 Implementation

A general coevoltion package, `fourgh.evolve.coevolve` was written to handle the coevoltion. It is capable of coevolving any object that implements the `Individual` interface. The `Individual` interface requires that an object can rate itself against another `Individual`.

The main class in the package is `Island`. An Island's constructor takes four parameters, two `IndividualFactory` objects (called `factory` and `newblood`), an `IndividualTracker` (called `tracker`) and a `String`, `nodename`. The `factory` parameter is to produce the initial population, while `newblood` is

optionally used to introduce new `Individual` objects from an avenue other than mutation. Every generation, the `IndividualTracker` object is used to save the generation's details.

Different rating mechanisms can be implmented by overriding the `ratePop` method of an Island, or by simply recompiling `Island` after rewriting the method.

The `DiskIndividualTracker` can be used to save `Generation` objects, which can then be passed on the command line to `IGC`, which in turn generates an output file readable by `IGCApplet`.

6.3 Future

While configuration by overriding methods worked all right for this project, any further use of the framework would justify the effort of introduction of some sort of generic configuration object that could be used for setting the parameters of the run, and passed to the `Individual`, `IndividualFactory` and `IndividualTracker` objects to customise their behaviour.

Chapter 7

Neural Network Framework

7.1 Purpose

A package was required to encapsulate and perform the neural network aspects of the CP. When JavaNNS proved less than ideal, `fourgh.surts` was written.

7.2 JavaNNS

Originally, the plan was to use the Java Neural Network Simulator (JavaNNS)[6] as the neural network package. JavaNNS's licence allows it to be used as a library, but it is designed to be used as an application. As consequence of this design the interface specification is not documented.

In order to work out how to use the JavaNNS as a library, several reengineering tools were applied to it. Rigi[14] and ClassToRsf[10] were used to analyse the way JavaNNS constructed and used neural networks.

It turned out that JavaNNS uses `native` methods to access a dynamically linked library that contains the core neural network functions. Unfortunately the native method interface used files to communicate. It was decided that writing a neural network package would be easier than getting JavaNNS working and putting up with the limitations that it entailed (requiring temporary files makes using neural networks in applets complicated).

7.3 Implementation

The classes in `fourgh.nn` can be divided into three groups: unit types, network types and examples. The unit types are:

NNUnit the `NNUnit` abstract class represents a unit. It specifies two methods, `calcActivation()` which is responsible for calculating the unit's activation and `getActivation()` which is responsible for returning a cached value from the previous calculation.

ConstantUnit a unit that always has the same activation

Weight weights are represented as an extension of `NNUnit` that multiplies the activation of a `NNUnit` by some value and uses that as its activation.

HidUnit a unit that returns the sum of the activations of the weights bound (or indeed, any type of `NNUnit`) to it, less a bias. It can used as both a hidden or an output unit. A biases could have been expressed as weights bound to `ConstantUnit` objects, but including them in `HidUnit` is both simpler and more efficient (saves the creation of two extra objects for each bias).

TanhHidUnit an extension of HidUnit that does tanh scaling.

InUnit a unit whose activation can be set.

In order to provide for network types other than feedforward, `NNUnit` objects are required to use `getActivation()` on `NNUnit` objects they referenced, rather than calling `calcActivation`. While a recursive method of updating activations would work for feedforward networks, with a recurrent network a recursive update system would recurse infinitly. Infinite recursion may be avoidable by adding extra parameters to `calcActivation()`, but this would be excessively complex and inflexible (limiting users to only one method of updating networks). As a consequence of this design decision, an object is required to oversee the updating (i.e. calling `calcActivation()`) of units. This duty can be incorporated into the objects that would be needed anyway to store and create collections of units (i.e. neural networks).

`FCFFNet` is the only such object currently implemented. It creates and maintains fully connected feedforward networks with one hidden layer.

Lastly there is the example series of classes. They were used for testing both `fourgh.evolve.coevolve` and `fourgh.nn`. The `Xor`, `Xor2` and `Xor3` are (2, 2, 1) feedforward NNs capable of outputing a value representing the result of xoring the two inputs. They all implement `fourgh.evolve.coevolve.Individual`. The rating method returns the negative of the sum squared errors of the two individuals passed to it. While this technically violates the agreed format of the rating method because the networks do not interact, it follows the agreement well enough to allow evolution.

The difference between the three Xor classes is how they use the `fourgh.nn` package. `Xor` uses and updates `NNUnits` itself. `Xor2` implements Xor by extending `FCFFNet` and `Xor3` simply uses a `FCFFNet` object.

7.4 Future

There are several features that would improve the package:

- TD units - A unit type whose activation could be bound to be the activation of another unit, with some time delay.
- TD Network class - for building and updating networks that use TD Units.
- Recurrent Network class and unit types or wrappers that can be updated iteratively.
- learning algorithms - a way of obtaining weights and biases other than mutation.

Chapter 8

Computer Player

8.1 Introduction

The computer players are all subclasses of the `fourgh.surts.Strategist` abstract class.

8.2 L0Strategist

The L0Strategist is an extremely poor CP. When called, it simply orders all of its units to move toward the centre of the map. It never attacks. It is useful for rating other CPs, and for testing the RTS.

All L0Strategists are identical as they have no variables. As a consequence, the mutate method simply returns an new L0Strategist.

8.3 L1Strategist

The L1Strategist is only slightly more complex than the L0Strategist. It has two variables, `attackprob` and `attacktank`. The value of `attackprob` controls the likelihood that a unit will try to attack during a tick. If the unit does attack, `attacktank` determines the chance that it will attack Tanks preferentially, or whether it will attack Factories preferentially. If the unit does not try to attack, then it simply moves toward the centre.

The mutate method on a L1Strategist varies attacktank and attackprob independently by a random amount in the range $(-0.1, 0.1)$.

Early evolution testing, before draw handling framework was in place, used a L1Strategist with *attackprob* set to 0.8 if mutation put it above 0.8. This ensured that units always eventually do move near enough to each other

to attack. Without this restriction, and with no limit on game length, games can last forever because some L1Strategists never order their units' to move.

8.4 NNStrategist

NNStrategist is a CP that uses a neural network at the centre. It works by preprocessing units into forces (groups of units), feeding the network information the sizes of those forces, running the net, and then interpreting the output as behaviours for the forces.

The preprocessing divides the map into qc different quadrants using a rectangular grid. It makes one enemy and one friendly force per quadrant (these forces may contain zero units). The friendly force at quadrant q is called F_q, and it contains all the friendly units the quadrant. Friendly units are units under the NNStrategist's control, that is, units on the NNStrategist's team[1]. The enemy force at quadrant q is called E_q, and is all units in quadrant q that are not friendly units.

A single value is then generated to represent the strength of the CP's team in each quadrant. This value is generated using the formula:

$$qin_q = \frac{(ft_q + ff_q * fv) - (et_q + ef_q * fv)}{scale}$$

where

qin_q is the input value derived from quadrant number q

ft_q is the number of friendly tanks in quadrant q

ff_q is the number of friendly factories in quadrant q

et_q is the number of enemy tanks in quadrant q

ef_q is the number of enemy factories quadrant q

$scale$ is $uc/3 + 1$

uc is unit count, the total number of units of any type and affiliation on the map.

fv is the factory value, a variable that is evolved in the NNStrategist

[1]SURTS does not allow allies, so these two sets are identical

The neural network has one input unit for each quadrant ($in_1..in_{qc}$), three more hidden units than quadrants ($hid_1..hid_{qc+3}$) and one output unit per quadrant ($out_1..out_{qc}$). The input units, to the value for the corresponding quadrant. The network itself is fully connected feedforward. The input units simply provide the network with the values of *qin* without additional scaling, while the hidden and output units have biases, and use tanh scaling.

The outputs are used to set the behaviour of the friendly units in the corresponding quadrant. Three behaviours are possible:

camp The units attack if they can (that is, there is an enemy in range and they are tanks), otherwise they they move toward the centroid of the friendly force.

sand Search and destroy - If the quadrant has enemies, the units attack if they can and move about at random if they can not attack. When the quadrant contains no enemy units, the units move toward the camping force with the highest qvalue. If no camping forces exist, then they move toward the occupied (by friends/enemies or both) quadrant with the highest qvalue.

flee The units run toward the force sanding force with the highest qvalue, regardless of whether they can attack or not

If $qout_q$ is less than -0.3 then the F_q's units are set to *sand*. If it is greater than or equal to -0.3 and less than 0.1 then the behaviour is set to *camp*. If it is greater than 0.1 then *flee* is used.

8.4.1 Rationale

Several decisions were made when designing the NNStrategist. This subsection explains the reasons behind the choice of neural network type, size and the feature representation method.

The aim of this CP was to use the minimum useful Neural Network.

8.4.2 Implementation details

The CP related classes are:

Features contains static methods for creating features

Force stores a collection of units and provides `getCentroid()`, which is used when a unit moves towards a force.

Strategist	An abstract class that implements `fourgh.evolve.coevolve.Individual`.
L0Strategist	a trivial extension of `Strategist`
L1Strategist	a slightly less trivial extension of `Strategist`
NNStrategist	an extention of `Strategist` that employes a neural network.
NNStats	contains static methods for extracting information from generations of `NNStrategist` objects.
PostOrder	An extension to `Order` used by `NNStrategist`. It allows destination to be set *after* the order has been assigned to a `Unit`. This allows the iteration over the forces setting orders to be done in a single pass. This process does not cause any conflicts with the assumption that `Order` objects are immutable because `Unit`'s orders are not accessed during the order setting loop.
SF1	SURTS's implementation of `fourgh.evolve.coevolve.IndividualFactory`, variously compiled to provide any of the three different `Strategist` classes in the `fourgh.surts` package.

Part III

Outcomes

Chapter 9

Results

The 'Outcomes' part of this document details the experimental outcomes, rather than the software outcomes.

Results occur in phases. Each phase consists of one or more runs followed by an analysis of the resulting CPs. A 'run' is the evolution of CPs under a specific set of conditions. Runs are usually substantial undertakings, taking some time to set up, and quite a long time to process.

To avoid confusion and mistakes, each run was given a codename, the name of a species of bird. This naming scheme avoids confusion because it is much harder to mistake 'hawk' for 'eagle' than it is to mistake 'RR:20:5:-1' for 'RR2-20:5:-1'. Similarly, the the bird name is harder to mistype in a manner that is a valid identifier.

Phases are detailed in chronological order, with one section devoted to each phase. Each section has the parameters for the runs in the corresponding phase, the details of the analysis and how those results motivated the next phase. The next two sections provides explanations of the variables that distinguish runs and a complete listing of variables for all runs.

9.1 Variables

Three major things can change between runs using NNStrategist, the rating method, the selection method and the population size (*pop*). The rating method controls how a CP is compared to one other CP. The selection method (*sm*) is what controls how combinations of ratings are used to determine which CPs survive between generations. The value of *pop* controls the base number of CPs in the population. Just over twice this number of CPs are actually rated each generation, as one individual with random weights and a mutated copy of each individual are added just before the rating stage.

The rating method always involves a CP playing one or more games against a CP (possibly itself). What varies here is the number of games played for a single rating. This variable is called *gpr* (games per rating). The other rating method variable is $drawv$, how much a value is given to both players in the even of a draw (in comparison with 1 for a win, and 0 for a loss). The advantage of having drawv variable, rather than simply setting it at 0.5, is that by setting $drawv$ to a negative value, -2 for example, draws can be actively discouraged under some selection methods. All three selections methods described in 6.1.2 are used. Note that $drawv$ has no effect in RR2 selection. The value of n for IFS is referred to as $ifsn$ in this section.

9.2 Runs

This section details the parameters used in all the runs, in all the phases.

phase	codename	variable	value
2	hawk	gpr	20
		sm	RR
		drawv	-1
		pop	5
2	eagle	gpr	20
		sm	RR2
		drawv	-1
		pop	5
2	kite	gpr	20
		sm	IFS
		ifsn	4
		drawv	-1
		pop	5
2	falcon	gpr	20
		sm	IFS
		ifsn	6
		drawv	-1
		pop	5
3	swan	gpr	200
		sm	RR
		drawv	0
		pop	5
3	drake	gpr	200
		sm	RR
		drawv	-2
		pop	5
3	duck	gpr	200
		sm	IFS
		ifsn	6
		drawv	0
		pop	5
3	goose	gpr	200
		sm	IFS
		ifsn	6
		drawv	-2
		pop	5
4	gull	gpr	20
		sm	IFS
		ifsn	6
		drawv	0
		pop	50
4	turn	gpr	20
		sm	IFS
		ifsn	6
		drawv	-2
		pop	50

9.3 Phase 1

Phase 1 involved testing the system using L0Strategist and L1Strategist. It also involved some informal experimentation with NNStrategist.

This testing discovered was the need for handling draws. When, for example, two L1Strategists both with 0% chance of moving play against each other the game would never terminate because some tanks never got close

enough to each other to be attacked. One possible method for detecting a draw of this type is to declare a draw when no shot has been fired for a sufficient number of ticks. Unfortunately, this method does not handle 'live' draws.

One surprising common (with completely new individuals from generation 1) example of a live draw is when one team has a single unit left, which is set to camp, and the other team has a large force of units including some factories which, because of particularly bad initial values, are fleeing towards the attacking unit. The difficulty here is fleeing units do not attack, so the single individual survives, but because the other team had all the factories, so the fact that the individual destroyed an enemy tank every 3 ticks on average has no effect, because the other team produces tanks at the same rate.

Because of complications like this, a more foolproof method of determing draws was introduced: game were limited to 1000 ticks.

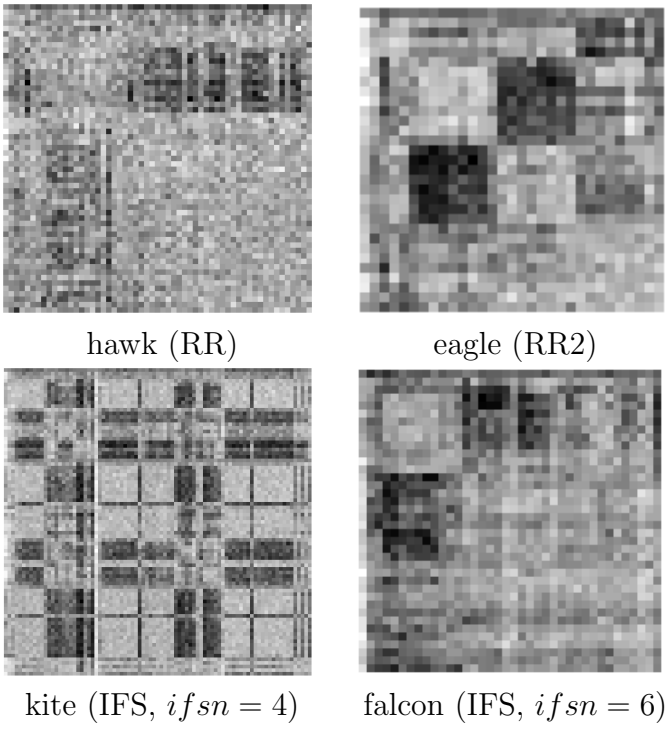
9.4 Phase 2

Phase 2 consisted of 4 runs, each with a different selection methods, but all with *gpr* at 20 and *drawv* set to -1, in order to discourage the 'run and hide' strategies that are both time consuming in the evolutionary process, and annoying to play against. The details of the difference between runs in this phase are covered in the captions for their IGC figures (Fig 9.1). IGC diagrams were explained in 6.1.4.

The symmetry of these diagrams suggests something was wrong. If the rating scheme was symmetric, the diagrams should be anti-symmetrical around the line $x = y$ (a line from top right to bottom left). The only thing that is not anti-symmetric in the ratings is caused by the negative *drawv*, which disadvantages both parties in the event of a draw. This suggests that the variability of the picture is being burnt out by draws. Implementation limitations prevented the IGC being re-run with a different rating method (i.e. with $drawv = 0$). In the next phase that limitation was eliminated.

9.5 Phase 3

Phase 3 makes several changes from Phase 2. Experiments with a very rough simulation of the NNStrategist (approximating a NNStrategist as a coefficient use to multiply a random number during rating) suggested that $gpr = 20$ was too small, so *gpr* was increased to 200. In order to better investigate the



hawk (RR) eagle (RR2)

kite (IFS, $ifsn = 4$) falcon (IFS, $ifsn = 6$)

Figure 9.1: Phase 2 IGC diagrams

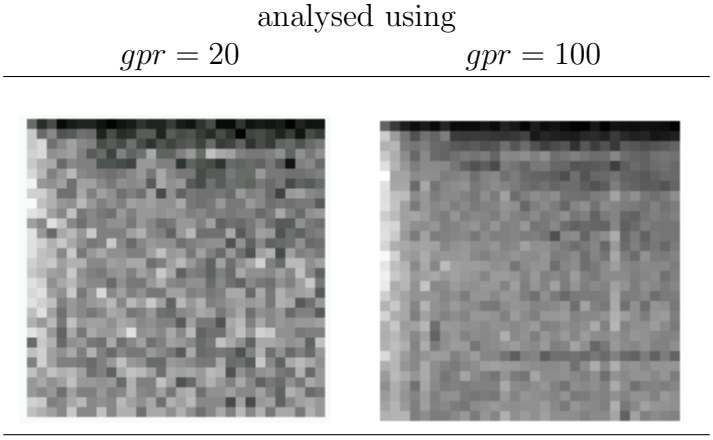


Figure 9.2: the effects of *gpr* on IGC diagrams, the duck run analysed with different *gpr* values and $drawv = 0$

effects of *drawv*, the selection methods were (somewhat arbitrarily) reduced to just RR and IFS with $ifsn = 6$ and runs were conducted with both methods with $drawv = 0$ and then $drav = 2$

Figure 9.5 gives an example of the increased smoothness that analysis with $gpr = 100$ gives over $gpr = 20$. This smoothness stems from the improved accuracy of ratings.

The $drawv = 0$ diagrams shown in Figure 9.3 show progress more clearly than those from the previous phase (Fig. 9.1. The two sets can not be properly compared though, as the phase 2 figures use a higher step size.

The black lines in the swan and goose analysis indicate that the strategies formed by those methods are vulnerable to run-and-hide strategies. Unfortunately, without using an order of magnitude more computer power to run each set of parameters more than once each, it can not be established whether which runs are afflicted with these lines is due to coincidence or the parameters.

Another interesting way of showing the progression of CPs over generations is by how much the NNStrategist values factories. Figures 9.4 and 9.5 graph this.

A possible cause to a susceptible to the run-and-hide problem is the low population used. The final phase explores this possibility.

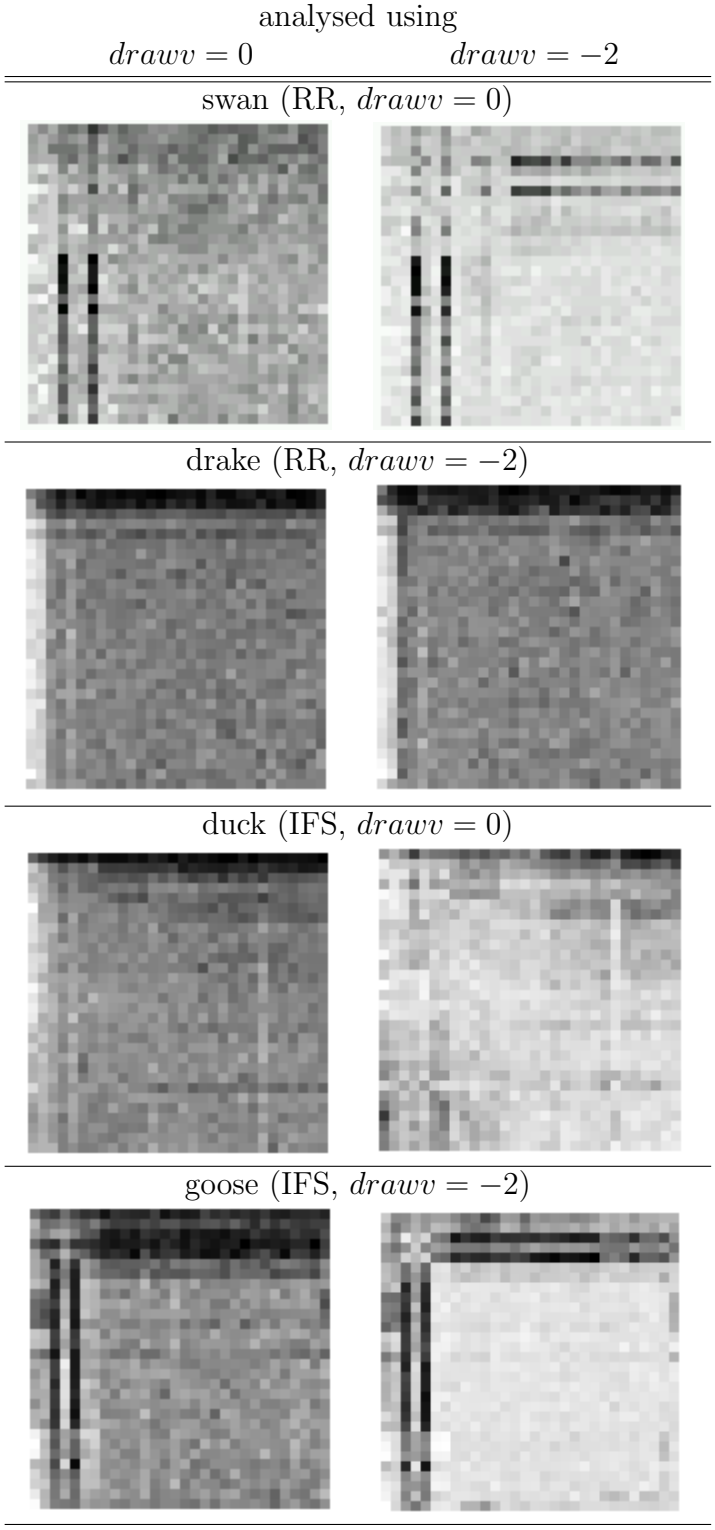


Figure 9.3: Phase 3 IGC diagrams. Diagrams created with $gpr = 100$

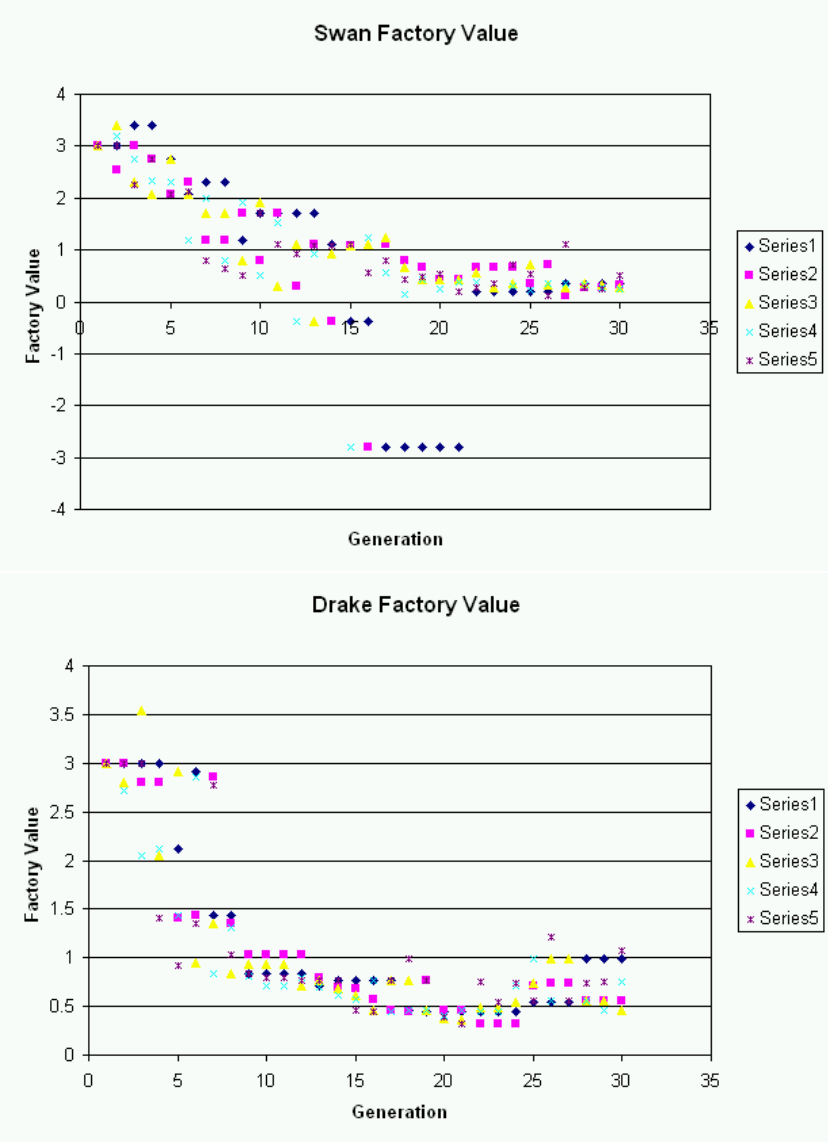


Figure 9.4: The Factory values for the five oldest strategies in each generation, phase 3, RR

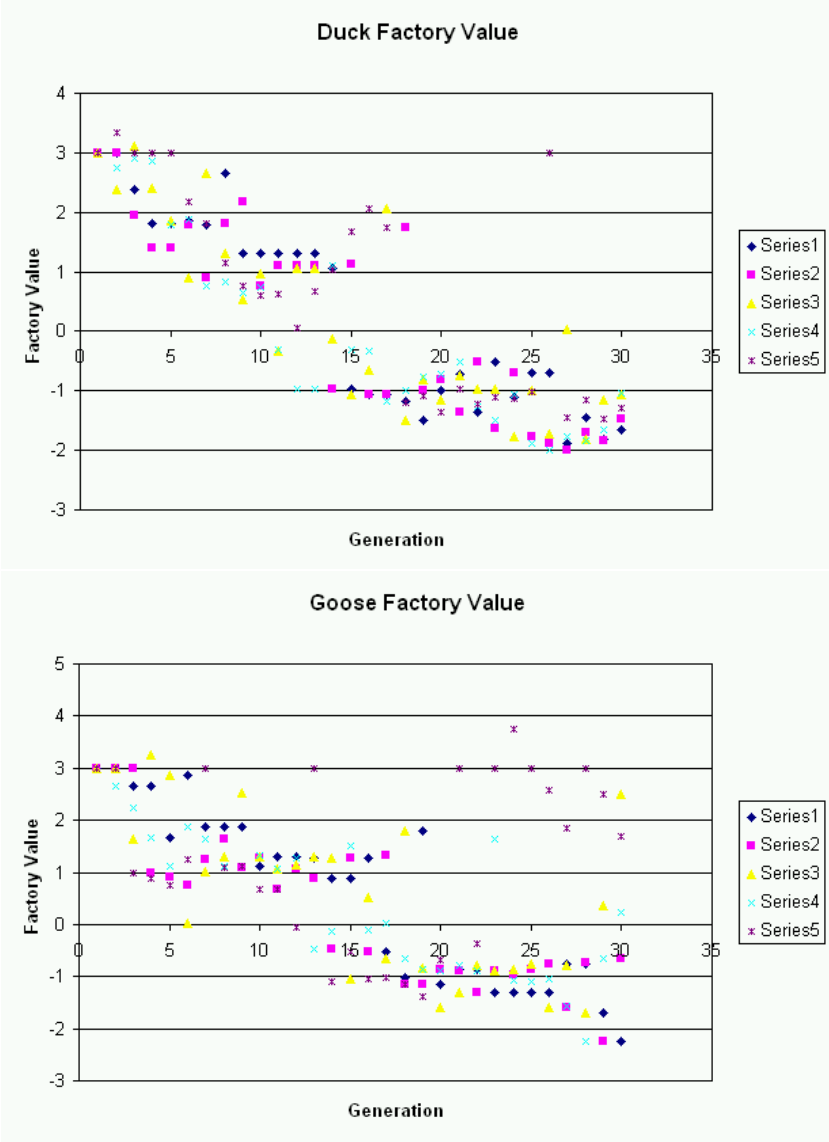


Figure 9.5: The Factory values for the five oldest strategies in each generation, phase 3, IFS

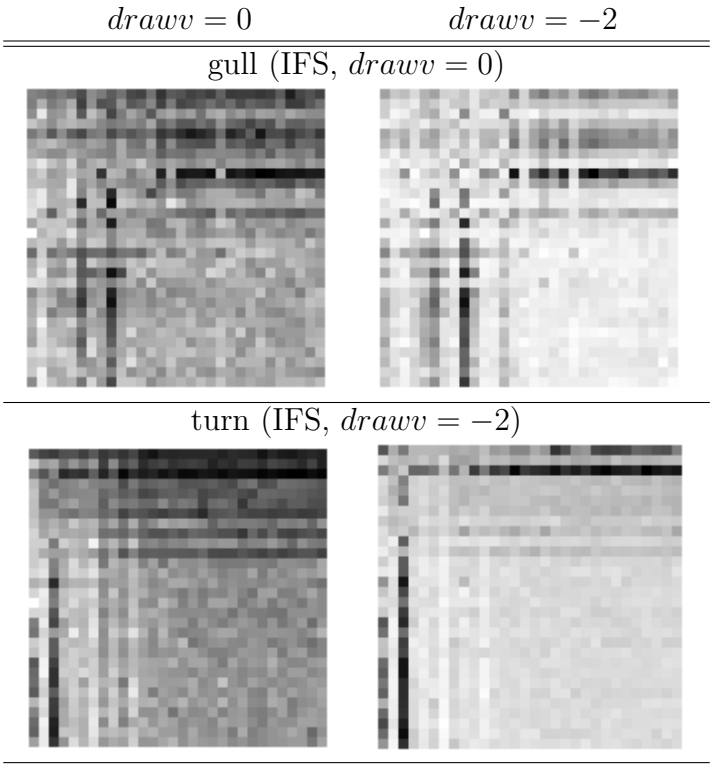


Figure 9.6: Phase 4 IGC diagrams. Diagrams created with $gpr = 100$

9.6 Phase 4

The limited success of the increasing *gpr* strategy suggested another possibility: increase instead the population size. IFS was used for both runs in this phase, because IFS performed best in Phase 3. To keep run times under a week, the increase in *pop* has a corresponding decrease in *gpr*.

The increased population size seems to have increased the distinct improvement period from 3 generations to about 15. The decreased smoothness of the diagrams may be attributed to the reduced accuracy of the ratings – the best strategies are unlikely to be lost in this case though, because of the large population size.

As for Phase 3, there is a figure showing the progression of fv (Fig 9.6).

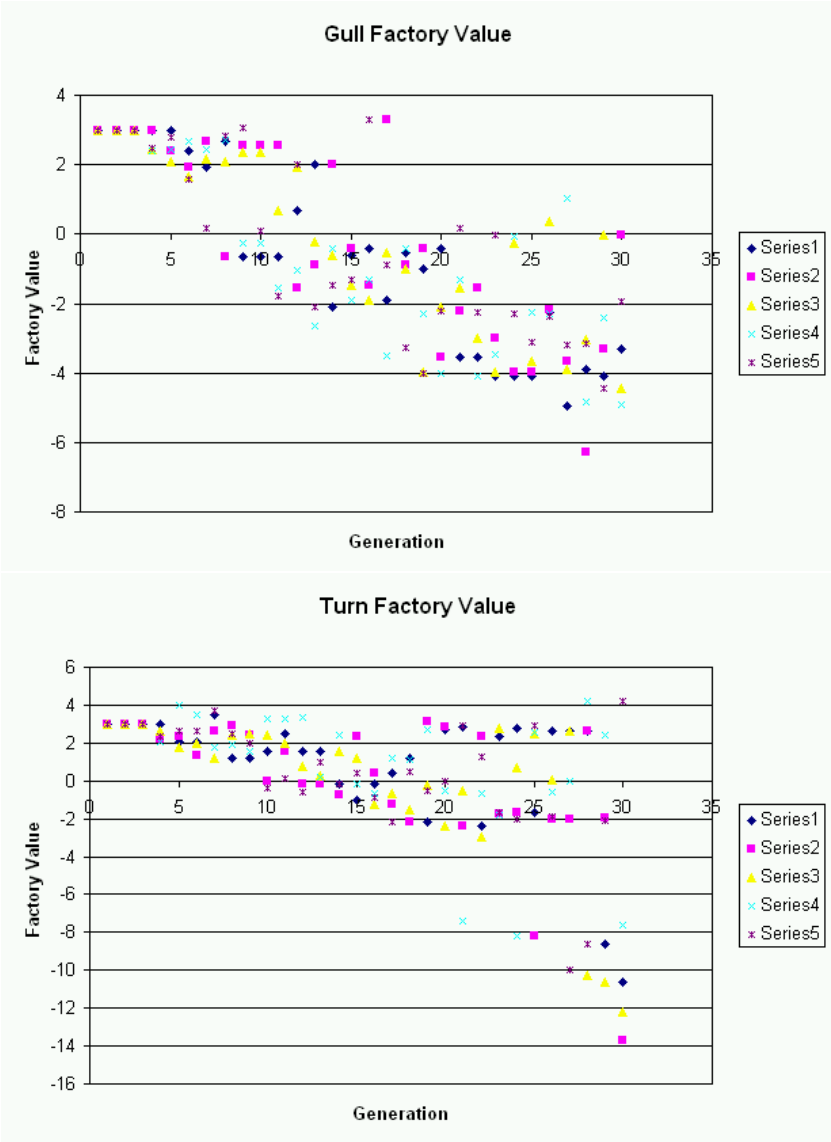


Figure 9.7: The Factory values for the five oldest strategies in each generation, phase 4

Chapter 10

Discussion

10.1 Introduction

This chapter discusses the difficulties encountered and the successes.

10.2 Problems

The basic problem is that designing neural networks is still an art more than a science. When this is combined with the high CPU cost of testing a neural network structure, either a lot of time, or a lot of processing power and a moderate amount of time is required in order to do more research in the area.

This also applies in part to using NN based CPs in production games. Given that using training or evolving a neural net is going to take some time after a game is otherwise finished, using a NN in a CP is which puts the release date back, which is a bad thing. Consequently, neural nets are going to be difficult to use in CPs in the first release of a game. However, if a game's computer player modules are implemented using conventional methods but designed with using a neural net in mind it may be possible to include NN CPs in an add-on pack. Alternately, a new and innovative CP might be a sufficient selling point to delay the game while the CP is brought up to par.

10.3 Successes

The neural networks based CPs did improve for several generations. The networks managed to do something that was not hard coded into them - they consistently worked out that factories should be weighed as less than

tanks. This changed the features provided to the network from strength in an area, to how well the factories in an area were defended.

They also managed to generate their own tactics. For example, on several occasions a strategy was seen to use a classic taunting tactic to fragment a large enemy force. Essentially this involves moving a few units close to the enemy force and fleeing with only a few of the enemy following

Chapter 11

Conclusions

This thesis has presented an environment for studying RTS computer players, a package for using neural networks and a framework for coevoltion. It has then gone on to use all these packages to create a computer player that uses a neural network. The computer player was then evolved to player better than a player with random weights and biases.

Evolution can be used to optimise values used in CPs. Especially when the values are the weights and biases of a neural network. However, to do this in a production environment will probably take longer than management is prepared to wait. NN CPs might be possible in addon packs.

Part IV

Appendices

Appendix A

RTS terms

- Real Time Strategy games (RTS) - third person games where the player controls troops and, often, needs to collect resources.
- units - the entities that a player gives orders too. Tanks, for example.
- base - a group of buildings
- main base - a player's main base is usually the player's largest group of buildings. It is usually built around the player's starting point.
- expansion - a base other than the main base. Usually created for convenience to a group of resources
- ranged - ranged attacks are attacks over distance, as with a bow and arrow, for example. Ranged units are units with ranged attacks.
- melee - melee attacks are non-ranged attacks (for example, swords). Melee units are units with only melee attacks.
- health - How many HP a unit has. A tank that can sustain a further two HP damage before being destroyed is said to 'have 2 heath' or 'be on 2 heath'
- HP or Hit Points - the unit of damage. For example, a Tank does one HP damage, and a Factory starts with 3 HP. So it takes 3 shots from a Tank to destroy a Factory.
- tick - the smallest unit of time that an RTS recognises
- victory conditions - a predicate that needs to be satisifed for one side to win. For example, team A may win if team B has no units remaining.

- add-on pack - Software released after the game itself that makes some modification the game, by adding additional unit types for example.

Appendix B

Java terminology and conventions

For those unfamiliar with Java and object oriented programming, this appendix covers some of the terms used. It also notes the naming conventions used, which allow methods, packages and classes to be distinguished based on the capitalisation of their names. These naming conventions are from Sun Microsystem's reccomended sytle [1].

method the smallest named unit of code. The term encompasses both functions and procedures. Method names are written with all but the first word capitalised, for example `getFactoryValue()`

class A collection of methods and variables that can be instantiated to form objects of the class. Class have all the words capitalised, `IndividualTracker` for example. A class may also be distinguished from other classes with the same name in different packages by prepending the name of the package it is in. Using the long form, the example would be written: `fourgh.evolve.coevolve.IndividualTracker`

package A collection of related classes. A class may only be in one package. Package names are written as lowercase strings separated by dots, `fourgh.surts`, for example.

Bibliography

[1] Code conventions for the java programming language. Web page, viewed 2001-10-21, http://java.sun.com/docs/codeconv/, 1999.

[2] Kumar Chellapilla and David B. Fogel. Evolution, neural networks, games, and intelligence., 1999.

[3] Kumar Chellapilla and David B. Fogel. Evolving an expert checkers playing program without using human expertise. *IEEE Transactions on Evolutionary Computation*, 5(4), August 2001.

[4] Dave Cliff and Geoffrey F. Miller. Tracking the red queen: Measurements of adaptive progress in co-evolutionary simulations. In *European Conference on Artificial Life*, pages 200–218, 1995.

[5] Paul J. Darwen and Xin Yao. Speciation as automatic categorical modularization. *IEEE Trans. on Evolutionary Computation*, 1(2):100–108, 1997.

[6] Andreas Zell Igor Fischer, Fabian Hennecke. JavaNNS: Java neural network simulator, user manual. Web page, http://www-ra.informatik.uni-tuebingen.de/forschung/JavaNNS/manual/ viewed 2001-09-18, 2001.

[7] D. Lin. *The Adaptive Time-Delay Neural Network: Characterization and Applications to Pattern Recognition, Prediction and Signal Processing.* PhD thesis, The University of Maryland, 1994.

[8] Jordan B. Pollack, Alan D. Blair, and Mark Land. Coevolution of a backgammon player. In C. G. Langton, editor, *Proceedings of Artificial Life V*, Cambridge, MA, 1996. MIT Press.

[9] Nicol N. Schraudolph, Peter Dayan, and Terrence J. Sejnowski. Temporal difference learning of position evaluation in the game of go. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 817–824. Morgan Kaufmann Publishers, Inc., 1994.

[10] Timothy Adam Smith. Classtorsf: Rigi standard format from Java classes. 2001.

[11] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, pages 58–67, March 1995.

[12] Sebastian Thrun. Learning to play the game of chess. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 1069–1076, Cambridge, MA, 1995. The MIT Press.

[13] Lex Weaver and Terry Bossomaier. Evolution of neural networks to play the game of dots-and-boxes. In *Artificial Life V: Poster Presentations*, pages 43–50, 1996.

[14] Kenny Wong. Rigi user's manual version 5.4.4. Web page, http://ftp.rigi.csc.uvic.ca/pub/rigi/doc/rigi-5.4.4-manual.pdf, viewed 2001-10-10, 1998.