41 0628836 2

426321

PHD/2006 CHU

ProQuest Number: 10183426

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.



ProQuest 10183426

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

AN INVESTIGATION INTO HYBRID ARTIFICIAL INTELLIGENCE TECHNIQUES APPLIED TO THE GAME OF GO

Julian James Churchill

A thesis submitted in partial fulfilment of the
requirements of The Nottingham Trent University
for the degree of Doctor of Philosophy

November 2006

Abstract

The use of Artificial Intelligence (AI) techniques has allowed many advances in computing and increasingly permeates modern computing applications within industry, commerce and entertainment. This thesis describes an investigation into hybrid AI techniques which focuses on applying these techniques to the game of Go, a two-player game of zero chance, and looks into how to combine AI techniques in a complimentary and effective manner. The hybrid approach was driven by the complexity of Go, which whilst being simple to learn and having very few rules, remains an incredibly difficult game to master. This applies even more so for a computer since the brute force approach which worked so well for computer Chess fails to produce even moderate Go programs. Therefore a further level of intelligence is required; the program must be smarter, more efficient and contain within it many abstract concepts that human beings so readily absorb.

The thesis is roughly split into two investigations involving the fields of Artificial Neural Networks (ANNs) and Genetic Algorithms (GAs). The former has shown some promise in the field of computer Go and naturally lends itself to be included as a component in a larger AI system. Different training methods and algorithms are compared with reference to Go and several network architectures of varying levels of complexity are tested. The ANNs developed are also used within a Minimax framework to show the improvement over a state of the art game tree search algorithm.

Later chapters discuss a GA-oriented approach to Go which resulted from attempts to create a flexible humanistic playing algorithm based on game tree search. This was initiated after the limitations of a standard brute force approach, even with the addition of ANN encapsulated knowledge, were fully realised. Additionally the ANNs developed earlier in the thesis are used to incorporate expert knowledge within the GA.

Notice Of Copyright

The content of this thesis is the author's own work and is, as far as he is aware, original.

No part of this thesis may be reproduced without proper acknowledgement to the author.

Unless indicated otherwise, all illustrations were generated by software developed by the author.

Acknowledgements

The author has been very lucky over the period of his life that this thesis encompasses to have experienced the wonderful support of family, friends and colleagues without which this thesis may never have been completed.

Firstly the author would like to express his sincere gratitude to Dr. Richard Cant for his consistent support, guidance and encouragement. Dr. Cant's supervision of the project has allowed it develop and finally bear fruit. The help and support of Professor David Al-Dabass as second supervisor is also gratefully acknowledged, especially when it came to the writing and submission of conference papers and preparing the related presentations.

The mental and financial support provided by the author's family over these years has been greatly appreciated and all his thanks and love go to Suzanne, his mother, Frank, his father and Charlotte, his sister for their kindness and generosity.

The author would also like to thank his girlfriend Jennie for her love, support and constant patience and his friends Daren, Rosie, Harry, Jan, Charlie, Chris, Jim, Kieran, Marnie and others past and present for lending their ears and occasionally their minds to the pursuit of an intelligent Go playing computer program.

Finally the author wishes to express his thanks to the Nottingham Trent University for providing the opportunity and funding to allow him to complete this thesis.

Table of Contents

Abstract i
Notice of Copyright ii
Acknowledgements iii
Table of Contents iv
List of Figures viii
List of Tables x

1 Introduction

1.1 Overview 1
1.2 Aims 1
1.3 Previous and Related Work 2
1.4 An Introduction to the Game of Go 2
1.4.1 Terms 3
1.4.2 Capturing Stones 4
1.4.3 The Aim of the Game 4
1.4.4 Scoring 5
1.4.5 Rank System 6
1.5 Why Go? 6
1.6 An Overview of the Field of Computer Go 8
1.6.1 Games Research 8
1.6.2 Development of Computer Go 9
1.6.2.1 Evaluation Functions 10
1.6.2.2 Artificial Intelligence in Go 11
1.6.2.3 Resources 13
1.6.3 Recent Programs and Research 13
1.6.3.1 Commercial Programs 13
1.6.3.2 Academic Programs 14
1.7 Summary 15

2 Go Programming and Machine Learning

2.1 Introduction 17
2.2 Use of Machine Learning in Games 17
2.3 Use of Artificial Neural Networks in Go Programming 18
2.3.1 TD and NeuroGo 18
2.3.2 Golem and WinHonte 19
2.4 Use of Evolutionary Computation in Go Programming 20
2.4.1 Breeding Neural Networks 21
2.5 Summary 22

3 An Overview of Game Tree Search Techniques

3.1 Introduction 23
3.1.1 Depth-First or Breadth-First? 23

3.1.2 Minimax 24
3.1.3 Alpha-Beta Pruning 26
3.1.4 Search Windows 28
3.2 Minimax Enhancements 30
3.2.1 Iterative Deepening 30
3.2.2 Best Move First and Enhanced Transposition Cutoffs 31
3.2.3 Move Ordering 31
3.2.4 Local Search Area 32
3.2.5 Forward Pruning 32
3.3 Evaluation Functions 33
3.4 Alternatives to Minimax 33
3.5 Summary 35

4 An Introduction to Artificial Neural Network Theory

4.1 Introduction 37
4.2 Architecture of an Artificial Neural Network 38
4.2.1 The Bias Term 38
4.2.2 Activation Function 39
4.3 Learning With Artificial Neural Networks 40
4.3.1 Supervised Learning 41
4.3.1.1 Backpropagation 41
4.3.2 Reinforcement Learning 42
4.4 Summary 42

5 Applying Artificial Neural Network Techniques to Go

5.1 Introduction 44
5.2 The Artificial Neural Network Ideas 44
5.2.1 A Move Scoring Network 45
5.2.2 Symmetry and Anchoring the Network 46
5.2.3 Input Field 48
5.2.4 Input Features and Schemas 49
5.2.5 Game Stage 51
5.3 Alternative Uses 51
5.3.1 Area Finder Network 52
5.3.2 TD Evaluation Function 52
5.3.3 Safety Estimating Networks 53
5.3.4 Urgency Networks 55
5.3.5 Competitive Networks 57
5.4 Summary 57

6 Neural Network Results

6.1 Introduction 59
6.2 Training the ANN 59
6.2.1 Non-Repeating Training Data 60
6.2.2 Comparing the Designs 62
6.2.2.1 Encoding the Edges 63
6.2.2.2 Additional Features 64

6.2.3 Varying the Training Parameters 65
6.3 Using Machine Learning to Solve Go Problems 69
6.3.1 Testing in Real Play 72
6.4 Summary 73

7 An Introduction to Genetic Algorithms

7.1 Introduction 75
7.2 The Algorithm 75
7.2.1 Fitness Function 76
7.2.2 Selection Process 76
7.2.3 Crossover Operator 77
7.2.4 Mutation Operator 78
7.3 Why Use a GA? 78
7.4 Practical Uses for GAs 78
7.4.1 Artificial Neural Networks 79
7.4.2 Intelligent Behaviour in Games 79
7.4.3 Path Finding 79
7.5 Genetic Programming 80
7.6 Summary 80

8 Development of the Hybrid Genetic Algorithm

8.1 Introduction 82
8.2 Motivations 82
8.3 Overview 83
8.3.1 The Evolutionary Core 84
8.4 The Hybrid Genetic Algorithm 85
8.4.1 Attack Phase 85
8.4.1.1 Finding an Attack Position 86
8.4.1.2 Finding an Attack Move 87
8.4.1.3 Refuting the Attack 88
8.4.1.4 Chromosome Size Control 91
8.4.2 Repair Phase 91
8.4.2.1 Repair By Mutation 93
8.4.2.2 Repair By Crossover 96
8.4.3 Reproduction Phase 97
8.4.4 Population Culling 99
8.4.5 Chromosome Trimming 99
8.4.6 Search Termination Conditions 100
8.5 An Example Run 100
8.5.1 The Attack Phase 101
8.5.2 The Repair Phase 103
8.6 Summary 104

9 Hybrid Genetic Algorithm Results

9.1 Introduction 106
9.2 Testing the Hybrid Algorithm 106
9.2.1 Assessing the Genetic Algorithm Configuration 107

9.2.1.1 Mutation and Crossover 109
9.2.1.2 Reproduction and Repair 110
9.2.1.3 Configuration Conclusions 111
9.2.2 Comparing Confidence Criteria 112
9.2.3 Comparing To A State Of The Art Minimax Implementation. 114
9.2.3.1 Making The Test Fair 115
9.3 Conclusions 117
9.4 Summary 118

10 Conclusions

10.1 Introduction 119
10.2 Contribution to Knowledge 119
10.2.1 Applying ANN Techniques to Go 120
10.2.2 Developing a Hybrid Genetic Alternative to Minimax 122
10.2.3 Using the Techniques Outside the Problem Domain 123
10.3 Future Work 124
10.3.1 GA Meta-Goals 124
10.3.2 Measuring the Robustness of Chromosomes in Generic GAs 125
10.3.3 Performance Extensions 125
10.3.3.1 Refining the ANN Design 126

11 References 128

12 Appendix A – Search Test Diagrams 137

13 Appendix B – Software Design – Modules and Classes 141

14 Appendix C – Software Design - Diagrams 154

15 Appendix D – Genetic Algorithm 158

16 Appendix E – Neural Network Results 163

17 Appendix F – Genetic Algorithm Results 166

18 Appendix G – Publications 173

List of Figures

1.1 Liberties 3
1.2 Rule Examples for Go 5
1.3 Prisoner's Dilemma – Outcomes for Prisoner A 9

3.1 A Minimax Tree 25
3.2 Pseudo Code – Minimax Function 26
3.3 An Alpha-Beta Pruned Minimax Tree 27
3.4 Pseudo Code – Alpha-Beta Function 28
3.5 Pseudo Code – MTD(f) Function 29
3.6 Pseudo Code – Iterative Deepening 30

5.1 Input Scan Field 47

6.1 Comparing Training Set Size 61
6.2 Comparing the Final Performance of the Input Schemes 62
6.3 Shape Areas 65
6.4 Problem Success Versus Time Limit Per Problem 70

7.1 One Point Crossover 77

8.1 Overview of the GA Algorithm 85
8.2 Attack Phase Overview 86
8.3 Attack Move Selection 88
8.4 Integrating the Attack Move 89
8.5 Defender Move Process 90
8.6 Attacker Move Process 91
8.7 Repair Phase Overview 92
8.8 Mutation Based Repair 93
8.9 Find Repair Move 93
8.10 Mutation Based Repair Integration 94
8.11 Defender Move Repair Process 95
8.12 Attacker Move Repair Process 96
8.13 Reproduction Phase Overview 97
8.14 An Example Chromosome DAG 99
8.15 One Stone Capture / Capture Sequence 101
8.16 A Random Chromosome 101
8.17 The First Attack 102
8.18 An Extended Liberty Capture Sequence 103
8.19 A Mutation Based Repair 104

9.1 Comparing GA Configurations 109
9.2 Comparing Mutation and Crossover 110
9.3 Comparing Reproduction and Repair 111
9.4 Comparing Confidence Criteria 113

9.5 Comparison of Confidence Levels of Morgoth Configurations 114

C.1 Morgoth Package 154
C.2 Genetic Algorithm Package 155
C.3 Goal Package 156
C.4 SGF Package 157
C.5 Util Package 157

D.1 Example Chromosome 162

E.1 Input Field Size 163
E.2 Learning Rate 163
E.3 Momentum 164
E.4 Hidden Neurons 164
E.5 Rank Range for Training Data 164
E.6 Number of Hidden Layers 164
E.7 Input Field Shape 165
E.8 Training Data Source 165

List of Tables

5.1 Input Scheme Details 49

6.1 Final Global Rank of Professionals Moves 63
6.2 Optimum ANN Parameters 67
6.3 Varying the Training and Architecture Parameters 68
6.4 Enhancing MTD(f) with an ANN (400s per problem) 71
6.5 Performance Against GNUGo 3.0.0 73

9.1 GA Configurations and Success Rates 108
9.2 Comparing the GA and MTD(f) Search Algorithms (120s) 116

F.1 'All' Configuration 168
F.2 'Without Reproduction or Repair' Configuration 170
F.3 With 'Reproduction By Crossover and Repair By Mutation' 172

Introduction

1.1 Overview

Machine learning techniques pose an appealing contrast to other branches of Artificial intelligence (AI); researchers hope they might discover something of the elusive complexity of life, of which the ability to learn is an essential element. The construction of an intelligent system combining a capacity to learn with appropriate deterministic algorithms exploits the advantages of logic-based computers whilst enabling a degree of experience and adaptability to be included in a potentially ever-developing system.

This thesis presents research into the development and evaluation of hybrid AI algorithms that utilise machine learning techniques coupled with classical search algorithms. Additionally, a novel Genetic Algorithm (GA) is developed and compared with the enhanced search algorithm. To conduct the research, the domain of non-cooperative games theory was chosen. This domain includes the class of complete information, zero chance board games such as Chess, Othello and Go. In particular the game of Go was the primary focus of this research.

1.2 Aims

The major aims of this research project are listed here.

1) To enhance the performance of the classic Minimax algorithm for game tree search with flexible, dynamic, domain specific knowledge by introducing a machine learning element using Artificial Neural Network (ANN) techniques.
2) To investigate appropriate problem domain representations and architectures for the creation of ANNs with the aim of improving a Go playing program based on tree search techniques.
3) To develop and implement an evolutionary based algorithm as an alternative to the Minimax method, designed to emulate a human game player as opposed to a complete deterministic search of the problem space.

The problem domain chosen for this research is games theory, in particular two person, zero chance board games, however the methods developed could be used to enhance any application of a Minimax based game tree search algorithm. This thesis demonstrates the potential for coupling machine learning methods with deterministic AI techniques in such a way that the benefits of all the approaches can be exploited. The evolutionary based algorithm described offers an intuitive alternative to approaching search problems by using a closer model of human play than traditional methods allow.

1.3 Previous and Related Work

The research presented in this thesis continues, in part, from work previously reported in the authors MSc dissertation [Churchill 2001]. This included very early efforts in designing and implementing an ANN to assign a quality value to proposed moves and a game tree search framework within which to use the ANN. The MSc dissertation explored the possibility of combining a naively developed machine learning module with the Minimax algorithm and presented some promising results and possibilities for future research that inspired the research presented here. Related publications by the author are included in Appendix H.

1.4 An Introduction to the Game of Go

The game of Go is an ancient oriental board game in the same class of games as Chess, Othello and Connect-4 where no random element is involved in play, such as a die or the shuffling of cards. These types of games are often termed zero chance. They also belong to the class of games defined in Games Theory as zero-sum because both players are competing for a fixed-size amount of reward and the sum of both players final reward is zero [von Neumann and Morgenstern 1944]. For an excellent introduction to the rules of the game see [Chikun 1997]. For the readers benefit a brief summary of the rules and examples of play are given below.

Go is usually played on a 19 by 19 grid of intersections, all unoccupied at the start of play. It is also common to see practice games played on smaller boards, typically 9x9 or 13x13. It is a two player game, one playing with black stones, the other with white stones. The players

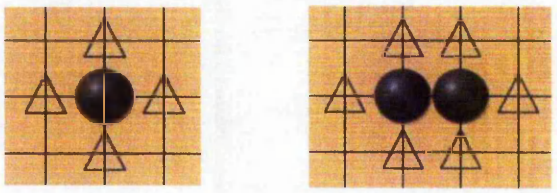
take turns to place one stone of their own colour on an empty intersection. Corner and edge points are valid places to play a stone, giving a total of 361 initial choices of move on an empty 19x19 board.

1.4.1 Terms

Below are some definitions for Go related terms that will be used throughout the thesis:

- *Connected* – Stones of the same colour are said to be connected if they are orthogonally adjacent to each other.
- *Liberty* – Empty intersections adjacent to stones are referred to as the liberties of those stones.
- *String* – Connected stones of a single colour are called strings. In computer Go literature the terms dragon and worm are also used. The set of liberties of a string is equal to the set of liberties of the component stones excluding duplicates. Due to this all stones in a string live or die together.
- *Group* – A set of same colour strings that will usually share a common fate. The strings in a group are often in close proximity and the potential for these strings to connect will be high.
- *Eye* – An empty point or set of connected empty points (an eye space) that is contained within a group. Various types of eye exist including real eyes, half-eyes and false-eyes.

Figure 1.1 demonstrates the concept of liberties, for a single stone (a) and for a multi-stone string (b).



(a) (b)

Figure 1.1 - Liberties

1.4.2 Capturing Stones

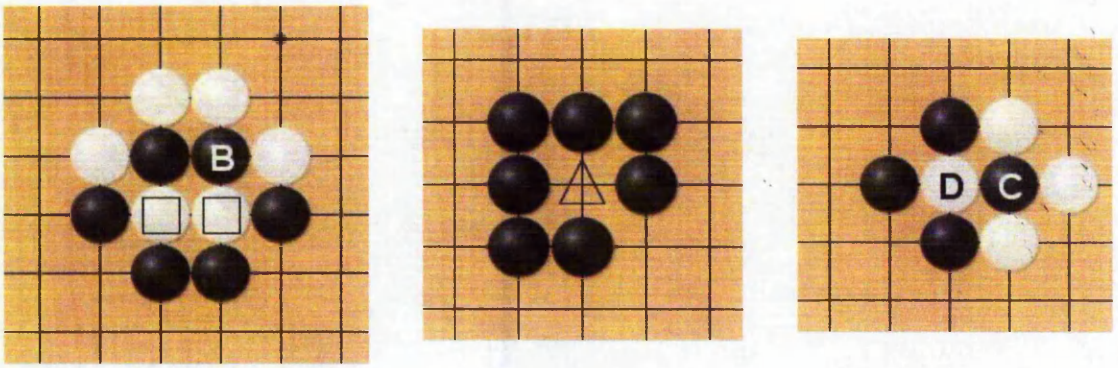
Once a stone is placed on the board it may only be removed if it is captured, which is achieved by occupying all the liberties of the stone or string with enemy stones. There are only two restrictions on the placement of stones; suicide moves are illegal unless they result in the immediate capture of enemy stones and a player may not play so that the board state is the same as it was at the end of their last turn. The second proviso is known as the Ko rule. This rule is in place to prevent a stalemate of repeated capture-recapture moves. For computer tournaments a Super-Ko rule is often used prohibiting the reoccurrence of any board state created during the game, to avoid n-step Kos which can result in an infinitely looping game.

1.4.3 The Aim of the Game

The aim of Go is to surround as much territory as possible whilst minimising your opponent's territory. Territory can be loosely considered empty areas surrounded by stones of a single colour. By the end of the game the board will have reached an equilibrium where further play by either side will not change the status of the stones on the board. Due to the illegality of suicide and being able to place only a single stone at a time, those groups of stones that can eventually form two exclusive inner areas of eye space will be uncapturable and are termed alive. These areas can never be reduced to less than a single eye each by the opponent, requiring a suicide move in two places in a single go – an impossibility.

To illustrate these rules please consider Figure 1.2 which shows various examples of play to illustrate the rules of Go. In Figure 1.2 (a) a capture situation is shown where the last liberty of the marked white stones has been removed by the black play at 'B'. The white stones would now be removed from the board and count towards the black players score at the end of the game. A basic single point eye formation is shown in Figure 1.2 (b). A Ko situation is depicted in Figure 1.2 (c), where move 'C' has just been played, capturing the white stone 'D'. The Ko rule prohibits white from immediately playing at 'D' to capture the black stone 'C' resulting in a repeated board state, however white may play at 'D' later in the game when the board state has changed.

The end of the game is signalled by consecutive passes from each player after a situation has developed within which neither player can make further progress. The board is then scored to determine the winner.



(a) Capture (b) An Eye (c) Ko

Figure 1.2 - Rule Examples for Go

1.4.4 Scoring

The most common approaches to scoring a completed board position use either area scoring or territory scoring. The two systems provide a definition of the score rather than a method to find it and they nearly always result in the same final score value. The term counting is more readily used to describe a procedure that players can use to find the score. The Chinese rule set uses area scoring. This is calculated by totalling the number of points containing a stone of your colour plus the number of empty points surrounded by your colour including those occupied by dead opponent stones. The Japanese rule set uses territory scoring, which is the number of empty points surrounded by stones of your colour, including those occupied by dead opponent stones, minus the number of your stones that have been captured or are dead on the board. This has the affect that moves placed within your own territory will unnecessarily reduce your score by one point per stone for the Japanese rules, but not for the Chinese. For Computer Go Chinese counting tends to be adopted since it is simpler to compute.

An additional number of compensation points are given to the player that takes the white stones due to the assumed advantage the black player has of placing the first stone. This is termed the Komi and is commonly set at 5.5 for board sizes of 9x9, 13x13 and 19x9 but may

differ from country to country and tournament to tournament. The inclusion of half a point in the Komi value ensures that the game will never end in a draw.

There are several different rule sets with differing scoring nuances, Komi values, Ko rules and special positional exceptions such as 'bent four in the corner'. For a comparison of some of the different international and tournament rule sets see [BGA 2004].

1.4.5 Rank system

The ranking system in Go is divided into two classes of grades, Kyu and Dan, where 30 Kyu is a typical absolute beginners grade. From 30 Kyu the grade number reduces as your skill increases, a good amateur being between 10 and 1 Kyu. After 1 Kyu you count Dan grades from 1 up to 7 for amateurs. Professional grades go from 1 Dan to 9 Dan by smaller increments, 1 Dan professional being roughly equivalent to 7 Dan amateur. The difference in rank between two players is used to calculate a handicap for the weaker player implemented either as a number of freely placed stones or placed according to a standard ordering, depending on the rule set. The weaker player always takes the black stones and if the difference is only 1 grade no handicap stones are given. After that a number of stones are given equal to the difference in rank. For example a 5 Kyu player would give a 10 Kyu player 4 stones. The player taking the handicap places the handicap stones as their first move. The star points on the board are used to place the stones for a fixed handicap placement. On a 19x19 board there are nine star points one in the centre, one in each corner four points from the edges and four in-between the corner star points four points from the top, bottom, left and right edge.

1.5 Why Go?

The problem domain provides researchers with a rich environment within which many aspects of Artificial Intelligence can be investigated. Since the early days of AI research Chess has been at the forefront of efforts to understand and emulate the human mind. The game was considered by many to epitomise the complexity and power of human thought and was considered to be an excellent test bed for the development of this fledgling field. Chess, as with Go, is a game of zero-chance. It can be deterministically solved given enough

computational resources to construct a complete game tree. As new techniques were developed and the power of computers grew the possibility of using a brute force method to solve huge game trees became more practical until a point where dedicated Chess computers were built that could beat the best players mankind had to offer [IBM 1997]. A challenge had to be found that would push research to its limits. Go is increasingly recognised as the domain successor to Chess and research efforts have steadily increased over the years.

To put the choice of Chess or Go into context compare the complexity of each game. The length of an average game of chess is around 40 moves with on average 10 legal moves available to each player per turn. This gives an approximate search space of around 10^{40} nodes. In comparison a game of Go can average 200 moves, with around 180 legal moves available to each player at each turn. The size of this search space has been estimated at around 10^{170} nodes [Allis et al. 1991]. The successful application of brute force techniques such as game tree search to Chess can clearly not be applied so readily to Go, even considering the technological advances that have allowed such impressive progress with Chess, with the very best programs run on purpose built, dedicated hardware. Simply throwing evermore powerful hardware at the problem will not solve it in the near future so we must look to more intelligent techniques such as those the field of AI is dedicated to developing.

Furthermore the emergent complexity evident in Go produced from a simple and small collection of rules gives rise to abstract, high-level properties that are very difficult to model on a computer. To give a sense of the enormous difficulty and potential benefit of developing AI techniques for Go programs this chapter ends with a quote from Cho Chikun, a highly respected 9 Dan professional Go player [Chikun 1997].

> *"Because of Go's complexity, it could provide the vehicle to help make some important advances in artificial intelligence. Most applied-AI tasks, such as vehicle manoeuvring or mechanical manipulation, take place in the real world. But this is a noisy domain which makes problems difficult to solve. Go, on the other hand, provides a clean environment, yet has many features in common with the targets of applied AI."*

1.6 An Overview of the Field of Computer Go

1.6.1 Games Research

The term 'Games Research' can be applied to a broad spectrum of disciplines including mathematics, economics and computer science. In recent times it has been more frequently associated with digital video games, however the fields of adversarial and non-adversarial games theory have made a large contribution to improving our understanding of many complex real world problems such as global economics, stock market prediction and the logistics of warfare. Game theory can be divided into two areas; cooperative and non-cooperative games theory. It was developed by John von Neumann, including its application to the important case of two player non-cooperative games. Additionally he proved the fundamental Minimax Theorem, the basis for the most common tree search algorithm used in game playing programs. With Oskar Morgenstern, Von Neumann published essential work on the formal conception of game theory including its relation to economic theory [von Neumann and Morgenstern 1944]. After this seminal publication it became apparent that the theories presented within could be applied to many areas such as warfare, politics, sociology and psychology.

Although the focus of this thesis is on non-cooperative games it may be useful to define the distinction between cooperative and non-cooperative games. Cooperative games theory concerns games where the players have the option of cooperating, typified by the famous Prisoners Dilemma problem [Tucker 1950]. The name of the problem derives from the situation that two hypothetical prisoners find themselves in after they have committed a crime together. Both are in custody, separated from each other so that they cannot communicate, and the police give them the following option; betray your partner in crime and walk free. They can either both keep silent and suffer a small sentence, one can betray the other to gain a large reward and the other will suffer a large sentence, or both can choose betrayal and both will suffer more than if they had both kept silent. The outcome matrix in Figure 1.3 shows the different resolutions for Prisoner A. The dilemma in the problem is that the prisoners cannot make the best choice without knowing what the other prisoner is going to do.

Prisoner A\Prisoner B	**Keep Silent**	**Sing Like A Canary**
Keep Silent	Small Sentence (1 month)	Large Sentence (4 years)
Sing Like A Canary	Walk Free	Medium Sentence (2 Years)

Figure 1.3 - Prisoner's Dilemma - Outcomes for Prisoner A

This thesis will focus on zero-sum games, which are defined as games where a fixed size quantity of reward is shared between the two players, so that the amount one player is winning by is equal to the amount by which the other player is losing; an equilibrium is maintained. Non-zero-sum games have a pool of resources of no fixed size that can be won by the players during the game. The Prisoner's Dilemma is an example of a cooperative non-zero-sum game.

Amongst the most popular non-cooperative games domains within which research has been based are Chess, Go, Backgammon and Poker. Chess and Go are both games of zero-chance; they involve no dice throws, wheel spins or random choices. Backgammon and Poker represent the class of games that do involve a random element. In the case of these games luck has a varying degree of influence over the outcome of the game, however they are still very much worth investigating as they are perhaps closer to real world problems, where the skilful problem solver must adapt to uncontrollable factors.

1.6.2 Development of Computer Go

Several papers offer reviews of the state of Computer Go covering different periods of the fields development. Early Go programs developed by Zobrist and others are discussed by Wilcox [Wilcox 1988]. An overview of the early nineties is presented in [Erbach 1992] and a millennial review of the state of the art, programming techniques and use of domain specific knowledge can be found in [Müller 2002]. Müller also discusses two informative case study games, concerning a computer versus a computer and a human versus a computer, to illustrate the difficulties and the achievements of Computer Go and the challenges for the future. In "Computer Go: An AI Oriented Survey" an in depth analysis is given of many of the techniques used to tackle Go over the years and displays the variety of domain driven methods that have been developed and applied to Go and other games [Bouzy et al. 2001].

1.6.2.1 Evaluation Functions

The earliest computer programs to play Go focussed on developing an effective and efficient evaluation function. Given a sufficiently well performing evaluation function with which to assess the board state a program should need little else to allow it to play to a very high standard. Unfortunately this function has so far eluded researchers and Go players alike, some reasons for this are the incredibly large search space within which a game of Go is played and the complex interaction of the stones, making simple determinations of life and death into huge search problems themselves.

Zobrist created one of the very first Go playing programs, using an estimate of stone influence radiating across the board to produce a field of influence to use as an evaluation function to guide move choices [Zobrist 1969]. Bouzy later went on to develop this idea further into the 5/21 algorithm [Bouzy 2003a] used by GNUGo [GNUGo] amongst other programs.

The algorithm itself assigns large colour symmetric values to the points beneath living stones, for example +128 for black stones and –128 for white stones, and uses repeated dilations and erosions of the board position to propagate these values across the board. For well developed, settled positions this produces a good approximation of the influence of the stones and in turn may be used to estimate territory. Unfortunately this process relies on accurate knowledge of the liveness of the stones present on the board, since only living stones are used as an influence source. This complicates the process considerably as a reliable method of determining life and death must be used before applying the influence function, possibly resulting in many complex tree searches to define the status of strings. Many variations on the influence function theme have been invented over the years, such as introducing extra influence sources according to potential connections, reflecting influence on the board edges and absorbing or blocking influence radiated over opposing source points.

A much more recent attempt to create an evaluation function based program came from the GNUGo team [GNUGo]. Along with the program's aforementioned use of Bouzy's 5/21 influence algorithm, further developed to include damping affects based on permeability of board intersections amongst others, GNUGo uses a complicated algorithm which incorporates many facets of playing Go. The actual strategy for estimating the score – an

evaluation of the current board state, involves extensive analysis of the relationships between the stones followed by reading (analysing move sequences) to determine their status. This information is pumped into the influence module at stages along the way to be feedback into the status analysis, and eventually used to calculate a stable value for the current board position. Once this is known various move generators are run to create a collection of candidates. These are evaluated primarily on their affect on influence and ultimately territory to choose the most valuable move to play.

The majority of Go programs based on explicitly programmed evaluation functions use the influence-as-an-approximation-to-territory paradigm. To this end many programs are essentially grand-scale evaluation functions that gather detailed information from a board state using expert knowledge of the domain and then apply a relatively simple influence based function to turn the status information into a game score.

1.6.2.2 Artificial Intelligence in Go

Artificial Intelligence (AI) efforts both academic and commercial can be loosely placed into three categories. The first considered here is Combinatorial Game Theory (CGT), which formally explores the game through mathematical analysis. In fact Go partly inspired Conway's work on Combinatorial Game Theory [Conway 1976] and can be seen reflected in the cellular automata based game he invented on a Go board – "The Game of Life", which shares with Go the property of emergent complexity. With CGT a board position is broken down into locally exclusive sub-games which are individually solved and combined at a later stage. Work on CGT within Go has resulted in several books, amongst them "Mathematical Go" [Berlekamp and Wolfe 1994] and at least one doctoral thesis [Müller 1995]. However the application of this work has so far been restricted to academic circles as applying the theory beyond well-defined endgames still requires much research. Recently Meijer and Koppelaar [Meijer and Koppelaar 2003] have attempted to expand on the field of CGT applied to Go by including interactions between local positions.

The second category includes all attempts to use traditional non-adaptive techniques such as game tree search. Most traditional Computer Go programs follow the scheme of using a very shallow global search coupled with deep but highly selective, goal directed searches to determine local tactical information such as group safety and connectivity. The tactical

searches are often embedded into the evaluation function, potentially creating complex and expensive recursive searches, hence the need for high selectivity in the move selection process. Large pattern databases are often tailor-made to suggest moves using the local position and optional criteria such as the number of liberties of a target string. These expert systems are often hand tuned by skilled Go players and represent considerable time, dedication and expert knowledge, reflecting the fact that the authors of the most successful commercial programs are themselves high ranked amateur Go players.

The final category, to which the present work belongs, consists of the use of general purpose machine learning techniques. These methods have been used to create both fully fledged Go programs and to tackle specific parts of large, modular Go playing systems. This category includes Artificial Neural Network (ANN) techniques and evolutionary methods such as Genetic Algorithms (GAs). For machine learning by far the most popular approach has been to use ANNs. This has resulted in moderate success for programs such as NeuroGo [Enzenberger 1996] which uses Temporal Difference (TD) learning, a reinforced learning approach that lets the program learn from its own games and facilitates the propagation of the reinforcement signal through time [Sutton 1988]. This allows the network to be rewarded for the foundation moves that lead to a rewardable state in the future as a means of handling the temporal credit assignment problem. Additionally NeuroGo uses a dynamically connected network to better represent the spatial attributes present on a Go board. Previous to this, Nicol Schraudolph and his research team looked into TD learning with a network architecture designed for the problem domain, laying the groundwork for future Go playing TD-based ANNs [Schraudolph et al. 1994].

At least two researchers have attempted to incorporate ANNs based on supervised learning methods into their programs. Golem was one of the first programs to include ANNs, using two networks, one designed for move ordering within a search tree and the other to prune moves at the root [Enderton 1991]. The first network focussed on speed to maintain an efficient search and the second on accuracy, including many local features to aid the evaluation. The program WinHonte also used ANN techniques similar to Golem but complimented this with two TD trained networks, one to estimate territory and the other to estimate the safety of a group of stones [Dahl 1999].

Recently Erik van der Werf has contributed to this body of knowledge through the publication of several papers on the various facets of using ANNs with Go [Van der Werf 2002].

1.6.2.3 Resources

Many active researchers and commercial Go programmers subscribe to the Computer Go mailing list, a highly useful resource for airing ideas, developing protocols and publicising tournament results [Computer Go Mailing List]. From discussing high-level approaches to programming Go, to low-level FPGA implementations, the mailing list remains one of the most useful resources available for Go programmers academic and commercial alike. Another Internet resource, the rec.games.go newsgroup, which can be accessed at <http://groups-beta.google.com/group/rec.games.go?hl=en>, often has researchers actively participating in discussions, although these tend to be less computer-oriented and more concerned with human play.

In addition to the above resources there exists an extensive online bibliography dedicated to publications concerning Go and Computer Go which is frequently updated [Enzenberger 2004]. Also a discontinued newsletter called The Computer Go Bulletin, "devoted to the generation and exchange of ideas about Computer Go" is available on the Internet [Computer Go Bulletin].

1.6.3 Recent Programs and Research

For the purposes of comparing the function of current Go programs it is helpful to define two groups, commercial and academic. Since the motivation for each group differs in a number of fundamental ways they result in different approaches to playing the game of Go.

1.6.3.1 Commercial Programs

Programs such as Many Faces of Go [Fotland 2002] and Michael Reiss' Go++ [Reiss] are amongst the strongest and most popular Go playing programs available to buy. They both use extensive Go specific knowledge in the form of databases containing patterns and associated move sequences, along with hard coded expert rule systems. These components have been

fine tuned over many years by the authors, both strong amateur Go players. It should be noted at this point that even with the advantage of strong human Go players producing these programs, there are currently no programs that come anywhere near a professional level of play. The most recent version of Many Faces of Go, version 11.0, claims its highest playing level to be around 8 Kyu, 8 grades below amateur Dan level and 14 grades below professional Dan level, if you take the 1 Dan professional grade to be roughly equivalent to 7 Dan amateur. When you also take into account the effect of repeated play against a program where the human player will adapt to the programs weaknesses but not vice-versa, the true playing level of these programs turns out to be lower than initial investigations would suggest.

To give a little more detail about the common design of many commercial programs here are a few words about Many Faces of Go [Fotland 1993]. It has been one of the leading programs in the Computer Go arena for many years and uses a combination of techniques such as a rule based expert system and a sophisticated representation of the board position, breaking it down into a composition of various levels of abstract knowledge. Information is updated incrementally to save time over what may potentially be a large amount of repeated calculation, the results of which would be mostly unchanged from move to move. A dedicated Joseki database is included, storing standard corner sequences of play. A database of 8x8 patterns for generating move candidates is also used, along with partial move trees optionally attached to the candidates to save time generating them through a Minimax search process. On top of this, hard coded algorithms to determine the score are linked into the rule based move suggesting system allowing the program to choose more risky moves if it believes it is behind or to adopt a safer, more defensive style of play if it is comfortably ahead. These programs, almost without exception, use the same architecture of shallow global tree search with deep, selective extensions using expert knowledge to select moves from some form of database or rule system.

1.6.3.2 Academic Programs

In contrast academically produced programs frequently have a learning component and tend to use more flexible techniques such as ANNs and Genetic Algorithms. To date only programs that use offline learning have been found, programs that have to be trained before release and frozen to play against an actual opponent. There are currently no programs that

learn online, that adapt to an opponent or learn from experience during a game as far as the author is aware. The strongest academic programs of the moment appear to be Explorer by Martin Müller [Müller 2000], which uses pattern matching in combination with local search and some elements of CGT, NeuroGo [Enzenberger 1996] based on a dynamically connected, TD trained ANN, Golois built upon an Explanation Based Learning system [Cazenave 2004] and Indigo [Bouzy 2003b] a more traditional search and pattern matching based program.

The limited success of using the common approach of applying a basic game tree search algorithm backed up with high performance hardware to the game of Go, has driven research in the development and enhancement of Minimax based algorithms. Research into alternatives to Minimax, including non-deterministic AI methods, has also been given new energy as researchers strive to solve this difficult problem.

As a prime example of how complex, perfect information games have driven research into improving and creating alternatives to Minimax consider Allis' Proof Number search algorithm (PN) [Allis et al 1994], Cazenave's Gradual Abstract Proof Search (GAPS) [Cazenave 2002] and Junghanns' Fuzzy Numbers [Junghanns et al. 1995]. The details of these algorithms are discussed in more detail in Chapter 4 "An Overview of Game Tree Search Techniques".

1.7 Summary

This chapter has presented an overview of the work contained in this thesis including the aims of the research namely to enhance the performance of the Minimax game tree search algorithm with domain knowledge harvested by an ANN, investigate architectures and representations of Go knowledge to train such an ANN and to develop an evolutionary based alternative to Minmax with the purpose of improving the efficiency of search within the very large search space presented by Go.

Previous and related work has been detailed and an introduction to the rules and elementary concepts of Go, along with an explanation as to why Go is an interesting problem domain have been discussed. In section 1.6 an overview of the current state of the field of Computer Go along with a look at the history of Computer Go has been presented including details of

recent programs, commercial and academic as well as a brief introduction to some avenues of research leading from this domain.

Go Programming and Machine Learning

2.1 Introduction

The concept of Machine Learning (ML) refers to a class of algorithms, techniques and paradigms that allow a system to change its behaviour over time as a result of experience, either presented to the program or gathered by it. To actually learn anything the ML algorithm must have a goal or set of goals to direct the adaptation. ML can be sub-divided in a variety of ways, perhaps the most relevant here is to look at it from the point of view of function optimisation. Common optimisation methods include Simulated Annealing - a minimisation technique inspired by the process metal undergoes when cooling to form a minimal energy structure, Artificial Neural Networks (ANNs) modelled on the brain and Evolutionary Computation, an umbrella term to describe methods that have a basis in evolutionary theory [Heitkoetter and Beasley 2001].

For all instances of a learning system, the process can be classed as online, offline or a combination of the two. These terms refer to whether the system learns as it is used *online*, adapting from decision to decision, or if the system requires a dedicated *offline* learning period and the ML component is frozen before use.

2.2 Use of Machine Learning in Games

Machine learning has been used to improve games-playing computer programs for over 20 years, although until recently this has tended to focus on board games such as Chess and Backgammon. For an example of an early program that learnt to play Tic-Tac-Toe on an Atari see [Hafner 1984]. With the prevalence of home computers and games consoles ML has increasingly infiltrated the commercial games market typified by 1997's Creatures, in which you nurture, feed and protect unique evolvable life forms that make use of ANNs and GAs to develop. An extensive bibliography for ML in strategic game playing can be found at

[Fürnkranz 2003] and a more general view of ML in games is presented in both [Scott 2002] and [Woodcock 2004]. For a survey of ML specific to Go see [Stoutamire 1991].

An excellent proof of application for reinforcement learning in games came when Tesauro released his ANN based Backgammon program, called TD-Gammon [Tesauro 1994]. It was the first expert level Backgammon program to have been produced and learnt by self-play using the Temporal Difference (TD) algorithm to train its evaluation function. It showed that with only a small amount of knowledge and an appropriate representation and choice of learning method, a program could be constructed to challenge the very best human players in the world. Others have attempted to transfer this success with TD to other domains, for instance NeuroGo [Enzenberger 1996], but without achieving quite the same level of success,

2.3 Use of Artificial Neural Networks in Go Programming

Over recent years the use of ANNs in Go programs has increased markedly, as has the number of academic papers published on Go as a whole. Using ANNs has become the most common example of ML in Computer Go. Of the programs that use ANNs, only one has been released commercially, WinHonte 1.03 in January 2005 [Dahl 2005] and nearly all the other programs have been produced as the result of academic research. The way that ANNs have been used in these programs is fairly uniform and can be categorised into board evaluation functions and move evaluating networks, sometimes both being combined in a single program.

2.3.1 TD and NeuroGo

From an academic point of view NeuroGo is arguably the most impressive Go playing computer program, using reinforcement learning via the TD class of learning algorithms. A dynamically connected network is used along with domain knowledge and self-play, much in the style of TD-Gammon. The connectivity of the network is dependent on the board position allowing additional domain knowledge, such as spatial relations on the board, to be inserted from move to move on top of that acquired through direct input.

The combination of heuristic knowledge and ANNs was partly inspired by work done by Schraudolph and collaborators where they compared the use of TD methods for playing Go with and without processed domain knowledge [Schraudolph et al. 1994]. They found that a network trained using a minimally processed board position as input, simply the state of each board point being one of *empty*, *black stone*, or *white stone*, took longer to train and played significantly worse than a network with a well designed input architecture. The superior network was designed to take advantage of the symmetries and invariances present in Go. The researchers also made informed choices as to the training procedure in relation to the networks' construction based on the initial experiments with a raw input network. A network was produced that achieved a good enough level of play to beat Many Faces of Go on a 9x9 board.

2.3.2 Golem and WinHonte

Two programs use ANNs in a more traditional way by combining the ML element with game tree search to create a hybrid search algorithm. This seems to be a not unusual practice across the games programming domain as a controlled method of introducing automated learning into an established playing algorithm.

The program Golem was one of the first Go programs to incorporate ML through ANNs [Enderton 1991]. Golem used an evaluation function that incorporated single-target tactical search to determine the status of a string. The connectivity between strings was found by applying some simple rules and fed into the evaluation function. In this way groups of same colour stones that would usually share a common fate, were identified for the next stage. An estimate of a group's strength was then made by calculating the ownership of unoccupied points and therefore finding the eyes or potential eyes belonging to a group. This was part of a two-pass process which included using Golem's territory estimating procedure in the group strength calculations. With each group's status and strength calculated an estimate of the current territory mapping could be made. Golem used this evaluation function, combined with a one-ply search to determine its next move.

Golem used two ANNs, both to give an estimate of how good a particular move was. The first was designed for speed and used to order moves within the single-target tactical search,

the second was slower but involved using more sophisticated input features. The latter network was used to prune moves at the root of the tree, the initial selection of candidate moves available from a given board position. Golem achieved a level of play estimated by the author to be around 14 Kyu.

WinHonte by Frederick Dahl took the application of ANNs to Go one step further by including two networks to take over the job of algorithms that could easily be more expensive for estimating non-trivial concepts [Dahl 1999]. Like Golem, the program was still based on a tree search paradigm using Alpha-Beta pruned Minimax search to find string status and connectivity. This in turn allowed groups to be identified and then group life and death status could be analysed. An initial influence function calculation was performed, after which an ANN was used to assign a group safety value for those groups that had an undetermined life and death status and the influence calculations were updated using that information. This was followed by an evaluation of territory by another ANN and a value for the board was generated. All the above steps were combined to form an evaluation function which was used with a shallow, full-board search to find the best available move. Again in common with Golem another ANN was used to prune moves at the root of the full-board search. WinHonte also achieved a respectable level of play with promising results that go some way to indicate the possibilities of ML in Go programming.

2.4 Use of Evolutionary Computation in Go Programming

A much less common example of ML in Go programming than ANNs is that of Evolutionary Computation (EC). This is a collection of computational systems, paradigms and algorithms that base their models on a selection of the key elements of evolutionary theory [Heitkoetter and Beasley 2001]. Included in this definition are categories such as Genetic Algorithms (GAs), Genetic Programming, Evolutionary Programming and Classifier Systems. Out of these there are only a few examples in the domain of Go programming, the most prevalent of which comes back to ANNs and using GAs to evolve the network weights through competitive evolution rather than by training on a data set.

Some even rarer ideas within the EC domain applied to Go are represented by initial attempts to evolve Cellular Automata (CA) rules to play Go [Rosin and Belew 1995]. Each board

intersection corresponded to a single cell within the CA and the behaviour of each cell was governed by a set of rules that could have the genetic operators of mutation and crossover applied during evolution. A set of CA rules was eventually evolved to beat Wally, a weak public domain Go playing program that has frequently been used for comparison purposes by other researchers [Newman 1994].

2.4.1 Breeding Artificial Neural Networks

Some researchers have taken the approach of evolving a set of weights for an ANN, sometimes including the evolution of a connection scheme, by using evolutionary methods, typically a GA. Using evolutionary methods gives the advantage that a network need not be restricted to a certain structure or any selectable parameter, for example the activation functions of individual neurons could be part of the evolutionary process. Donnelly et al. report experiments in evolving a position evaluation network for Go with a GA [Donnelly et al. 1994]. In Donnelly's experiments the networks were evolved to assign a score to moves on 9x9 boards using the whole board as input. A population of 32 ANNs was used with genetic crossover and mutation to drive the competitive evolution. The population fitness was shown to improve over time; unfortunately no experiments were conducted on the final networks to determine they're playing strength outside of the population.

An alternative method to GAs for evolving ANN parameters is the SANE (Symbiotic Adaptive Neuro-Evolution) system which has been used by several researchers to control the evolution of ANNs for Go [Moriarty and Miikkulainen 1997]. SANE uses two populations, one of neurons and the other of network blueprints, rather than a single population of complete networks. This allows the efficient evolution of structure and weights so that the evolution of each population can react to the other. Richards et al. used this system to generate a network to beat Wally on boards from 5x5 to 9x9, however it was noted that the training time increased rapidly and for larger boards the method may not scale well, a common problem in Go programming [Richards et al. 1996].

Lubberts and Miikkulainen also present experiments with SANE for ANN evolution for Go but focus on a comparison between evolution against a specific opponent, in this case GNUGo, and co-evolution [Lubberts and Miikkulainen 2001]. Competitive co-evolution is

the evolution of two populations each of which consist of individuals that compete directly against individuals from the opposing population as a means of evaluating fitness. The researchers show that co-evolution allows the networks to achieve better levels of play than the SANE method, which appears to stop improving after it can beat its specified opponent.

2.5 Summary

This chapter presented an overview of the field of Machine Learning applied to Computer Go. A brief introduction to the use of Machine Learning for games generally was given and then focus was turned to ANNs and Evolutionary Computation approaches to Go.

The use of the Temporal Difference class of reinforcement learning algorithms has allowed important advances in certain games such as Backgammon [Tesauro 1994] and has been applied to Go in several programs, most notably in NeuroGo [Enzenberger 1996]. Other notable programs using ANNs are Golem and WinHonte. In recent years the prevalence of ANN based research into Computer Go has increased dramatically as shown by the number of published papers and theses on the subject.

The most common use of Evolutionary Computation within the Computer Go field has grown from research into the development of ANNs for Go, primarily as a method of determining the weights and structure of ANNs. Some efforts have been made to investigate the application of Cellular Automata within some success.

An Overview of Game Tree Search Techniques

3.1 Introduction

In graph theory a tree is defined as a connected acyclic graph. Trees can be easily discussed using familial terms, for example a node within a tree may have child nodes and sibling nodes. Each node in a tree can have a maximum of one parent. Typically the tree will have a node with no parent, termed the root node of the tree. For the purposes of this thesis it is only necessary to consider game trees for 2 player adversarial games such as the Minimax algorithm deals with. The tree structure is constructed by finding all possible move options for each player in turn; each ply[1] being represented by one depth level in the tree. The moves are identified given the current game state and the tree is generated to some specified depth.

For many two player adversarial games it is more appropriate to model the game tree with a Directed Acyclic Graph (DAG), a class of graph similar to a tree where nodes may have more than one parent. This allows a node to be reached by multiple paths which is often the case in board games where a particular position can be created by different sequences of moves. Typically a game tree would be represented by a tree data structure with multiple paths to a single state being facilitated through the use of a Transposition Table[2].

3.1.1 Depth-First or Breadth-First?

There are two main ways to search a game tree. The first is depth-first search, which starts searching at the root and continues down a branch until it reaches either a leaf node or the search depth limit. This search method then backtracks to continue down

[1] A single move by one player

[2] See section 3.2

the nearest, unsearched branch. A stack data structure is used to hold the tree nodes in the required order for depth-first searching. If a queue is used to store the nodes to visit then a breadth-first search is generated, the second method of searching a game tree.

Apart from the order of node expansion the two methods differ in the following ways:

1. Breadth-first is always guaranteed to find the shortest solution.
2. Depth-first may find a solution quicker than breadth-first for some trees.
3. Memory requirements. Breadth-first must store all expanded nodes in memory whereas depth-first only needs to store its current path

Iterative deepening depth-first search combines the best attributes of both methods and is discussed further in section 3.2.

3.1.2 Minimax

The Minimax algorithm is a depth-first game tree search algorithm derived from the Minimax Theorem [von Neumann and Morgenstern 1944]. It is a method for choosing the next move in a game where both players take turns to play and are trying to maximise their own score whilst minimising their opponent's. For a Minimax search the player next to move will typically have the position of the maximising player and their opponent will be the minimising player. At each ply the algorithm determines the best available option for the player based on whether they are attempting to minimise or maximise the score. An example Minimax tree is shown in Figure 3.1.

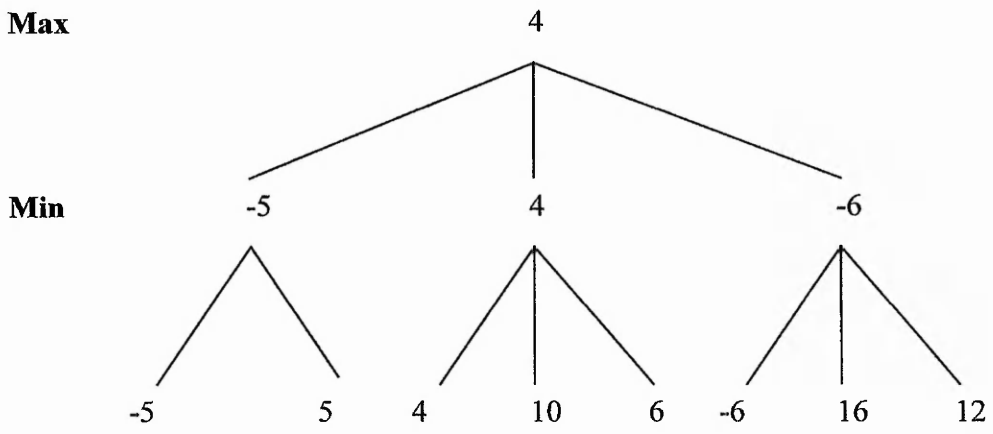


Figure 3.1 - A Minimax Tree

The example Minimax tree in Figure 3.1 shows the final tree state after the algorithm has been run. The value of the root node is the Minimax value of the tree and represents the best result for the maximising player assuming that both players choose their best available moves. The values are generated in a depth-first manner, propagating from the leaf nodes by applying an evaluation function and working the values back up the tree to the root. At each node either the maximum scoring move available for the subtree will be chosen or the minimum scoring move, depending on which players turn it is. Figure 3.2 shows pseudo-code for this algorithm.

```
function Minimax(Node node, integer depth) : integer
    if depth == 0 then
      return evaluate(node);
    else if node == max_node then      // MAX node
      best = -INFINITY;
      Stack child_nodes = expand(node);
      while child_nodes != empty do
        g = minimax(pop(child_nodes), depth-1);
        best = max(best, g);
      return best;
    else                               // MIN node
      best = +INFINITY;
      Stack child_nodes = expand(node);
      while child_nodes != empty do
        g = minimax(pop(child_nodes), depth-1);
        best = min(best, g);
      return best;
```

Figure 3.2 – Pseudo Code - Minimax Function

3.1.3 Alpha-Beta Pruning

There are various ways to improve the efficiency of the Minimax algorithm. The first and most effective technique to consider is Alpha-Beta pruning. This method always produces the same root value as Minimax but with considerably less work identifying sub-trees that will never logically affect the final result. Two values are maintained; alpha, to represent the lowest possible score that the maximising player can achieve, and beta, to represent the highest possible score that the minimising player can achieve. Alpha is initially set to minus infinity and beta to plus infinity.

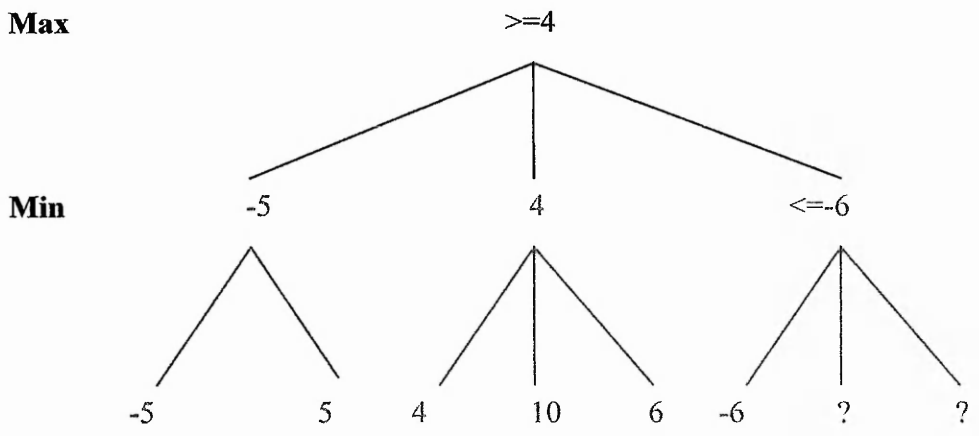


Figure 3.3 - An Alpha-Beta Pruned Minimax Tree

Figure 3.3 shows the Minimax tree from Figure 3.1 with Alpha-Beta pruning applied. This is a fully calculated tree and the root value is 4, the same as Figure 3.1. After the first minimising subtree, marked in red, we know the root value, must be –5 or greater, the bound being stored as alpha. After the second subtree, marked in blue, this has been revised to 4 or greater. The third subtree, marked in green, only needs to have the first child evaluated to determine an upper bounds or beta value for that subtree of –6. Since the previous subtrees guarantee the final result must be 4 or greater we can safely ignore the rest of the green subtree. When alpha becomes greater than beta or beta becomes less than alpha, we know that we can prune the subtree where this bounds overlap occurs. Alpha values are updated at maximising nodes, beta values at minimising nodes. Figure 3.4 shows pseudo-code for the Alpha-Beta enhanced Minimax algorithm.

```
function alpha-beta(Node node, integer alpha, integer beta, integer
  depth) : integer
    if depth == 0 then
      return evaluate(node);
    else if node == max_node then          // MAX Node
      g = -INFINITY;
      Stack child_nodes = expand(node);
      while child_nodes != empty && g < beta do
        g = max(alpha-beta(pop(child_nodes), alpha, beta, depth-1), g);
        alpha = max(alpha, g);
    else                                   // MIN Node
      g = +INFINITY;
      Stack child_nodes = expand(node);
      while child_nodes != empty && g > alpha do
        g = min( alpha-beta(pop(child_nodes), alpha, beta, depth-1), g);
        beta = min(beta, g);
    return g;
```

Figure 3.4 – Pseudo Code - Alpha-Beta Function

3.1.4 Search Windows

Extensions of the Alpha-Beta principle lead on to complicated but more efficient variations such as Aspiration search. Initial values for alpha and beta are chosen to create a narrow search window, which will return either an accurate result within that window or if the actual Minimax value lies outside the window then the search must be run again with a larger window size. This method relies on good guesses for initial alpha and beta values, which may be difficult to find for complex problems. The logical extension of the search window idea is to reduce the window to a minimal size, termed null-window, so that every search iteration will find a cutoff within the tree as quickly as possible and return a bound on the true tree value. The NegaScout algorithm is a high-performance implementation of this idea and was the algorithm of choice for Chess programmers for many years [Reinefeld 1983].

Another null-window based Alpha-Beta variant is MTD(f) [Plaat 1996]. This algorithm is the current best-performing Alpha-Beta variant and is remarkably simple to implement on top of a functioning Alpha-Beta implementation with a Transposition

Table[3]. Like most null-window based algorithms, MTD(f) repeatedly calls the Alpha-Beta function supplying initial values for alpha and beta so that a zero-sized search window is created. At the completion of each iteration a single bound on the Minimax value is returned, either the Alpha-Beta function fails high – above the search window, resulting in a lower bound or fails low – below the search window, resulting in an upper bound. The bounds are then used to determine the values of alpha and beta at the next iteration. The MTD(f) function continues whilst the bounds dictate a greater than zero sized window. When this is no longer true the Minimax value has been found and the search can stop. Repeatedly re-searching the same tree is very inefficient but if a Transposition Table[3] is used to store the searched nodes then the overhead becomes negligible compared to the performance gained.

```
function MTDF(Node root_node, integer first_guess, integer
  depth) : g
    g = first_guess;
    upper_bound = +INFINITY;
    lower_bound = -INFINITY;
    do
      if g == lower_bound then
        beta = g + 1 ;
      else
        beta = g;
      g = alpha-beta(root_node, beta - 1, beta, depth);
      if g < beta then
        upper_bound = g;
      else
        lower_bound = g;
    while lower_bound < upper_bound;
    return g;
```

Figure 3.5 - Pseudo Code – MTD(f) Function

MTD(f) requires a good first guess for initial alpha and beta values for the first call to the Alpha-Beta function. The closer the first guess is to the actual Minimax value the quicker the algorithm will converge. A simple method for producing a good first guess could simply be to apply the evaluation function to the root position.

---

[3] See section 3.2

3.2 Minimax Enhancements

Many methods have been developed to make Minimax search more efficient – both in terms of memory usage and search speed. From an algorithmic point of view reducing the potential search space is the most common direction to approach the reduction of these criteria. One enhancement that lays the foundation for many more is the use of a Transposition Table (TT). This is a table of data used to store information about previously searched nodes in the game tree, such as alpha-beta values or bounds including the depth to which these values are valid, saving the time required to recalculate these values. The TT also provides a practical benefit for coding efficiency – if the creation of a node requires substantial memory allocation then the time to recreate the node and re-allocate the memory can be saved.

3.2.1 Iterative Deepening

Iterative Deepening (ID) is a technique that combines the best attributes of depth-first search and breadth-first - completeness of search and a guarantee of finding the shortest path from breadth-first search and minimal memory requirements from depth-first. ID repeatedly performs depth-first searches with an increasing depth limit. The repeated searches would make this process very inefficient unless a Transposition Table is used to reduce the practical costs of memory-reallocation and re-searching the subtrees.

```
function IterativeDeepening(Node root_node, integer max_depth)
    for depth = 1 to max_depth
      minimax_value = alpha-beta(root_node, depth, -INF, INF);
    return minimax_value;
```

Figure 3.6 - Pseudo Code – Iterative Deepening

The implementation of ID is very simple as can be seen from Figure 3.6 and it is essential for an efficient MTD(f) implementation, the call to `alpha-beta()` being exchanged for a call to `MTDF()`. The *minimax_value* variable should be fed back each iteration into `MTDF()` as the first guess.

3.2.2 Best Move First and Enhanced Transposition Cutoffs

Two further techniques are made possible by the presence of a TT – Best Move First and Enhanced Transposition Cutoffs (ETC) [Schaeffer and Plaat 1996]. Both are simple additions to the search algorithm, giving variable performance improvements dependent on the problem and search tree shape.

To facilitate the implementation of Best Move First each TT entry should include a reference, hash key or other note indicating the best child for each node as ascertained from lower depth iterations or previous searches. The Alpha-Beta function uses this information to investigate the best child as the first node, even before generating the full set of children for the current node. Often this will yield a cutoff and save the time needed to consider the other inferior children.

ETC uses the stored alpha and beta bounds in the TT from previous searches[4] to check if any expanded children of the current node will cause a cutoff without requiring the recursive depth-first calls to `alpha-beta()` on each child. This can increase performance by reducing the tree size but the extra cost of checking the alpha beta values for every node must be weighed against the benefits.

3.2.3 Move Ordering

An Alpha-Beta search always aspires to create a minimal search tree and in a tree with perfect node ordering, such that the minimum number of nodes is examined, the maximum number of cutoffs will be generated. Improving the ordering of the nodes will improve the algorithms performance.

The History Heuristic is one method to create a better ordering of nodes through the identification of moves which have shown themselves to be good in other parts of the tree or in previous searches. A table is maintained in which a tally for each move is recorded and incremented every time that move is shown to be the best move from a

[4] Or branches – hence Enhanced *Transposition* Cutoffs

node or causes a cutoff [Schaeffer 1989]. Moves which never occur as best moves or cause cutoffs need not be stored. The History Heuristic scores are used to order the expanded nodes and in this way the performance of specific moves, not nodes, across the tree can be monitored and the success translated to other parts of the search tree.

3.2.4 Local Search Area

In particular to board games one obvious enhancement is to restrict node expansion to the minimal relevant board area. Calculating this area is a problem specific issue but is essential for large search trees that can be defined within a set of bounds.

3.2.5 Forward Pruning

When all of the above techniques have been covered but the tree is still too large to search in the allocated time or memory available then forward pruning could be considered. This is a class of techniques that apply rules or methods of evaluating a node to decide whether or not it should be pruned without being expanded. Depending on the search problem some forward pruning techniques may be admissible – admissibility being defined as a technique that is guaranteed to *not* cause the value of the search tree to differ from its true value. For instance a board position in a game may occur such that a winning terminal state cannot be achieved from that position. Such a node can be safely pruned for the player trying to reach a winning state if they have better sibling nodes available.

Inadmissible heuristics have no such guarantee; they may result in an incorrect tree value being calculated due to pruning branches that would have influenced the real tree value. An example of forward pruning would be generating a heuristic score to represent the worth of a move and pruning all moves that score below some threshold. To handle very large searches forward pruning can be essential, however the choice of using such methods reduces the 100% confidence level that a fully admissible search guarantees, so they should not be used where confidence in the result is critical.

3.3 Evaluation Functions

An evaluation function is usually specific to a class of problems for any particular game, such as achieving checkmate in Chess or capturing a given string of Go stones. This is the function that will drive the Minimax search from the terminal nodes of the tree.

Search algorithms may be sensitive to the granularity of the evaluation function, i.e. MTD(f) benefits from a coarser grained evaluation, meaning fewer passes are required to determine the Minimax value. Of course this trades evaluation accuracy for speed, but this is an important balance that will be specific to the application and structure of search algorithm chosen.

In the case of Go an accurate global evaluation function to determine the current score would require recursive searches to determine the status of stones on the board. Not until the very end stage of the game is the board position stable enough to divide into self contained sub-games that will have a negligible effect on adjacent sub-games. No function has been found that will give an accurate score without some form of status search at the very least. This is one of the reasons that Go is so difficult for computers to play.

3.4 Alternatives to Minimax

A number of alternatives to the Minimax paradigm have been created over the years and a brief description of a few is presented here.

Proof Number search (PN) focuses on AND/OR trees and has proven to be a viable alternative to Alpha-Beta search where a binary goal is defined, for instance capture goals in the case of Go – either a string will or won't be captured [Allis et al 1994]. PN search is a best-first search algorithm, always expanding the most promising looking node first. Most best-first search algorithms require a heuristic function to determine the best node available; PN search uses the shape of the search tree, i.e. the node dispersal pattern of internal nodes and the values of the leaves. This allows PN

search to efficiently handle non-uniform branching within trees – particularly appropriate for games where sequences of forced moves must be played out, creating an unbalanced tree. On the downside the memory requirements can be high since the whole game tree must be kept in memory during the search. Some attempts have been made to reduce the memory requirement [Winands 2001].

Cazenave's Gradual Abstract Proof Search (GAPS) is a blanket algorithm that can be laid over Alpha-Beta search or PN search amongst others [Cazenave 2002]. GAPS combines Abstract Proof Search – Cazenave's earlier highly selective, best–first search algorithm, with Lambda Search [Thomsen 2000] to maximise the algorithms control on search behaviour, whilst utilising abstract game properties to implement very high move selectivity.

The possibility of a fuzzy evaluation, perhaps one where a confidence value is assigned along with an evaluation score, leads to the question of how to represent and use such values in the context of tree search. The concept of Fuzzy Numbers in search is defined as the uncertainty of the evaluation of the problem state [Junghanns et al. 1995]. This definition generates several issues when using fuzzy numbers in search, such as having to define new min/max operators to function on fuzzy values and developing new fuzzy evaluation functions. The primary motivation behind Junghanns' work on Fuzzy Number search was to demonstrate the worth of using fuzzy numbers to express uncertainty in search – a highly relevant topic for games even in the case of well-defined goals such as capture, where the unknown state may be represented as levels of safety using fuzzy numbers, so propagating more information than might otherwise be gained through a standard Minimax search.

Some search methods attempt in some way to tackle a long standing problem of Minimax search – referred to by Junghanns as the Scalar Value problem. This reflects the fact that for a game of any level of complexity, Minimax always deals with a single value for evaluation. The state of very simple games may be well suited to being represented by a single scalar value however for more complex games, a lot of highly relevant information found during the tree search is lost in the conversion to a scalar value. Further research towards propagating more substantial information from

leaf nodes to other parts of a search tree may well produce interesting results but it is by no means a simple task.

3.5 Summary

This chapter has given an overview of common game tree search techniques used in two player adversarial games, where a game tree may be constructed by alternately taking either players legal moves. Given an evaluation function for leaf nodes, such as the value of a board position resulting from this path in the tree, it can then be evaluated by the Minimax algorithm to determine the maximum value of each move in the tree for either player. This is based on the idea that both players will be attempting to increase their overall score in the game whilst reducing their opponents score.

Many methods are available to enhance the performance of a Minimax game tree search such as Alpha-Beta pruning. This technique allows parts of the game tree to be pruned away so that it is not necessary to spend the time to calculate every node in the tree. This is the most popular and most commonly effective improvement to Minimax. Importantly it is also an admissible technique – it is guaranteed to always give the same final result for a tree as Minimax itself. Some techniques can be formulated that are inadmissible – they remove branches from a tree based on a probability that the branch will have no affect on the evaluation of the tree. This can result in an incorrect evaluation however for very large search spaces the trade-off of using forward pruning against the possibility of receiving an inaccurate answer must sometimes be made.

Search windows are discussed in this chapter as one logical utilisation of Alpha-Beta values. A window between two values is searched within and when a value in the tree is generated that is outside this window it causes a cutoff and a re-evaluation of the tree based on this cutoff value. The window is reassessed to incorporate the cutoff and shifted in the direction that the cutoff implies the tree value is towards. This technique can be very successful and has lead to state of the art algorithms such as MTD(f) and NegaScout which have both been used in highly rated Chess programs.

Further discussion of simpler enhancements to Minimax takes place in section 3.2, including Transposition Tables, Iterative Deepening and Move Ordering. Some dissussion is also made on the usefulness of different grained evaluation functions and how they interact with some Minimax enhancements. Finally alternatives to Minimax are discussed with particular consideration given to Proof Number Search, Cazenaves' Gradual Abstract Proof Search and Fuzzy search.

An Introduction to Artificial Neural Network Theory

4.1 Introduction

This chapter describes the theoretical basis behind Artificial Neural Networks (ANNs) and includes brief descriptions of the main classes of training and their associated algorithms. ANN techniques provide methods to optimise a set of parameters to produce some expected output from a function and belong to the family of techniques that include Simulated Annealing and Evolutionary Computation [Heitkoetter and Beasley, 2001].

An ANN is a collection of simplistic processing units connected by a number of free parameters, termed weights, and is used to represent a complex function. The weights of the network are modified using a training algorithm to optimise the function represented by the network and in this way the network can be perceived to learn. The simplest ANN possible is called a perceptron, a single neuron with j input weights. This has been shown to be limited to learning linearly separable functions [Minsky and Papert 1969]. For non-linearly separable functions, such as the exclusive OR function a multi-layer perceptron must be used with non-linear activation functions. This is discussed in section 4.2.2. In fact multi-layer ANNs with non-linear activation functions and enough hidden neurons have been shown to be very powerful tools capable of mapping any function with any number of parameters [Hornik et al. 1989].

One of the strongest properties of ANNs is the ability to generalise over the data they were exposed to during training. Given the correct training conditions an ANN is capable of giving a useful response to unseen input based on its training experience. For problems where all possible input cannot be realistically accounted for and trained upon, ANNs may offer a suitable solution.

4.2 Architecture of an Artificial Neural Network

There are many different types of ANN but they can all be distinguished by the presence of certain elements; a set of neurons, a definition of the network's connectivity (its architecture), a set of rules dictating the neurons' response to input and a learning algorithm. A typical ANN will have its neurons organised into layers, an input layer, an output layer and zero or more hidden layers. The number of neurons in each layer required to achieve a satisfactory standard of learning is problem dependent, as is the number of hidden layers. The connections between the neurons are referred to as the weights and each has a value assigned to it. These weights are the variables that a learning algorithm will adapt. It is usual to assign random values to the weights initially, although a careful selection of initial values may increase the efficiency of the chosen learning algorithm.

A single neuron would have a number of input connections any of which may be activated with a signal depending on the pattern of input. The total input into a neuron with n inputs is calculated as the summed total of the input signal from of each of its incoming connections, x_i, multiplied by the associated weight of the connection, w_i. An optional activation function, $f()$, may be applied to the weighted summed input to produce the units output, y (1). The input neurons will usually have an activation function equivalent to the identity function so that the activation of the neuron is the same as the net input to the neuron.

$$y = f\left(\sum_{i=0}^{n} x_i w_i\right) \tag{1}$$

4.2.1 The Bias Term

It is common to have an additional weight for each neuron called the bias term. This can be thought of as originating from an imaginary neuron that has a permanent activation of 1. The bias term is sometimes called the threshold value of the neuron. Representing the bias term as a weight from an external neuron allows the term to be learnt just like other weights. In (1) x_0 is always equal to 1 to represent the bias neuron output, the bias term itself being w_0.

For a neuron with N inputs an N-dimensional space can be defined in which exists a hyperplane. This N-dimensional space is shared by all neurons within a single layer, each with their own hyperplanes, the positions of which are defined by their neuron's input weights. The hyperplane of a neuron defines a boundary between its on/off state, hence the use of the word threshold for the bias term. For neurons with non-binary output this boundary will have valleys and ridges such that a range of output values on either side of the boundary is possible.

Without a bias term the boundary, represented by a hyperplane in binary output neurons, is constrained to always pass through the origin of the input space. The use of a bias term allows the boundary to move away from the origin, giving greater flexibility to what individual neurons can represent in the input space. Depending on the problem and the choice of activation functions, even a multi-layer network with enough hidden neurons but without bias terms, will be restricted in what it can learn.

4.2.2 Activation Function

There are many valid choices for the activation function of a neuron, another factor that must be tailored to suit the problem. The activation or squashing function, as it is sometimes called due to it's 'squashing' of a value into a certain range, is applied to the summed weighted input to a neuron.

Activation functions may be linear or non-linear, however if the expected solution for an optimisation task is non-linear then a multi-layer network using non-linear activation functions will be necessary. These activation functions introduce non-linearity into a network making it capable of approximating non-linear functions. A multi-layer network with linear activation functions can always be reduced to a single layer network with linear activation functions, so the use of multiple layers alone will not help with non-linear problems. This can be demonstrated by examining the affect of the association law of matrix multiplication on multi-layer networks [Callan 1999]. For each layer in the network we have a matrix of neuron output values and a matrix of weights to multiply them by. The association law of matrix multiplication tells us that for the matrix multiplication A*B*C, where A, B and C are matrices, a matrix D exists that is equivalent to B*C.

In ANN terms this means for a multi-layer network with linear activation functions, an equivalent single layer network exists where the one layer of weights is equivalent to all the layers of weights combined in the multi-layer network.

The sigmoid function is a good example of a non-linear activation function that is often used in ANNs. It changes the summed weighted input into a continuous range from 0 to 1:

$$f(net) = \frac{1}{1+\exp(-net)} \tag{2}$$

4.3 Learning With Artificial Neural Networks

To train an ANN you need to choose a training method suited to the task. There are three classes that training methods can be divided into:

a) Supervised learning, a class of methods that attempt to map a set of inputs to a set of outputs supplied as examples of desired behaviour e.g. backpropagation.

b) Unsupervised learning, a class of methods that use a set of example inputs and extract clusters in the data via a specified function, typically a distance function e.g. self-organising map. This type of learning is useful for classification problems such as recognising handwritten characters.

c) Reinforcement learning, a class of methods that attempt to learn a map of input to output (or reaction which may be a more relevant term for this class) to maximise a reward signal based on the performance of the network through the use of the mapping.

The focus of this research concerns supervised learning. For readers that would like to know more about unsupervised learning they are referred to the references. Section 4.3.2 describes reinforcement learning using Temporal Difference methods only, as background for a short experiment described in a later chapter.

4.3.1 Supervised Learning

4.3.1.1 Backpropagation

The most common learning algorithm used to train an ANN is the backpropagation algorithm, an example of supervised learning [Rumelhart et al. 1986]. After an initial forwards propagation of input values through the network to calculate it's output, this algorithm uses backwards propagation of error values to update the weights. A generalised form of the Widrow-Hoff or delta learning rule is used to adapt the weights during the backwards error propagation, reducing the function error by effectively using steepest descent minimisation. The generalised delta rule has been derived for multi-layer perceptrons [Rumelhart et al. 1986] and is presented below, where w_{ij} is the weight connecting neuron i in the current layer to neuron j in the next layer. x_i is the output from neuron i, δ_j is the error value from neuron j and η is the learning rate:

$$\Delta w_{ij} = \eta \delta_j x_i \tag{3}$$

In addition to the generalised delta rule the backpropagation algorithm requires a formula to propagate the error values from the output layer back to the input layer. The error for an output layer neuron is simply the target value, t_j, minus the actual output value, o_j, multiplied by the derivative of the activation function $f'(net_j)$ presented in (6):

$$\delta_j = f'(net_j)t_j - o_j \tag{4}$$

The error function for neurons in the hidden layer is shown in (5), where k is the ANN layer after j, one layer closer to the output layer:

$$\delta_j = f'(net_j)\sum_k \delta_k w_{kj} \tag{5}$$

For backpropagation a non-linear activation function is required that should be continuous, differentiable and monotonically increasing [Callan 1999]. The sigmoid function (2) fits these criteria and has a simple derivative:

$$f'(net_j) = f(net_j)[1 - f(net_j)] \quad (6)$$

4.3.2 Reinforcement Learning

Reinforcement learning (RL) is based on the idea of rewarding good performance and punishing bad performance as an agent explores its environment by trial and error. There are many ways to attempt to create a solution to an RL problem, ANN techniques being one of them. Other techniques include Genetic Algorithms, dynamic programming and Q-learning [Watkins 1989]. The general class of problems where the reward is delayed in time relative to the actions that led to that reward is of specific relevance to the work presented in this thesis as it equates to the situation where we have many game records and their associated outcomes but have no idea how to assign rewards or punishments to an agent (in our case an ANN) for selecting individual moves during that game. The Temporal Difference (TD) class of algorithms attempt to solve this credit assignment problem [Sutton and Barto 1998].

Given a series of states leading to some reward, in the case of Go a game record leading to a win or loss by a number of points, the TD method takes each state in order and uses its own output as an estimate of the correct output. This estimated output is used with the observed reward generated in the next time step to allow an immediate update of the output estimate without having to wait until the end of the state sequence to calculate the reward. Monte Carlo methods are another example of RL but in contrast to TD they must wait until the end of the state sequence before updating their output estimate. The process of updating an estimate based on an existing estimate like this is known as bootstrapping. The extension of TD learning for ANNs is presented in detail in [Sutton 1988].

4.4 Summary

This chapter has given an introduction to basic Artificial Neural Network Theory. ANNs are collections of neurons, simple processors which combine their input perform some operation on it and produce an output value. These neurons are connected to form a network where the connections between them, known as weights, can represent parameters in complex functions. By using ANN training algorithms these weights can be optimised to better approximate a function that is representative of the training data being fed into the networks.

The first half of this chapter discussed the architecture of a basic ANN along with simple theory such as how the output of a neuron is calculated and the purpose of the bias term and activation function. The second half focuses on learning algorithms and has a brief summary about supervised learning and reinforcement learning.

Applying Artificial Neural Network Techniques to Go

5.1 Introduction

This chapter introduces the investigation of Artificial Neural Network (ANN) techniques and their application and incorporation into a hybrid Artificial Intelligence system. The ideas developed for using ANN methods within a Go playing program and the designs created from those ideas will be examined. Further to this the investigation will include a comparison of different input field sizes, contrasting naive and feature rich input schemes, and a look at alternative training methods.

5.2 The Artificial Neural Network Ideas

Many uses for ANN methods can be imagined for application to the problem of playing Go. This section details the ideas that were pursued to the design and implementation stages and presents a brief summary of those that only made it to the design stage.

The intention of the investigation into ANNs was specifically to increase the efficiency of resource intensive game tree search methods. The route focussed on was to create an ANN to generate a score given a local board position. The network would be centred on a target point where a potential move could be made and the input could possibly include some pre-processed features. The score assigned to the proposed move could be used to order nodes in a game tree, prune them if desired using a defined minimum quality move threshold or even to allow the search to determine when to pass according to the availability of high-scoring moves.

Using an ANN to score potential moves presents two problems; what input is relevant to scoring a move and what do we mean by 'score'? There are no known methods that can be used during a human game to give an accurate, defined value to any given available move. However since it is a deterministic game involving no random elements it should be possible

to assign a precise score that represents the worth of playing a certain move at a certain time. ANN theory indicates that a multi-layer perceptron should be able to learn a function with an unlimited number of parameters, given enough adaptable weights within the network. The complexity of the game has meant that no such function has been developed yet and it is highly unlikely to be discovered in the near future. One limiting factor to this development is the computational capacity of current hardware. To help deal with these problems it maybe necessary to estimate the move value function using smaller than optimal network sizes and efficient input representation and pre-processing. Generating a high quality function estimate was the focus of the ANN designs.

The obvious solution to selecting relevant input is to use as much raw input data as is available and let the network classify it. This was one of the initial experiments with further work to refine the input representation. The use of the phrase 'to score' could not be used in the absolute 'one move is better than another' sense so the title of plausibility score was adopted, indicating the score produced by the trained ANNs was some approximation to how likely this move was to be played in a manner that would favour the player placing the stone. These networks could then be used to roughly order moves independently of each other. Ideally they should be ordered relative to each other and this idea was pursued and will be discussed later in section 5.3.4.

Now that the purpose of the network has been defined, to order moves according to a plausibility estimate, we must now consider how to use it within a Go playing program. The simplest method would be to score all possible legal moves at a given position and select the highest scoring move to be played but this ignores the fact that the ANN will have learnt only a subset of the knowledge and skills required to play a game of Go.

5.2.1 A Move Scoring Network

The main focus of research has been the move scoring network, a simple fully connected, multi-layer, feedforward network. Many different input schemes were formulated, the most promising of which are detailed here. The generic network design had three-layers with at least one input neuron per board point in the input field, based on a 9x9 section of board centred on the proposed move. Only one output neuron was required to give a plausibility score for the proposed move. The board state was taken just before the move was to be

played. This meant the point the move would potentially occupy was always empty. Other researchers have conducted different field size and shape experiments at the same time as the author [Van Der Werf 2002]. The experiments presented here aim to be a comprehensive set concerning more than input field shape and size, including input representation, the effect of extra features, training algorithms and parameters, network architecture and network use.

As mentioned above the network could, at its simplest, be used to select the highest scoring move to play. When thought of in conjunction with game tree search two options become available. Firstly the network might be used to order nodes within a search tree. The performance of the alpha-beta pruning Minimax algorithm can be improved enormously with efficient node ordering. In contrast to the first, the second use is a method that may invalidate the accuracy of an alpha-beta game tree search. Forward pruning can be used to remove nodes in a tree before the algorithm has logically proven they can be safely removed and will have no influence over the final search result. When forward pruning is used a score is generated for each node using the ANN, allowing pruning based on this value. Using the ANN in this manner could be beneficial for time-critical problems, allowing stricter control over the size of the search tree and therefore the time taken to search it.

5.2.2 Symmetry and Anchoring the Network

Several options were available to indicate to the network the move to be scored which in turn raised the question of how to anchor the input pattern extracted from the board. The pattern could be taken as it is, leading to symmetry redundancy issues, or the pattern could be orientated and translated so that the proposed move is always anchored to the top left corner of the input representation. The second proposal would mean that the important board state information around the move being scored would only be included from one quarter of the move's local environment. A move centred network seemed the appropriate choice to include the maximum board information as well as allowing rotation and reflection of the input pattern for training purposes, pivoting on the scored move.

A board position in Go has the same properties as its colour reversed position, just as it does if subject to reflection or rotation. This meant there was a large amount of redundancy in the training set due to the symmetries of rotation, reflection and colour. The first two symmetries are clear; a board pattern may be rotated and/or reflected to produce another pattern. The

third, colour symmetry, is produced by swapping the colours of the stones, all black stones become white and all white stones become black. The legal properties of a position are unaffected by symmetry, all three symmetries when applied to a legal position will always produce another legal position.

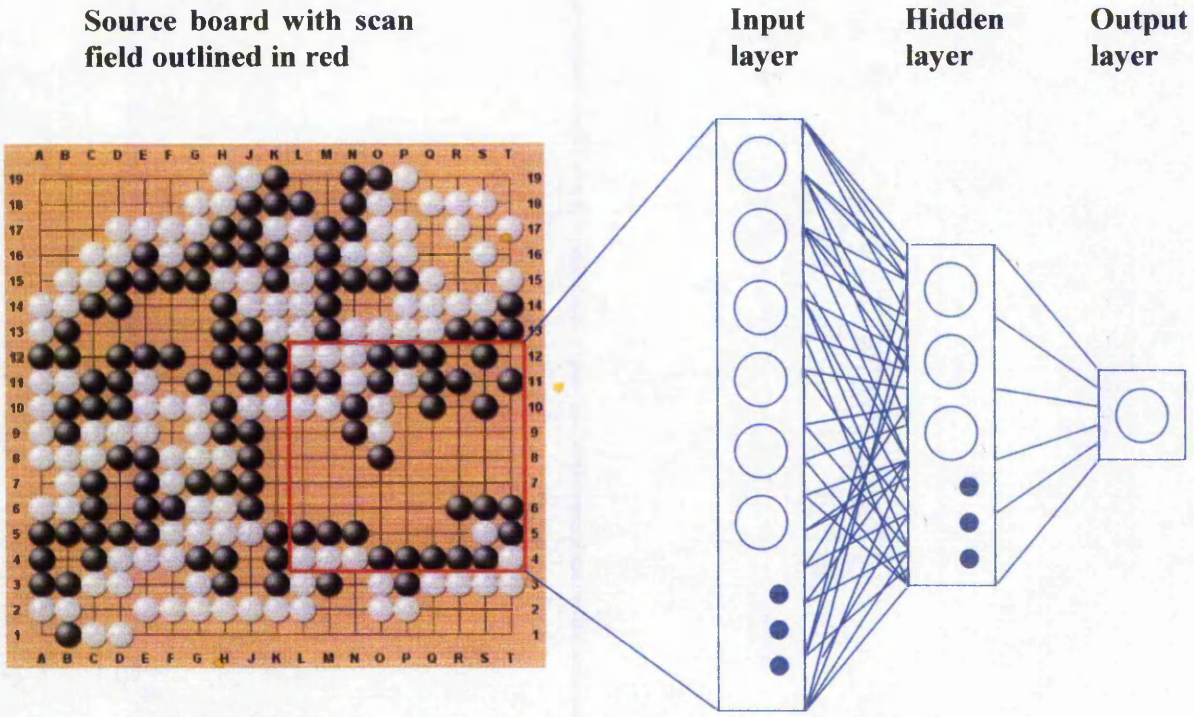


Figure 5.1 - Input Scan Field

To reduce the 16 potential patterns arising from a single position, it was rotated so that the closest board edges were always to the top left, providing a constant anchor for the ANN. Whenever a pattern was presented to the network, in actual use or in training, the pattern was anchored accordingly. This didn't entirely remove the redundancy created by the rotational and reflectional symmetries but did remove it where the symmetry was broken by the board edge, such as in commonly played patterns like opening sequences, termed joseki, normally played near to the corner of the board.

To allow the network to learn from the point of view of black and white the network needed to be presented with a pattern and its colour reversed version, effectively doubling the training time. One solution to this was to always present a pattern from a single colour's point of view, requiring a colour reversal of all stones on the board when the board is presented from the opponents viewpoint. An improvement on this process was made by modifying the abstraction from actual input - black stone, white stone, empty point to ANN input values, to take into account the point of view required. This created a simple mapping from *ourcolour*, *theircolour*, *empty* to the network input layer. This method of dealing with colour symmetry required a check at each point when pre-processing the network input, whether the stone at the point was the current players colour, *ourcolour* or the opponents, *theircolour*, however it did remove the need for a colour reversal step. The move ordering network is shown in Figure 5.1.

5.2.3 Input Field

The choice of a 9x9 input field was made after some brief experiments into field size, the preliminary results indicating that a 9x9 board section would give the best results. Further experiments in field size have been conducted after more reliable and efficient input schemas were implemented and have given some alternative results, which will be discussed later in this thesis. The ideal input field size for Go played on a standard 19x19 board would be 38x38 to maximise available board state information to the ANN. Since the network is centred on the move being scored, 38x38 is required to encompass the entire board for all possible move positions. However, actually training such an enormous network would have many practical problems. Time and hardware would be required in copious quantities so any method for reducing the number of neurons in this network must be used. To give some idea

how long it took to train earlier implementations, the first 9x9 designs took several months to train to stability and that is with a mere 81 input neurons. For 38x38 it would require 1444 input neurons for this early state. With later, more efficient designs where multiple state neurons per board point were used, it is closer to 6000 neurons needed for input of the raw board situation alone. Along with this the intended purpose of the ANN, to be used within a game tree search mechanism for move ordering and branch pruning, meant that the network would be used hundreds, maybe thousands of times a second. A small efficient network was required for such use, so ways of incorporating and identifying pertinent information outside of the input field was essential.

5.2.4 Input Features and Schemas

A lot of information can be extracted from a raw board position. One of the challenges of designing a network was deciding which features would contribute positively to the network's performance and learning capability. Some impacted negatively on the ANN; they overbalanced or misled the network during learning due to the feature being strong relative to the others.

One feature that was found to produce a negative effect on the performance of an ANN trained to play Go by other researchers was the distance to the last move played and this will be discussed in the next chapter.

Input Scheme		State Neurons	Distance Neurons	Other Features
S1	**- Raw**	1	0	-
S2	**- Binary states**	2	0	Binary encoding of states
S3	**- 3 states**	3	0	-
S3D2		3	2	-
S3D18	**- Binary distance**	3	18	-
S4	**- Offboard state**	4	0	-
S4D2		4	2	-
S4D18		4	18	-
S4D18Libs	**- Liberties**	4	18	Liberties of surrounding strings.
S4D18DLM	**- Distance to last move**	4	18	Distance to last move.
S4D18ELM	**- Emphasise last move**	4	18	Last move emphasised in input.

Table 5.1 - Input Scheme Details

Redundancy within feature sets can easily occur, so experiments were conducted with networks with varying input architectures to assess the impact of the features. The full set of features available and the actual input schemas used are detailed below and summarised in Table 5.1.

The initial network design (S1) used only raw input from the board with a very simple encoding scheme, 1 neuron per point, with the value indicating the board point content as follows: *black*(+1), *white*(-1), *empty*(+2), *offboard*(-2). A useful technique was quickly discovered from this design and incorporated into the future designs. By supplying individual neurons to represent each possible state of a point on the board the networks were able to train to a much higher standard than the raw network. Various combinations of states were investigated to determine which were essential to the quality of training and to what degree. This included using 2 neurons per point to represent only the states of *ourcolour* and *theircolour* (S2), adding a third neuron to define the *empty* state (S3) and a fourth neuron to represent the *offboard* state (S4). For the networks that did not have an *offboard* state defined an implicit encoding was used such that none of the state neurons were activated where an ANN input point was located off the board.

No explicit encoding of the distance to the board edges was included in the initial multiple state input schemes however it quickly became obvious that including such information might benefit the network's learning capability. To begin with 2 neurons were used to encode the distance as simple integers in designs S3D2 and S4D2. This led to the refinement seen in S3D18 and S4D18 where 18 neurons were included for distance encoding allowing 9 binary indicators to input the distance value to each of the two nearest board edges. For a distance of *n* only the *nth* neuron in each group of 9 neurons would be activated (+1), the others being inactive (0).

To reduce the processor time required for training and using the networks, a more compact input scheme was explored to encode the board point states. A binary encoding of the 4 available states that was successfully used by other researchers [Schraudolph et al. 1994] using 2 neurons per board point was tested. The states were represented as follows: *ourcolour*(+1, 0), *theircolour*(-1, 0), *empty*(0, +1) and *offboard*(0, -1). This representation had the benefit of allowing two classes of state to be easily distinguished by the network at

the input stage, that of the presence of a stone – *ourcolour* or *theircolour*, and the absence – *empty* or *offboard*.

Three schemes were designed to test what appeared to be the most promising features to include. The first used the liberties of the immediate strings surrounding the point being scored as additional input (S4D18Libs). Each direction, north, south, east and west, had 4 dedicated input neurons, allowing one of each set of 4 to be activated to indicate whether 1, 2, 3 or >3 liberties were available to the string in that direction. If no string was present in a particular direction then none of the neurons were activated.

Two schemes were concerned with features relating to the last move played. The scheme S4D18DLM used a single neuron to input the Manhattan distance to the last move and S4D18ELM emphasised the position of the last move by supplying a stronger input at that point compared to the surrounding points.

5.2.5 Game Stage

It had been recognised that there was scope for networks to be trained for use during specific stages of a game of Go, however this brought up the question of how to decide what stage the game was in. In Go this is a non-trivial problem as you may find different sections of the board developing faster than others and you may find moves more usually associated with the endgame being played within the first 50 moves of a 250 move game. Hence simply counting how many moves have been played would not serve as a good indicator of the current game stage. Due to this problem and the estimated cost of solving it, it was decided to concentrate on the simpler network training ideas and keep this idea for future development.

5.3 Alternative Uses

Many alternative uses of ANN techniques were considered and some of these ideas were selected for further investigation. Amongst these were Urgency Networks, Safety Estimating Networks, Temporal Difference trained evaluation functions and competitively trained move scoring networks that used a hybrid training scheme involving stages of supervised and unsupervised learning [Cook 1999].

5.3.1 Area Finder Network

After developing the first move scoring network designs it was realised that an area selection mechanism could help to focus a move search on relevant board areas. This would create a more efficient move selecting algorithm and open up areas of research that would otherwise have been impractical to pursue. Some work has been conducted by other researchers into segmenting a Go board during a game to divide it into more manageable sub games [Müller 1995]. Identification of areas of linked stones or just empty points could help to reduce the number of moves being scored by the ANN and possibly save thousands of matrix multiplications. A brief exploration of the effectiveness of such a mechanism and how it might be implemented as an ANN was explored in earlier research [Churchill 2001]. During this research a network was implemented and trained with the aim of it being able to identify a single sector, $1/9^{th}$ of the board, to focus on. This was a naive design, ignoring the influence of stones over one another across the board and simply attempted to discern any pattern that may have been apparent in the training data. As it turned out the results were unfavourable, but directions for future research incorporating a better segmentation scheme and connectivity analysis between stones became apparent with the possibility of producing a useful mechanism to focus the move scoring network.

5.3.2 TD Evaluation Function

Some marked success has been achieved in the application of AI techniques to games through the use of a class of methods named Temporal Difference (TD) [Sutton 1988]. TD-Gammon, a self-teaching backgammon playing computer program was the first to use ANN techniques, specifically TD, to achieve near grand master level play [Tesauro 1994]. The initial experiments with the TD(λ) algorithm for TD-Gammon used no built in backgammon knowledge and still managed to achieve a strong, intermediate level. This indicated great possibilities for the games domain. There has been much research on applying the TD(λ) algorithm to Go [Schraudolph et al. 1994;Enzenberger 1996] with varied levels of success. Enzenberger's NeuroGo program involved the use of a dynamic ANN architecture to better represent the relationships between stones in a board position, expanding on ideas presented by Scraudolph et al. The network was trained using the TD(λ) algorithm and has proven to be

a strong player with good results at tournaments and in the Computer Go Ladder [Computer Go Ladder].

TD methods are a type of reinforcement learning, one of a class of learning methods that can be useful when no output pattern is known at the time of training. An output signal from the network is generated after exposure to some input signals from it's environment, in the case of Go this could be a board position and the output a measure of the quality of that position with respect to a particular player. A reward or reinforcement signal is then generated to indicate how good or bad the networks output was and is fed back into the network. Often the reinforcement signal is delayed until a set of input patterns has been presented, for instance this could be after every position in an entire game has been processed. This is a particularly important point since it is not until the actual end of a game that the winner is known and hence how well the network has performed and a final, accurate reinforcement signal to reflect the performance can be calculated. Until this point the reinforcement signal is generated using the network's output as an estimate of the desired output state.

Initial experiments were pursued using TD to train an ANN as a board evaluation function. The results were not encouraging enough to dedicate more research time to TD.

5.3.3 Safety Estimating Networks

Bouzy's 5/21 influence estimation algorithm [Bouzy 2003a] is a widely used method for calculating the influence radiated by stones across a board and can be used as the basis for an evaluation function. A major problem that all evaluation functions based on the static analysis of a board position have, including Bouzy's 5/21, is that correct knowledge of the status of the stones is essential for a reliable evaluation. For all but the simplest enclosed cases, this information usually requires detailed searches through the potential game tree. These additional searches can dominate the resources used by the evaluation function and for any reasonably developed game position they will be complex. The searches may have competing goals, creating precedence problems, so a faster, more efficient alternative than a complete tree search to determine status must be found. The idea behind a safety estimating network was to create and train a network that did just that. Attempts to implement such a network have been conducted by the creators of WinHonte [Dahl 1999] and Golem [Enderton 1991] as briefly described in Chapter 2. From the author's personal observation WinHonte has

achieved some success in this area, which remains a strong point of the program's playing technique, although no opportunity has become available for the author to investigate the use of such networks in Golem.

An ANN solution would seem the obvious choice for the development of a safety evaluation function introducing the intuitive element of experience, which human players can harness so readily and apply to a board position with a glance. Only when confronted with an urgent position or one where a degree of confidence in the stability of the stones is lacking, must a player resort to reading out the local situation – and so it should be for a computer program.

The design settled upon for the network required the output to be a value between 0 and 1 to indicate the probability of the string of stones targeted being alive at the end of the game. An output of 0 meant the stones were unconditionally dead and an output of 1 meant the stones were unconditionally alive. Values between these extremes represented an estimate of the chance of life. The implementation of the network used the S4D18 input schema.

The network was trained using the same training data as the other experimen harvested ted from NNGS. A pre-training analysis phase was conducted to generate appropriate input/output training pairs. First of all for each stone that was captured during a game an input/output pair would be generated at the point of capture, where the input vector was an area of the board centred on the stone about to be captured and the output vector was 0 indicating unconditional death. The algorithm then stepped backwards through the game to the point the stone was placed on the board. For each game step a new input/output pair was created with the output vector increasing linearly so that upon reaching the placement position the output vector equalled 1.

A second reverse iteration of the training game focussed on positive examples. The same number of living stones at the end of the game were chosen as captured stones found in the earlier section of the analysis to give the network a balanced training set. Stones on the board that were alive at the end of the game were assigned an unconditionally alive value of 1 for the output vector of the training pair. The analysis phase then stepped backwards through the game, linearly reducing the safety estimate until the stone was placed on the board.

Initial training results were interesting but due to time constraints development did not proceed further in this area. On reflection some obvious flaws in the network design became apparent. There was no accounting for conditional life and death, the fate of the stones being considered may have been dependent on which colour was next to play. Also, when training on dead status examples, stepping backwards through the game and linearly increasing the output vector to reach 1 when the stone was first placed could have given the network conflicting training information. A better approach would have been to only create another training pattern if a stone was added or removed from the area covered by the networks input field, e.g. the 9x9 area centred on the dead stone. This would avoid dozens of training examples being generated with the same input state. Equally this could be applied to the positive training examples.

5.3.4 Urgency Networks

Urgency is an important factor to consider when choosing the best global move. Another factor that must be considered is the size of the move, referring to the Go concept of big and small moves, which, roughly speaking, relates the size of a move to how much apparent profit can be made from it. To complicate matters the size of a move isn't necessarily related to the urgency of playing it. For instance a big move may help to build a loose territorial framework in the opening but there may be a smaller move in a local situation, perhaps to determine the life and death of a key group of stones, which is far more urgent and ultimately bigger. The most urgent moves will not always be obviously connected to a large territory securing or reducing sequence, at least not until later in the game when the moves worth becomes apparent.

The urgency network was developed to cope with the problem of having move plausibility scores that related specifically to the board state and not to each other - therefore lacking any sense of whether one move should be played before another. A network architecture that made the move scores relative to each other would be much more useful and accurate for the task of ordering them.

To be able to adjust the scores so the network could output a relative urgency value required the network to consider at least two moves at a time. A goal was set to train a network that could determine whether one move was more urgent than another. In actual play a sorting

algorithm could be used to order the full set of available legal moves at a given position when used in conjunction with the urgency network.

Two designs were created, the first of which used the entire 19x19 board as input, introducing two additional neurons per board point to the S4D18 input schema and dropping the redundant *offboard* state neurons. The additional neurons were labelled MOVE A and MOVE B, each to be activated for only a single, exclusive board point, to indicate the locations of the moves to be compared. Two output neurons were specified to allow the network to output a relative score for each proposed move. This allowed the program using the network to compare the moves relative to each other. If the scores were not different enough, i.e. both were less than 0.5 or greater than or equal to 0.5, then no confidence in the quality of the relative comparison was assumed.

The second design took advantage of the previous move scoring network experiments and attempted to use some of the trained weights from those networks that had proven successful. It used the same input schema as S4D18 and so could use a ready trained set of weights from any network that used the same schema. An additional 2 hidden layers of neurons were added as adaptable weights and the original first two layers taken from the ready trained networks were frozen, with the aim of allowing the network to assimilate information about relative scoring in the additional two layers.

The training of the first design iterated through the training games taking two consecutive moves of the same colour at each board position and randomly presenting the first move as MOVE A or MOVE B. This was done so that the network would avoid learning a bias towards either MOVE A or MOVE B representing the more urgent move. The earlier played move was given an output value of 0.9, the later move, 0.1.

The training of the second design used a reinforcement scheme to adapt the weights. Again two consecutive same colour moves were scored and compared at each ply in the training games. This time if the earlier move failed to score at least 0.2 better than the later move a reinforcement factor of +0.2 was added to the output score and used as the new output training vector for the earlier move. The output score for the later move then had a reinforcement of –0.2 added to its score and this was used as the later moves output training vector.

Some initial results were gathered from these designs. The first design stabilised quickly with a poor performance score, suggesting that perhaps the network had only learned that the more urgent move was MOVE A 50% of the time and MOVE B the rest of the time. The second design also stabilised very quickly, the extra hidden layers adapting to the ready trained weights fast, however the performance never bettered that of the original source network indicating that nothing new was learnt and that the network had merely adapted around the source weights. There is the possibility that the reinforcement signal was not strong enough to affect the weights, if the assumption is made that the network had learnt only a coarse division between move classes.

5.3.5 Competitive Networks

An attempt was made to expand previous work conducted by another researcher into hybrid competitive ANN training [Cook 1999]. The training process involved a period of unsupervised training of the weights from the input to the hidden layer to allow the hidden nodes to extract clusters from the training data, in a similar way to Radial Basis Function networks [Callan 1999], and to then follow this with a period of supervised training for the hidden to output layer weights. After the unsupervised period the modified weights were frozen before moving onto the supervised training.

This hybrid method was applied to train the move scoring network design S4D18 with minimal success. Training progress was complicated to monitor during the initial stage whilst the hidden to output layer weights were still random, but after the transition to the supervised period an increase in network performance was noted. However this network did not achieve the same level as the raw network design S1, trained by supervised methods alone.

5.4 Summary

This chapter has detailed the application of ANN techniques to the game of Go. It began with a discussion on how to apply domain knowledge in an ANN and how an ANN might best be represented to effectively capture it. The core idea was to use the ANN as a move evaluating function where it could be given a board position surrounding a potential move and output a

value, which could be used to order a set of moves, typically to enhance a Minimax based search algorithm.

Several options for the input structure of an ANN were given, including size and shape of the input field and the inclusion of domain specific features such as liberty counts, in an effort to find the most efficient way to capture as much information as possible from a Go board position. Careful consideration was given to the encoding of board point states and a selection of options were presented for use in experiments, the results of which are in the following chapter.

This chapter concluded with a brief discussion of alternative uses of ANN techniques and the problems associated with them, which had been investigated in parallel to the move evaluation network.

Artificial Neural Network Results

6.1 Introduction

This chapter presents results and a discussion of the experiments with Artificial Neural Network (ANN) techniques made in an attempt to automatically learn some useful domain knowledge for the game of Go. These results have been derived from work involving the designs presented in the previous chapter. This chapter is divided into two sections; firstly the process by which the ANNs were trained and the process followed to find the optimum training conditions, and secondly, using the ANNs in the problem domain and measuring the performance improvement gained through Machine Learning.

6.2 Training the ANN

Unless otherwise indicated the conditions for training were kept constant throughout the experiments and were as specified here. Some initial experiments were run to determine reasonable parameters for training and from this the basic settings were selected as follows: learning rate 0.25, momentum 0.7, backpropagation with incremental weight update, 40 hidden neurons, 1 hidden layer, square input field (rather than diamond shaped), 9x9 input field size and S4D18 as the default input scheme. For detailed results tables covering these initial investigations refer to Appendix E. Several input field sizes gave similar training performance narrowing the choice to 5x5, 7x7 or 9x9, the largest of which was chosen to allow the network to have as much information available as possible. Whilst this, and the choice of a square input field over diamond, resulted in a larger number of neurons in the network it was felt that a larger field size would best complement the game tree search. One supporting argument for this choice is that a line of play consisting of a sequence of moves would reasonably be expected to occupy points on the board in such a way that moves occurring further from the origin of the line of play lie deeper in the sequence. To discover the consequences of stones at these points would therefore require a larger search time and it would be useful if the effects could be anticipated by the ANN. This can be illustrated by considering a local position where the fate of a string of stones depends on the position of a

friendly stone forming what is known as a ladder breaker. The larger the input field, the more often such stones will be taken into account for local positions.

The training data was taken from the No Name Go Server archive, a public Internet server for playing Go [NNGS]. Nearly all games on NNGS are played between human players however a small proportion may have been played between computer programs set up to access the server and play automatically. A total of 395,972 game records in SGF format [SGF 1997] played between the years 1995 and 2003 were gathered from NNGS and the games were filtered so that only games played between ranked players in the range 25 Kyu to 5 Dan were used. Given an average game length of 200 moves, this left a total of around 80 million individual moves. Two training pairs were generated for each move in each game by selecting the move actually played as a positive example and randomly selecting a legal move, excluding the actual move, as a negative example. This produced a total of around 160 million training patterns.

For the final assessment of the trained networks 200 professional games were randomly selected from the Nihon Ki-in Championship and the Tengen, Meijin and Honinbo tournaments. To test the quality of the network it was presented with each of the 43,760 positions in this 200 game test set from which the ANN scores of all legal moves were calculated and ranked in order. The rank of the move played by the professional player was recorded and this value was averaged across the entire test set. To monitor training progress a small validation set was constructed consisting of 10 professional games chosen from the Nihon Ki-in Championship, the Tengen tournament and NNGS that were not in the final test set or used for training. The networks were tested with this validation set every 20,000 training patterns to determine progress.

6.2.1 Non-Repeating Training Data

The initial experiments showed that NNGS provided the network with many times more training data than was required to reach a stable configuration. This led to the use of non-repeated training sets so that the network was only presented with a particular training pair from the data set once. The networks were trained without repetition and were able to achieve a stable configuration after seeing only a fraction of the total available data. To confirm that

there was no loss from this method of working, three fixed training sets were constructed for comparison. They consisted of randomly chosen NNGS games from between 1995 and 2003, the smallest set containing 20,000 patterns (56 games), the middle sized set 100,000 patterns (269 games) and the largest set 140,000 patterns (375 games).

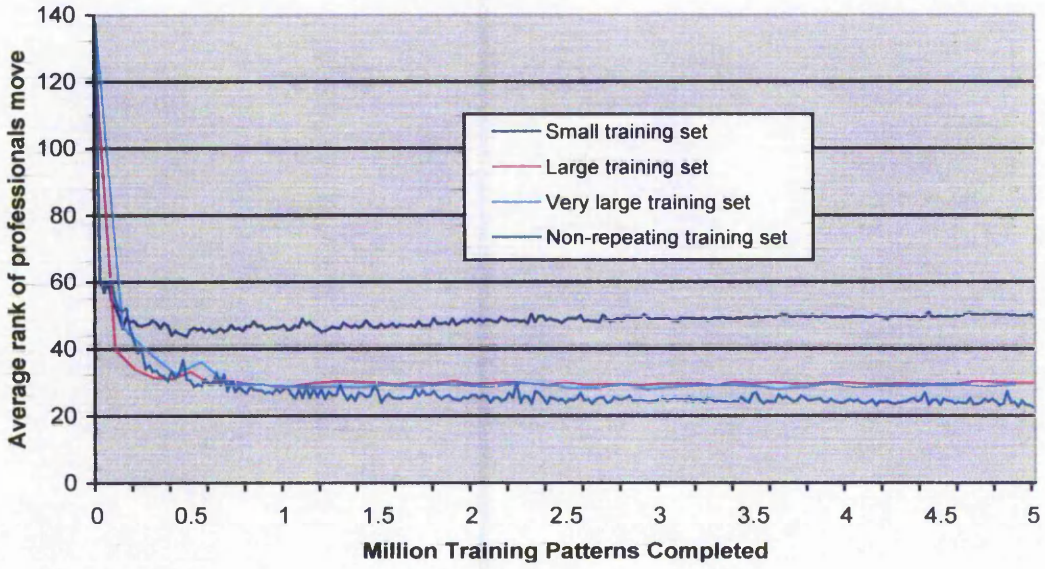


Figure1 6.1 - Comparing Training Set Size

The non-repeated training set outperformed all of the fixed size sets as can be seen from Figure 6.1. Clearly there is a strong correlation between the size of a training set and the achievable training quality. Although the non-repeated set yields less stable results from epoch to epoch, the quality difference is significant; the largest fixed size set achieves an average rank of ~30, compared to the non-repeating set's average rank of just over 20. Also of note is the slight quality reduction after about 0.5 million patterns for the small training set, indicating over-fitting. The performance increases as the training set size increases and it seems likely that it would asymptotically approach that of the non-repeating set.

There are obvious advantages of using a non-repeating training set when conditions allow. There is no need to be concerned with over-fitting, since no repetition is taking place and a specific training set need not be created, the network effectively being fed from a stream of data. There are disadvantages too, for instance training algorithms that rely on repeated set

and batch weight update operations will require special treatment and may not perform as well as expected.

6.2.2 Comparing the Designs

Figure 6.2 shows a comparison of the final quality of each ANN input scheme presented in the previous chapter and Table 6.1 lists the values for the mean rank, standard deviation and standard error for each input scheme. It could be assumed that a network that could achieve a very high rank consistently (say <4) would be able to play at a professional level without the need for any other mechanism such as game tree search. The results show that all of the networks are some way from achieving this ideal. However it is worth noting that the means are somewhat distorted by a few extreme values, where the professional move was ranked much worse than 100. This is reflected by the fact that most of the median ranks are around 11 while the means tend to be between 21 and 25. It is reasonable to assume that the extreme values occur in situations where tactical issues outweigh the usual "shape" criteria when selecting a move. This was one of the reasons for including a count of the surrounding string's liberties in S4D18Libs. Although the results do show an improvement, it is marginal. This is probably because the situations in which liberty count is significant are too rare to make much impact on the network during training.

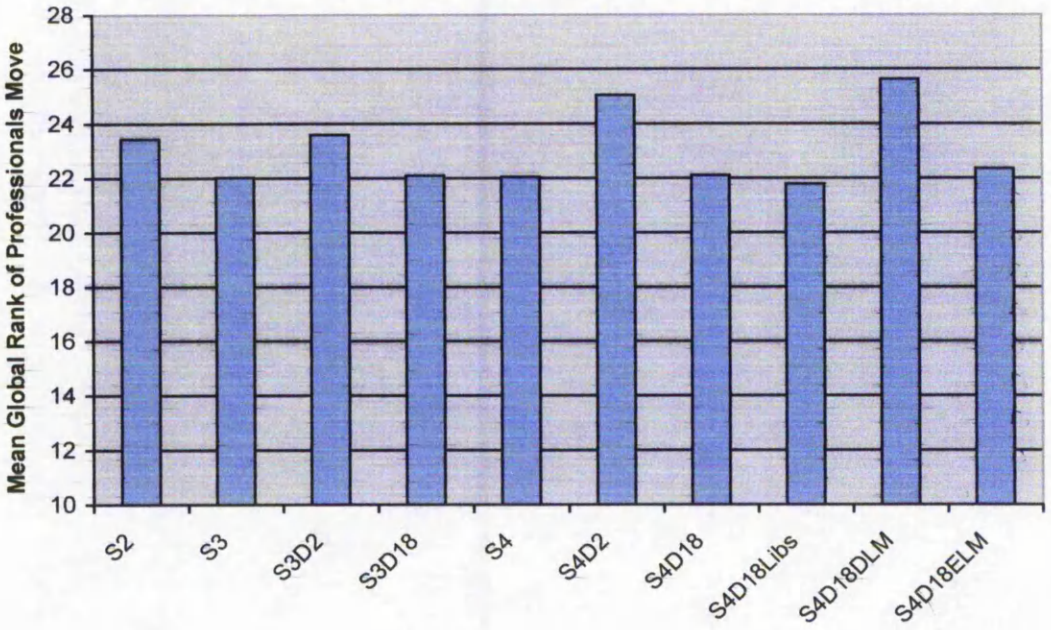


Figure 6.2 - Comparing the Final Performance of the Input Schemes

The first design, S1, has been left off the histogram to allow a clearer comparison. The final average rank achieved by the S1 network was 50.02. Considering this value compared to the rest of the designs, it is clear that moving from a single neuron per point design to using multiple state neurons per point results in a major improvement. Explicitly separating the states is a simple form of input pre-processing which would otherwise require a number of weights or possibly an entire layer of free parameters to be dedicated to this straightforward classification task. Further improvements are apparent as the various states are allocated individual neurons.

Input Scheme	Mean	Standard Deviation	Standard Error	Median	Mode
S1	50.02	47.06	0.22	36	1
S2	23.44	30.56	0.15	12	1
S3	22.02	29.64	0.14	11	1
S3D2	23.59	30.32	0.14	12	1
S3D18	22.09	29.61	0.14	11	1
S4	22.04	28.50	0.14	11	1
S4D2	25.05	32.20	0.15	13	1
S4D18	22.10	28.11	0.13	12	1
S4D18Libs	21.78	29.51	0.14	11	1
S4D18DLM	25.61	31.86	0.15	14	1
S4D18ELM	22.35	29.53	0.14	11	1

Table 6.1 - Final Global Rank of Professionals Moves

6.2.2.1 Encoding the Edges

There are several more points to note from Figure 6.2; first, all the designs are in a fairly narrow performance range. The major differences between them can be credited to the inclusion or exclusion of the distance neurons and the distance to the last move feature. Encoding the distance to the edges with only 2 neurons (those designs labelled with D2) results in a distinct performance reduction. Using 18 neurons for the distance gives similar performance to not using them at all. From this, the conclusion that the ANN had all the information it required about the board edges purely from the state neurons is reasonable. Additionally the network with only 3 state neurons (*ourcolour*, *theircolour*, *empty*) performed as well as the network with 4 neurons (additionally *offboard*). This implies that the *offboard* state neuron was redundant. A strong enough signal appears to be extracted from the training data to encode *offboard* states as simply being a void state where none of the 3 states are activated. This seems to be enough to allow the network to learn the relevance of the board edges.

6.2.2.2 Additional Features

The design with the distance to the last move encoded as an explicit feature, S4D18DLM, showed a pronounced performance drop compared to the other designs. This correlates with reports by other researchers who found a similar effect when including this feature in their own experiments [Enderton 1991].

The position of the last move seems an intuitively important feature, since it is common in Go for play to be contained to a local area until the situation has stabilised enough that one player can take a chance to gain the initiative and play elsewhere. This leads to game records containing varying length sequences of moves being played close to each other, providing the network with a strong training stimulus in favour of this feature. Additionally the network must learn to distinguish when it should stop playing in the local game and *tenuki* (a Go term meaning to play away from the local situation). Simply including the distance or position of the last move played does not supply enough information about the conditions required to *tenuki*.

Explicitly including the local move in the input scheme, without some strategic knowledge about when to change to a different local game or the importance of keeping the initiative, causes the feature to dominate the network's play, putting unwarranted emphasis on moves close to the last one played. This lack of strategic knowledge is not viewed as a problem with the design since the networks trained here and by other researchers were designed to look at a static local position without temporal knowledge or therefore any form of tactical or strategic planning. These concepts are more suited to an approach via mechanisms such as game tree search.

Another feature related to the last move played was used in design S4D18ELM in which the position of the last move is emphasised by exaggerating the input signal. This feature has a small negative affect on performance but is not as dominant as the distance to the last move.

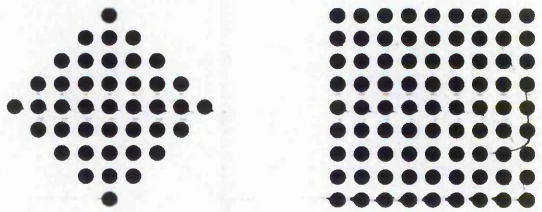
Using the liberties of the surrounding strings, as in S4D18Libs, gives a small performance improvement at the cost of the extra processing time used to generate the liberty information for the network. It is unclear whether this cost is justified but the improvement is small compared to the extra processing required such that it would be unlikely that any significant

performance improvement could be gained from further investigation into the use of this feature with the designs specified.

6.2.3 Varying the Training Parameters

As well as experiments to compare the various input schemes, adjustment of the training parameters and network architecture was also explored. The final optimum parameters found often varied from the initially established defaults. To summarise these Table 6.2 presents the optimum values for each variable. Table 6.3 summarises the results for varying the training and architecture parameters with the best results for each parameter highlighted.

Most of the variables in Table 6.3 are dependant on trial-and-error to determine their best values. The choice of input field shape is worth further discussion as it leads to some insights about the connectivity of stones on a Go board. Two field shapes were tested, square and diamond using the 9x9 field size to dictate the height and width of the shape.



41 points 81 points

Figure 6.3 - Shape Areas

The square and diamond field shapes have very close average ranks and considering that the diamond shape uses just over half of the input points that the square shape does this is a remarkable result. An improvement in training quality has been observed by other researchers when using a diamond shape of equivalent area over a square input field [Van Der Werf 2002].

Assuming stones closer to the target move have a greater influence on the move than those further away, a reason for the performance improvement may be that the diamond shape allows more of the relevant stones to be included in the input. If the Manhattan distance from

the target point to the outside points in each shape is considered, illustrated in Figure 6.3, it will be noticed that the diamond shape includes all points reachable for a Manhattan distance of *n*. The square shape includes points outside of this criterion. The improved performance with a diamond input field implies that connectivity between stones relies more on the Manhattan distance than the physical distance. This implies in turn that if the input was restricted to points within a certain Manhattan distance, the network would be able to make full use of the stone influence property; stones closer to a point have greater influence over that point. Additionally this indicates that points outside a given Manhattan distance are irrelevant unless considered alongside other points at an equal Manhattan distance. This is exemplified by the fact that in Go only stones cardinally adjacent to one another can be considered in direct contact, an obvious correlation to the use of the Manhattan distance.

Ideally an ANN could be created and trained using the values specified in Table 6.2 to yield an optimal network that performs better than any other within the bounds specified in this thesis. However, due to complex interaction between the variables it has been difficult to train a stable network to a standard comparable to the networks presented in Figure 6.2. The best performing network developed using a selection of the optimum values achieved an average professional move ranking of 31.921. The network used S4D18 with a square 7x7 input field, batch weight update, 0.01 learning rate, 0.7 momentum rate and 1 hidden layer with 40 neurons. This result indicates that the optimisation system for these parameters is frustrated and further improvement would require an exhaustive search.

Variable	Optimum Value
Input scheme	S4D18Libs
Input field size	9x9 (7x7 if reduction in processing is considered)
Input field shape	Square (Diamond if reduction in processing is considered)
Training games rank range	5k-1k
Weight update method	Batch
Learning rate	0.01
Momentum rate	0.1
Number of hidden layers	2
Number of hidden neurons	>140

Table 6.2 - Optimum ANN Parameters

Parameter	Parameter Value	Final Average Rank
Input field size	4	26.4412
	5	23.1216
	6	23.3903
	7	22.2596
	8	22.6053
	9	22.0959
	10	24.1592
Input field shape	Square	22.0959
	Diamond	22.1856
Training games rank	1d-5d	23.8196
	5k-1k	23.4672
	10k-5k	25.1546
	20k-10k	28.2212
Weight update method	Batch	20.8879
	Incremental	22.0959
Learning rate	0.001	21.8135
	0.01	19.3955
	0.15	21.0740
	0.25	22.0959
Momentum rate	0.1	21.2021
	0.3	21.4954
	0.5	21.8132
Number of hidden layers	1	22.0959
	2	20.7209
	3	20.9931
	4	22.9122
Number of hidden neurons	20	24.5925
	40	22.0959
	80	20.8547
	140	19.9058

Table 6.3 - Varying the Training and Architecture Parameters

6.3 Using Machine Learning to Solve Go Problems

To evaluate the potential benefit of coupling an ANN with a Minimax based game tree search mechanism a series of experiments were conducted to allow the comparison of the Minimax algorithm with and without the ANN. The network was available for ordering nodes in the search tree and for forward pruning. The Minimax algorithm used was a zero-window alpha-beta based search algorithm called MTD(f) [Plaat 1997]. To maintain efficiency a transposition table was implemented and MTD(f) was called from within an iterative deepening framework using an iterative step of 1. The breadth of search was controlled by using the ANN to forward prune, applying an arbitrary pruning threshold of 0.35 against which candidate moves were judged. All experiments were conducted on a system running Windows 2000 with an AMD Athlon 1.67Ghz processor and 768Mb of RAM. A finalised ANN using the default training and architecture parameters mentioned at the start of Section 6.2 was used where the configuration required it.

The configurations were tested on a set of 19 capture problems. This type of problem was chosen because it avoids the difficulties of defining a complex evaluation function, the only goal is 'has the target been captured or not?'. Of the problems used sixteen were taken from "Graded Go Problems for Beginners" [Kano 1990] and three problems were created specially for these tests. The problems used from Graded Go Problems Volume 1 were numbers 39, 87 and 88 and from Volume 2 were problems 2, 3, 7, 10, 13, 32, 35, 41, 43, 68, 70, 79 and 86. Diagrams of the custom-made problems are provided in Appendix A. Since MTD(f) is an exhaustive search algorithm it is capable of solving all the problems given sufficient time. However, given the size of the search space the amount of time required would be very large and so the algorithms were tested by varying the amount of time available and observing the success rate. In these tests the time limit was varied from 50 seconds to 400 seconds per problem.

It is reasonable to assume that any practical game tree search system would define some search boundary for a problem position as a means to focus the search and avoid investigation of irrelevant space. To this end a local search boundary was implemented for all the algorithms. To adapt to the expansion of the potential search area as greater depths in the game tree were explored, a dynamic boundary was used which expanded recursively from the

target string to encompass any strings within 2 points of any other string inside the boundary. Tests conducted without the boundary showed extremely poor performance, with only a 5% success rate.

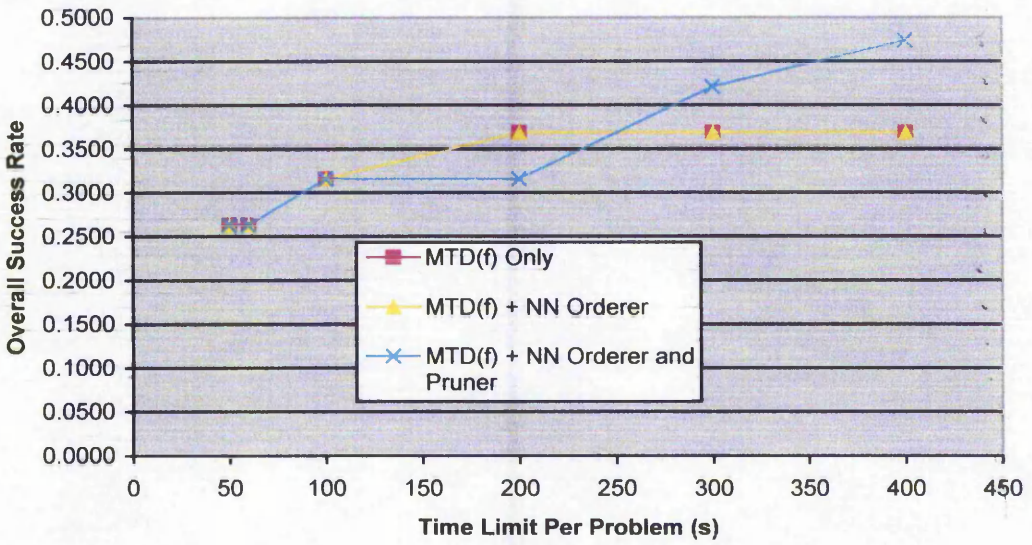


Figure 6.4 - Problem Success Versus Time Limit per Problem.

Figure 6.4 shows that using the network only to order moves seems to have little effect on performance whilst ANN pruning is ineffective for short time allowances. This is probably because of the overhead of using the network at every node, which is not compensated for by the reduction in quantity of nodes until the tree becomes larger. At 400 seconds per problem however the ANN pruning shows a significant gain. Table 6.4 shows the details. The success rate for using the network by itself is included to show that the look ahead capability provided by the tree search complements the ANN. The bracketed value in the success column shows the confidence bounds for the test set. The first figure indicates the percentage of tests that the algorithm thought it had got correct but were wrong, the second figure shows the percentage of tests the algorithm thought it had got correct and were actually correct. This means the algorithm must have proven, according to its own criteria, that the solution found was the correct one. Tests where time was overrun but the correct answer was present in the final search tree do not contribute to the bracketed figure. The improvement factor column shows the improvement over 'MTD(f) only' for each configuration and for each of the success rate, confidence bounds, time taken, nodes traversed and nodes evaluated.

Configuration	Success Rate	Time Taken(s)	Nodes Traversed	Nodes Evaluated	Improvement Factor
MTD(f) only	32% (0-32%)	5565	49,564,748	41,573,723	1.0 / 1.0 / 1.0 / 1.0
MTD(f) + ANN orderer	32% (0-26%)	6524	4,218,013	3,140,399	1.0 (0.81) / 0.85 / 11.75 / 13.24
MTD(f) + ANN orderer and pruner	47% (0-42%)	6135	3,827,756	2,807,714	1.47 (1.31) / 0.91 / 12.95 / 14.81
ANN only	26%	-	-	-	-

Table 6.4 - Enhancing MTD(f) with an ANN (400s per problem)

As can be seen from the table, using the ANN to order and prune nodes in the game tree produces the best performance for these capture problems. Compared to game tree search alone there is an improvement in success rate of nearly 50%, whilst the combined algorithm outscores the ANN on its own by an even larger margin.

The reduction in number of nodes traversed is even more substantial; with the network as an ordering mechanism and a pruner, the reduction is by over a factor of 12. For deeper searches this factor should increase as the size of the potential search tree increases exponentially. Part of the reason that the actual performance increase is comparatively small is the extra processing time used by the network at each node, however this problem only occurs because a trivial evaluation function can be used for the problems in the test set. If the search algorithm were to be used with a more sophisticated evaluation function, as would be the case for actual play, then the burden of the ANN processing would be less significant.

The proposed solution would also be useful in actual play but only if the program can recognise when it has found the correct answer. The bracketed results in Table 6.4 apply in this case. These show that there are no false positives for any of the algorithms. Using pure tree search one would not expect to see false positives, but when the tree is pruned they can occur. It is encouraging to note that this problem has not shown up during the tests that have been performed.

6.3.1 Testing in Real Play

It is difficult to test the MTD(f) - ANN combination in a complete game playing program because of the effect of the evaluation function. To be able to play a game of Go where a useful result could be obtained, rather than a complete whitewash with the maximum worst score being attained with most games, a complex and accurate evaluation function needs to be developed. Looking at the effort and man-hours that have gone into the best playing programs available it quickly becomes clear that such an effort is not practical alongside an academic investigation. Even with a reasonable evaluation the level of play that can be achieved often results in many unstable situations during a game, where the overall score varies dramatically from move to move. This can lead to extreme final game scores, making a comparison with another program unreliable when looked at over an averaged set of games. However, a set of games were run using some very simple evaluation functions to see if a performance boost could be observed when using the move scoring network.

The tests were run against the well known open source Go program GNUGo (version 3.0.0) [GNUGo]. Playing on a full size board (19x19) the results were very poor but some success was achieved on a 9x9 board. The influence evaluation function used a simple algorithm that calculated an approximation of the influence generated by stones over empty board points and from this an estimation of territory was made. To maintain simplicity and speed no account was taken of whether the stones were alive or dead. The stone count function worked as its name suggests and hence rewarded capture of opposing stones and the saving of ones own. Liberty count was similar but gave an extra weighting for stones and strings with more liberties.

All of these functions are easy to calculate but have serious and obvious shortcomings. The remaining function "move finder" is more interesting because it actually uses the ANN as an evaluation function. The concept is that if the ANN score of a move indicates how "good" it is then a position with higher average ANN scores has more "good" moves available – and hence could be regarded as a better position. The results are shown in Table 6.5 below. Some games were played against GNUGo using the game tree search and an evaluation function only (no ANN pruning or ordering) but no wins were recorded in these games and so they are not shown in the table. Further games were played with the ANN alone but no wins were

recorded during this set of tests (some wins *have* been recorded in this way but these used a slightly different ANN version and so are not directly comparable).

Search Depth	Win Rate by Evaluation Function				
	Influence	*Liberty Count*	*Move Finder*	*Stone Count*	*Average*
1	13%	0%	0%	0%	3%
2	13%	0%	0%	33%	12%
3	13%	0%	0%	0%	3%
4	7%	20%	27%	0%	13%
5	0%	7%	0%	7%	3%
6	7%	0%	7%	20%	8%
7	33%	13%	0%	40%	22%
8	20%	33%	27%	13%	23%
9	7%	7%	33%	7%	13%

Table 6.5 - Performance Against GNUGo 3.0.0

In conclusion, even with ANN pruning, the standard pattern of a game tree search algorithm plus an evaluation function is too inflexible to use in actual play. Further research needs to be conducted to investigate the use of ANNs with more flexible search techniques that could allow a variety of different evaluation criteria to be used at the same time.

6.4 Summary

This chapter presented experimental results from training a selection of ANNs with varying architectures. The method of training was based on non-repeating training data. This method compared to using a constant finite set of data was shown to produce networks that were trained to a higher standard but also highlighted the complex interactions with algorithms that rely on batch weight update operations such as RPROP.

Out of all the ANNs trained there were several features that appeared to contribute most to the training performance; single point state encoding, the use of sufficient hidden neurons and an input field of shape and size that most effectively captures the important local board state. The largest leap in training performance was made by moving from a single neuron per board point representation to a neuron per state per board point. This did have the side effect of

increasing the overall training time and processing required but dramatically improved the quality of the networks produced.

Finally the use of the trained ANNs in combination with Minimax based game tree search was shown to improve performance over a set of elementary Go problems when compared to using either Minimax or the ANN by themselves. The combination resulted in better node ordering, allowing a more efficient alpha-beta search and a large reduction in node traversal and evaluation and therefore less time spent evaluating tree nodes. This was counter-balanced by the extra processing time required by the ANNs to evaluate and order moves.

An Introduction to Genetic Algorithms

7.1 Introduction

The theory of Genetic Algorithms (GAs) is a sub-field of Evolutionary Computation which includes Evolutionary Programming, Evolution Strategies, Classifier Systems and Genetic Programming. These are all methods used to globally optimise a function and are most useful in cases where the solution space has many local optima. The information presented in this chapter has been gathered from various sources including [Heitkoetter and Beasley 2001, Kendall 2000, Russell and Norvig 1995].

John Holland presented the classic GA paradigm in his 1975 publication "Adaptation in Natural and Artificial Systems", a work widely regarded as seminal to the GA field [Holland 1975]. The field of Genetic Algorithms is loosely based on the evolutionary process observed by Darwin, where a population is driven to improve from one generation to the next due to environmental pressures and competition within the population – survival of the fittest. A population of potential solutions to a problem is maintained and evolved over time using a function to measure the fitness of population members. This allows the algorithm to select individuals, based on their fitness, to undergo processes analogous to evolutionary phenomena, such as mutation and reproduction, in an attempt to increase the average population fitness from generation to generation.

7.2 The Algorithm

This section describes a generic GA and the components that are required to construct it. The main properties of a GA are:

a) A population of potential solutions
b) Fitness function

c) Selection process
d) Crossover operator
e) Mutation operator

Using these components the algorithm below can be devised:

1. Create an initial population P_0.
2. Assess fitness of individuals in P_t, where t is the current generation.
3. Select population members to continue to next generation, P_{t+1}.
   - Elitism: Select a number of population members based on fitness.
4. Apply genetic operators to P_t to fill remaining places in P_{t+1}:
   - Crossover: Select parents based on fitness.
5. Apply mutation operator to P_{t+1} - randomly mutate zero or more population members.
6. Go to 2 and repeat until search termination criteria are achieved.

7.2.1 Fitness Function

The fitness function allows the GA to apply the principle of competitive evolution, giving some basis for judging an individual against another. Typically this is implemented as a heuristic function measuring the degree to which a potential solution solves a problem. An example could be a path-finding optimisation problem where the goal is to find the shortest solution, such as the Travelling Salesman Problem (TSP). The fitness function might return a value indicating the route length for this solution.

7.2.2 Selection Process

At several points throughout the GA a method of selecting one or more population members will be required. Various methods exist for this task, roulette wheel and tournament selection are two examples of this. Selection is also required to choose parents for the crossover operator and to choose a proportion of the next generation's members via elitism.

7.2.3 Crossover Operator

The crossover or recombination operator facilitates the combination of chromosome sections to produce new potential solutions, allowing the intermingling of genetic material within the gene pool. One of the simplest forms of recombination is referred to as one point crossover and is found by choosing a random point within a chromosome and taking the genes before and after that point from each parent to produce two children as illustrated in Figure 7.1. This idea can be extended to the concept of N-point crossover, two point crossover being another common example.

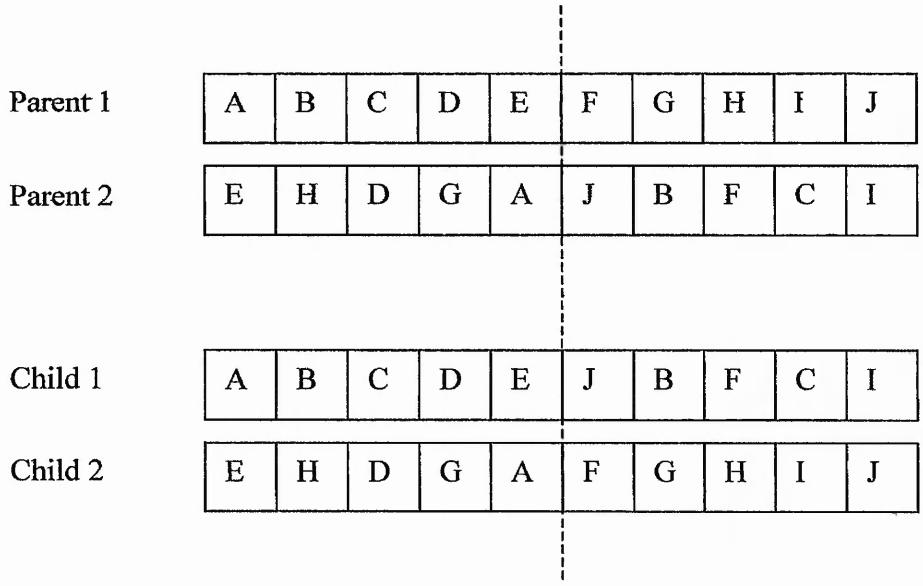


Figure 7.1 - One Point Crossover

In cases where preserving the proximity of genes to one another is not a requirement, uniform crossover may be used. This method selects child genes randomly from each.

For problems where the repetition of genes is not desirable, for instance in the Travelling Salesman Problem (TSP) where a chromosome represents a sequence of cities to visit and each city should only be visited once, more complex crossover operators must be used. Order Based Crossover is suitable for the TSP and works by partially completing a child with one parent and duplicating the order of the unused genes in the first parent to match the order that they occur in the second parent. The

gaps in the child can then be filled in from the ordered first parent. The second child is created by using the second parent as the first and repeating the process.

7.2.4 Mutation Operator

The mutation operator is essential if the initial population does not contain all the required genetic material to construct a complete, satisfactory solution to the optimisation problem. With this operator the GA can introduce new material to the gene pool, avoiding local optima that might result from a stagnant, inbred population. A small chance of mutation is usually applied to each chromosome or gene within a chromosome, a typical figure being around 0.001.

7.3 Why Use a GA?

For problems with large search spaces that would take a long time to explore using traditional methods like A* search, stochastic approaches such as GAs allow the search space to be traversed much faster, often with a result being obtained in considerably less time.

Flexibility and the potential for re-use are important benefits of a GA approach. By combining a method to encode candidate solutions into a suitable format and a fitness function to measure the success of individuals, a generic GA implementation can be produced to explore any optimisation problem.

7.4 Practical Uses for GAs

Optimising a set of parameters for a function has been a common use for GAs. An example of this might be a Chess evaluation function where the parameters represent the relative worth of the pieces. Where these values would have been hand-tuned by a programmer or chess expert, a suitably configured GA would be capable of evolving a useful set of values that would theoretically continue to approach the optimum the longer the GA was allowed to run.

7.4.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) can be thought of as functions with a number of parameters (weights) that need to optimised, therefore producing a viable set of weights for an ANN would be an appropriate task for a GA. Some systems take this a step further and include the architecture of a network as an evolvable sub-system. The SANE method does this, using a population of neurons that hold their weight values and connection specification [Moriarty and Miikkulainen 1997]. The fitness of the population is measured by creating a set of random networks from a neuron population and measuring the neuron's average performance amongst these random networks. Another example of using a GA to evolve an ANN can be seen in David Fogel's book about the Checkers program Blondie24 [Fogel 2002].

7.4.2 Intelligent Behaviour in Games

GAs have been used to generate intelligent behaviour and even to allow constructs within a game or system to adapt through their experience of the environment. One example can be found in "The Primordial Life Notebook", where an online colony of artificial life *biots* struggle for survival, their evolution against the environment guided by a GA [Spofford]. The use of ANNs and GAs in commercial games has increased in recent times as the limits of static, finite state, machine based AI has been realised – especially when the growth of massive, multiplayer online gaming is considered, where it is easy to find a challenging opponent. Use of evolutionary techniques allows the AI to adapt and learn from experience, creating a fresh and constantly challenging experience for the games player. An example of a relatively recent application of GAs and ANNs for the purposes of entertainment is the game Creatures. This was groundbreaking with its introduction of hybrid evolutionary techniques more often associated with academic research than video games.

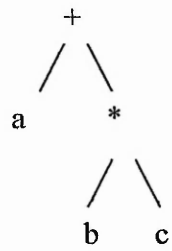
7.4.3 Path Finding

A commonly cited example of the use of GAs is their application to the Travelling Salesman Problem. This problem involves finding the optimum path to travel through

a network of connected cities with the constraint that each city must be visited only once. As mentioned earlier in this chapter, crossover operators must be used that preserve this constraint by avoiding duplicating cities in solutions.

7.5 Genetic Programming

Genetic Programming is an interesting field produced by the application of GAs to generate computer programs, where an evolving population of solutions to a problem consists of candidate programs [Koza 1992]. In this field the programs are represented as trees where the individual syntactic structures in a program are represented by tree nodes. An example tree for the program "a + b * c" is given in [Heitkoetter and Beasley 2001] as:



Genetic Programming (GP) uses the same techniques as Genetic Algorithms to evolve the solution population but special treatment must be made with crossover due to the non-linear chromosome structure of a program in GP. The crossover operator typically chooses parents by taking randomly selected subtrees from the population, biased by the individuals fitness. In a typical implementation of GP the mutation operator is not used.

7.6 Summary

GAs provide an evolutionary mechanism through which complex functions can be generated, assessed and selected giving a constantly improving pool of knowledge. This method might be compared with the process of gaining experience over time, a phenomenon which living creatures use to improve their performance in their environment and that of their offspring. The fundamental ideas of evolution upon

which GAs are modelled allow for great flexibility in the application of these ideas and this remains one of GAs most beneficial features.

There is also the potential for continuous, unsupervised improvement, a feature which AI systems in the future must include in one form or another if they are to achieve success with complex problems. This is especially true in areas where the limits of human beings are constantly tested, where intuition, experience and knowledge all play their part in a successful outcome. In these cases it is being increasingly found that megalithic software systems supported by huge expert databases cannot provide a viable solution.

Development of the Hybrid Genetic Algorithm

8.1 Introduction

This chapter details the development of a hybrid Genetic Algorithm (GA) created as an alternative to classical game tree search. In particular this is concerned with cases where the search space is too large to practically consider a high-branching tree search at a useful depth, but good search space coverage is still desired. To forge this hybrid algorithm classical GA techniques are combined with domain knowledge harvested through the use of Artificial Neural Networks (ANNs). The design and development of the algorithm is described and the various stages of development are examined. The discussion is divided into motivations, an overview of the design and a description of the phases of the algorithm. The following chapter details the experiments performed to test the configuration of the hybrid GA and compare it to a state of the art implementation of Minimax game tree search.

8.2 Motivations

The common approach to tree search problems for adversarial games such as Chess and Go, where each side is trying to maximise their score whilst minimising the opponents, is to use a Minimax based search algorithm [Plaat 1996]. Typically Alpha-Beta pruning is used to reduce the size of the tree and various heuristic enhancements are employed and tailored to the problem domain to improve the efficiency of the search. This can be an effective method to tackle tree search problems, however its use is limited for large search spaces, where the practicality of a full tree search diminishes as the size of the search space grows.

The algorithm presented here was designed to overcome this search space issue and at the same time to better simulate a human player's move selection process. The deterministic nature of Minimax search leads to a somewhat misrepresentative

scheme for tactical search. This hybrid GA provides a more flexible and intuitive approach, which is shown to be a viable alternative to Minimax for a restricted class of Go problems, where large search space tree searches are often utilised.

When faced with a large search space and especially in time critical situations such as playing a game against the clock, Minimax can become overwhelmed by the number of options available, most of which fail to result in any new or useful information. A GA in comparison could traverse the search space much faster than a tree search would be able to, however GAs are not guaranteed to find the answer. GAs work by evolving potential solutions until the population converges on the answer. The time taken to reach this convergence will be dependent on the problem complexity and the GA configuration. A trade off between search space coverage and accuracy must always be made in cases where the time available limits the search capabilities.

The hybrid GA emulates what might be thought of as a loose tree search. It effectively performs an intuitive tree search using knowledge gained from initially random explorations of candidate solutions. This knowledge guides future investigation of interesting branches, along with domain specific knowledge introduced through an ANN module as described in earlier chapters of this thesis.

8.3 Overview

From a broad perspective the hybrid algorithm can be seen as a traditional GA. It manipulates a population of solutions using mutation to introduce new genetic material to the gene pool and a crossover operator is defined to allow the recombination of genes within the population. For the algorithm described in this thesis a chromosome represents a potential sequence of moves or line of play, and genes within the chromosome represent individual moves. A goal specific to the class of problem being investigated is provided to the algorithm for the purpose of assessing fitness. A move evaluating ANN is also available to bias the random selection of genes. This can be used to create new chromosomes, generate attack moves or to produce potential mutations amongst other uses.

Due to the attack/refutation basis for measuring the quality of chromosomes, they must be represented by a more complex structure than the traditional linear gene sequence. The population members in this algorithm are stored as Directed Acyclic Graphs (DAGs) to allow refutation sequences to be recorded and, in turn, attacked themselves.

The control flow of the algorithm follows the generic idea of GAs. An initial population of solutions is created, then evaluated based on the goal followed by a period of adaptation through mutation and reproduction biased by the chromosome evaluation. The algorithm repeats the evaluation and adaptation stages until some criteria, usually evaluation based, are achieved. For a detailed English description of the algorithm in a step-by-step format along with an example chromosome please refer to Appendix D.

8.3.1 The Evolutionary Core

The initial population of solutions is generated using any available goal-specific generators followed by the creation of pseudo-random chromosomes, produced using the ANN scores to bias the random choice of genes. This is done until the population limit is reached.

The heart of the algorithm takes place within a loop that repeatedly iterates over an attack phase, a repair phase and a reproduction phase, see Figure 8.1. This continues until either the time limit is reached or a solution that fulfils the specified confidence criteria is found. Most of the work that determines the direction that evolution will take, which parents to recombine, selection of a crossover point and selection of mutation candidates and sites within a chromosome, is done within the attack phase. Rather than solely relying on a measure of fitness related to the goal to assess the success of a chromosome, the algorithm tests the population by statistically analysing the robustness of a chromosome.

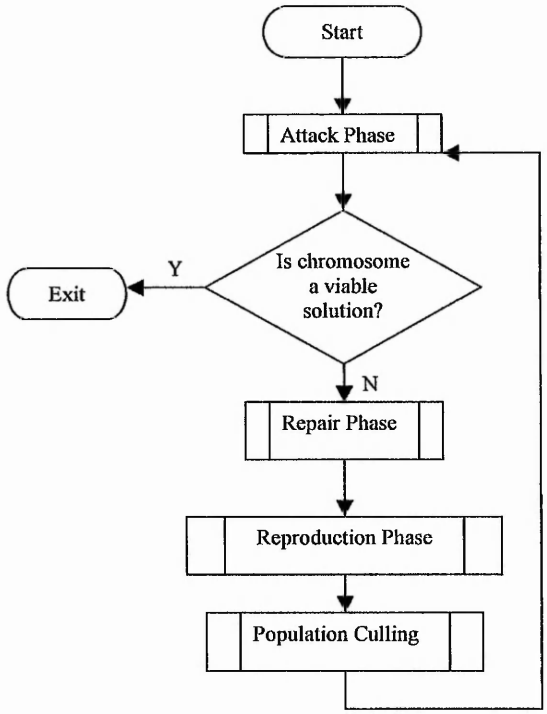


Figure 8.1 Overview of the GA Algorithm

8.4 The Hybrid Genetic Algorithm

This section describes the component phases of the algorithm as well as mechanisms for controlling chromosome size, population size and terminating the search.

8.4.1 Attack Phase

The purpose of the attack phase is to analyse a chromosome to determine hotspots where evolution may be required to fix a potential flaw. This is an iterative process - analysis is guided by information gathered during previous runs of the attack phase and so becomes more refined as the algorithm proceeds. The analysis is conducted by repeatedly attempting to insert attack moves into a chromosome and measuring the effect. This method of testing allows information to be gained about the location of weaknesses in chromosomes, which in turn can be used to direct mutation and reproduction to specifically target these flaws. This is an important advantage that the hybrid method has over traditional GAs. A measure of fitness based on goal

completion and solution length is retained for population culling purposes. Refer to Figure 8.2 for an overview of the attack phase process.

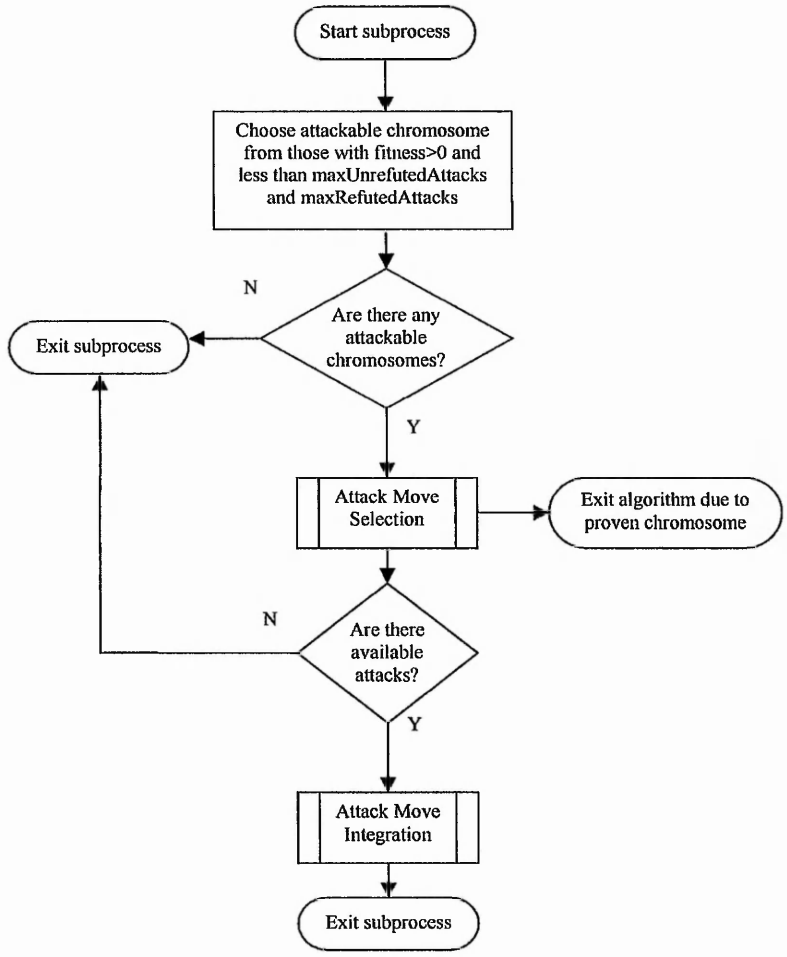


Figure 8.2 Attack Phase Overview

8.4.1.1 Finding an Attack Position

Figure 8.3 shows the process for selecting an attack position and an attack move. Each run of the attack phase causes a single chromosome to be randomly selected, biased towards those lines of play with the lowest numbers of unrefuted attacks. This favours the stronger, more robust chromosomes, allowing the algorithm to focus its efforts on those candidate solutions most likely to produce a successful line of play.

At this point each gene in a chromosome will have one of three states:

1. Never been attacked.
2. Had a number of refuted attacks.
3. Has a single unrefuted attack. May have multiple refuted attacks.

All genes within an unrefuted attack subtree of the DAG are excluded from further attack until the flaw is repaired. A position to attack within the chromosome is chosen randomly from the remaining valid genes. An alternative to selecting freely from the valid genes would be to use an additional bias towards genes with a lower number of refuted attacks to encourage attacks against those genes that haven't been tested as much as others.

8.4.1.2 Finding an Attack Move

Attack moves are selected from all available legal attacks and the random choice is again biased by the ANN evaluations. An option was included to allow expert move suggesters linked to specific goals, to influence the choice of attack using domain specific knowledge. For the experiments documented here, the capture goal used for the tests had a mechanism available called the extended liberty move suggester, which produced an ordered list of select moves based on an estimate of the value of second order liberties of the target string.

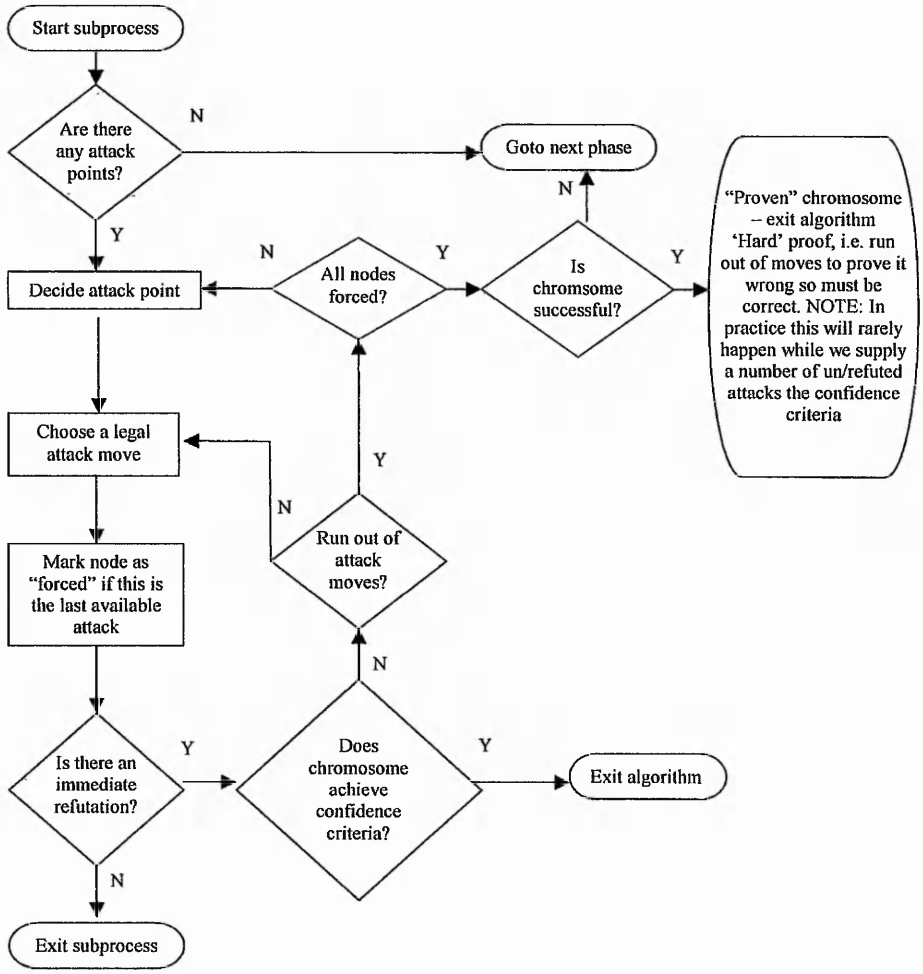


Figure 8.3 Attack Move Selection

8.4.1.3 Refuting the Attack

After the attack is inserted into a line of play a highly focussed refutation search is performed aimed at incorporating the attack into the chromosome tree and finding a refutation for it, see Figure 8.4. Figures 8.5 and 8.6 show the defender and attacker move processes respectively. The attack may be immediately refuted due to being ineffective at changing the outcome of the line of play. If this is not the case more complex potential refutations are looked at using domain specific heuristics, goal specific move suggesters to create refuting move sequences and a population search to find part-chromosomes that may refute the given attack. The refutation methods at this point are deterministic, one-shot attempts. The repair phase, which follows this attack phase, is concerned with using non-deterministic methods to refute out-standing attacks. This list states each refutation option available at this stage:

1. Ineffective attack. Makes no change to the outcome of the line of play.
2. Domain specific heuristics:
    - Mirror the enemy's move. Play an attack sibling as a defender move. The only domain specific heuristic used at this time.
3. Goal specific move suggester sequences:
    - Extended liberties linked to the capture goal.
4. Population search for part-chromosome refutations.

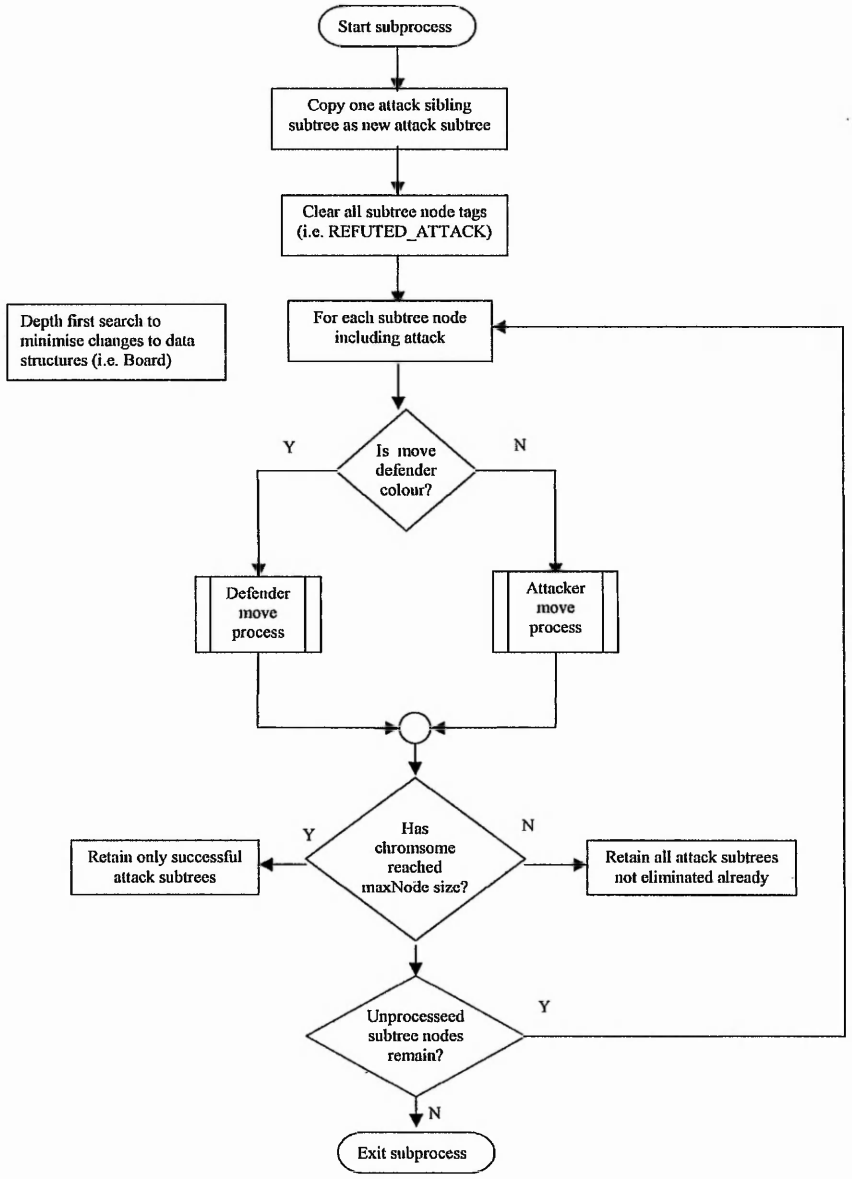


Figure 8.4 Integrating the Attack Move

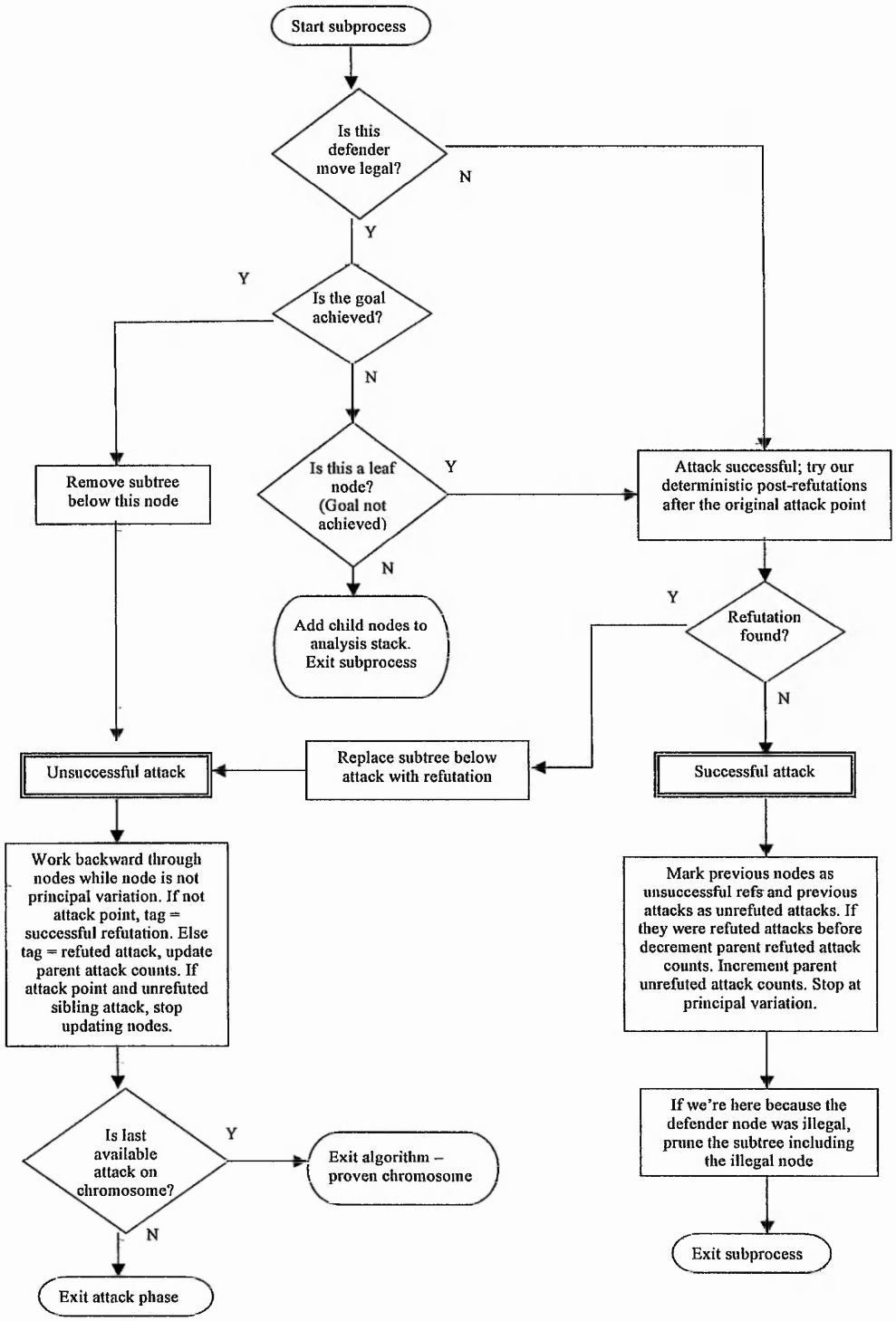


Figure 8.5 Defender Move Process

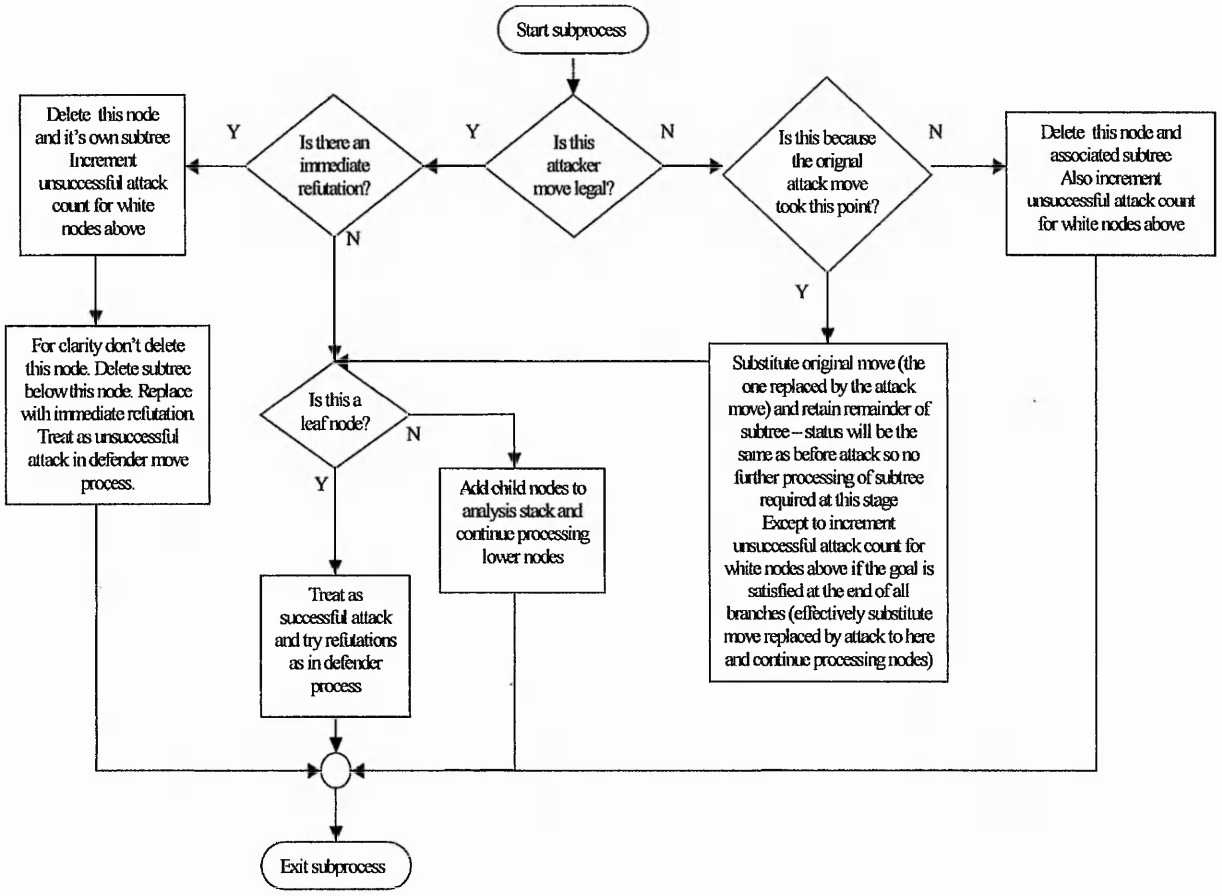


Figure 8.6 Attacker Move Process

8.4.1.4 Chromosome Size Control

A chromosome trimming process is also applied at this stage for those lines of play that grow larger than a specified size. Since refutations for attacks are stored in the chromosomes, facilitating attacks on the refutations in a future generation, the chromosomes can expand rapidly and a method of control becomes necessary.

8.4.2 Repair Phase

The next phase provides an opportunity for unrefuted attacks to be nullified after their generation in the attack phase. See Figure 8.7 for an overview of the repair phase. This may have been the current attack phase or a previous one, the repair phase makes no distinction as to when the attack was generated, it is simply aware that the attack

remains unrefuted. In contrast to the attack phase, which immediately attempts to apply deterministic refutations to refute an attack when it's created, the repair phase can use non-deterministic methods at any point in a chromosome's lifetime. Typically these refutations are based on ANN biased random move sequences and are applied using the mutation or crossover operators. One chromosome is chosen for a single repair attempt per iteration out of all the lines of play with outstanding unrefuted attacks. The chromosome is chosen randomly, biased by its number of unrefuted attacks, so that the lower the number of unrefuted attacks the more likely it will be chosen. This is to encourage chromosomes that appear to be more likely to produce a good solution to be repaired first. Since an unrefuted attack must be repaired before another can be generated at any one gene, the unrefuted attack to repair is chosen randomly without bias. The actual repair site, where modification can take place, is selected from nodes within the failed subtree whose root node is the unrefuted attack.

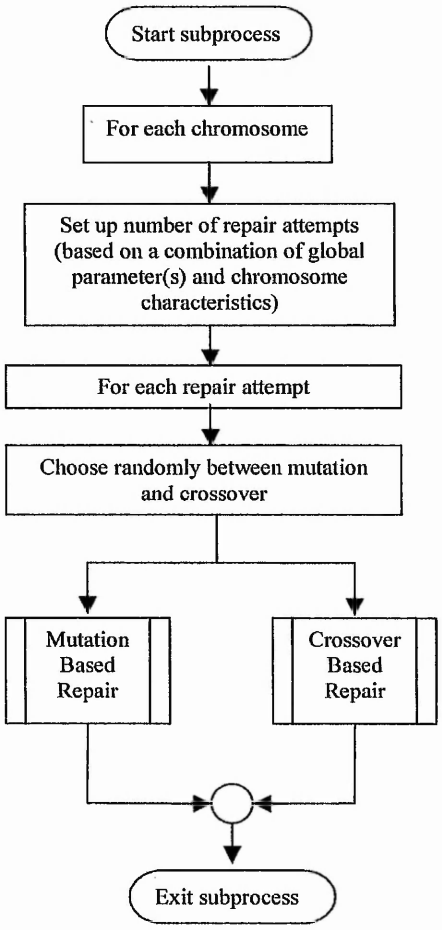


Figure 8.7 Repair Phase Overview

8.4.2.1 Repair by Mutation

Mutation works just like classical genetic mutation; a new gene is randomly selected, in this case biased by the ANN scores, and inserted into the chromosome at the repair site. Figure 8.8 and 8.9 give an overview of repair by mutation. An integration process is run to determine if the attack has been refuted by this mutation or not and if necessary to allow sequence continuations to be appended after the mutation in an attempt to complete the goal and therefore repair the chromosome. Figures 8.10, 8.11 and 8.12 elaborate on the integration process.

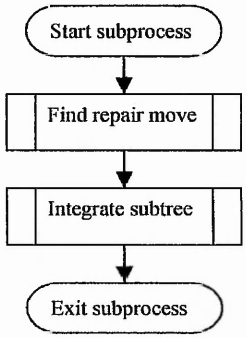


Figure 8.8 Mutation Based Repair

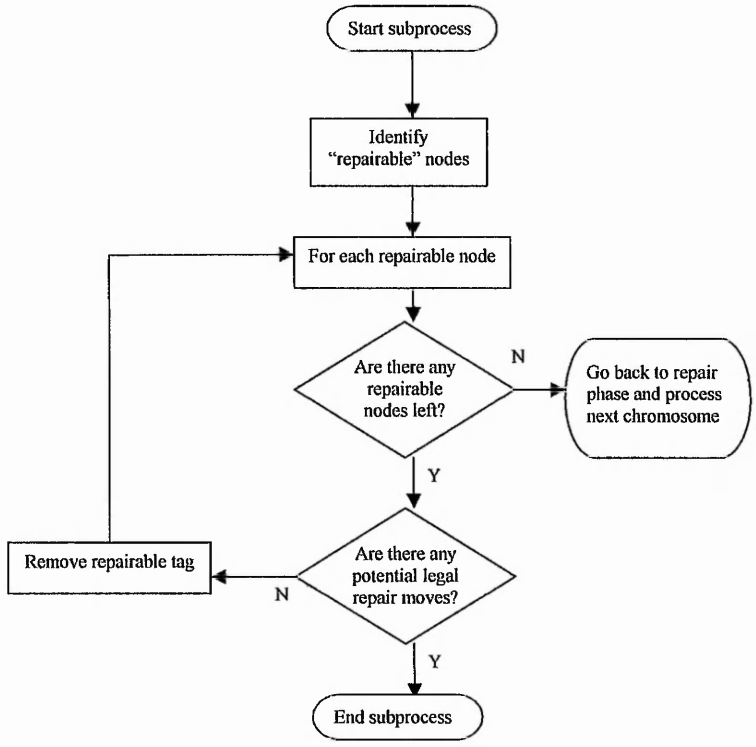


Figure 8.9 Find Repair Move

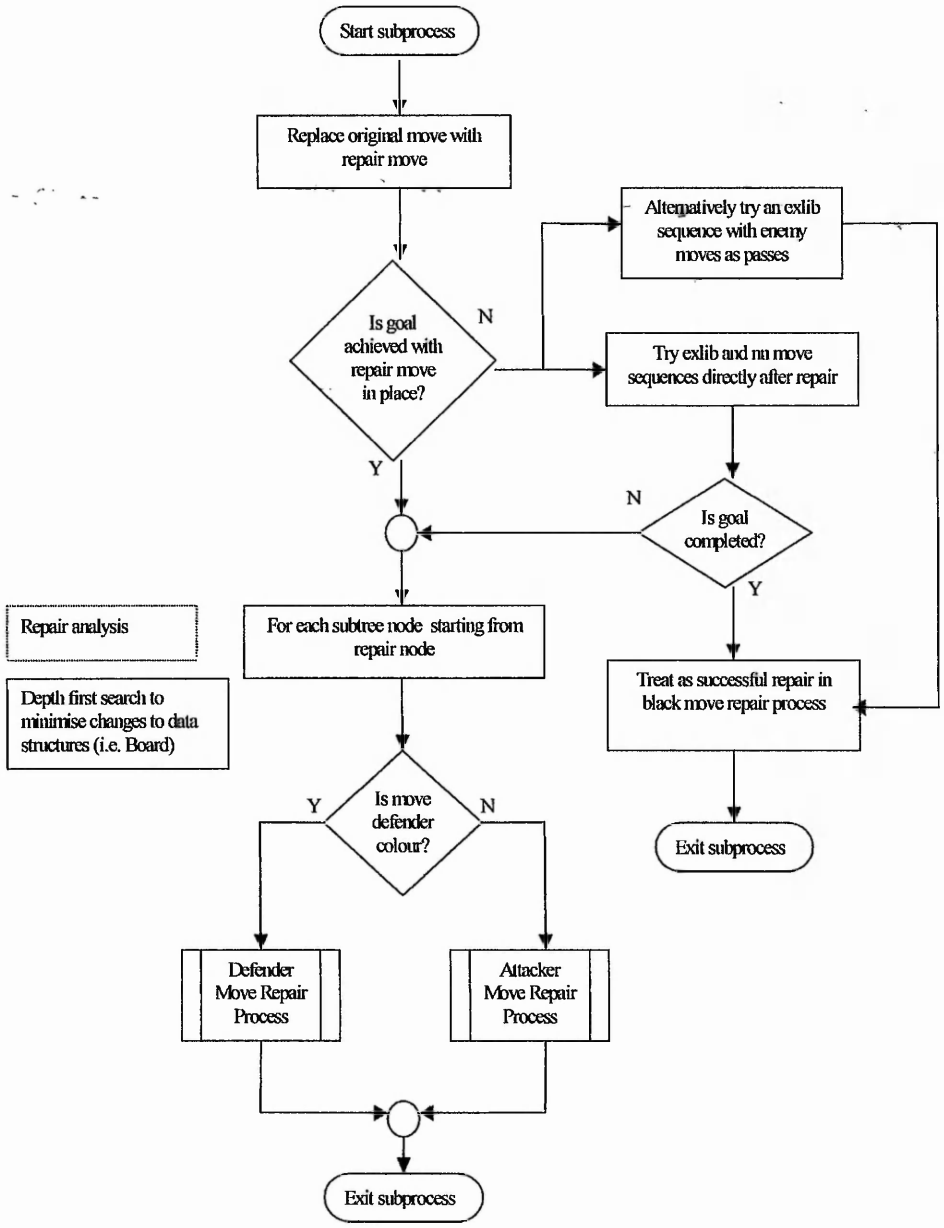


Figure 8.10 Mutation Based Repair Integration

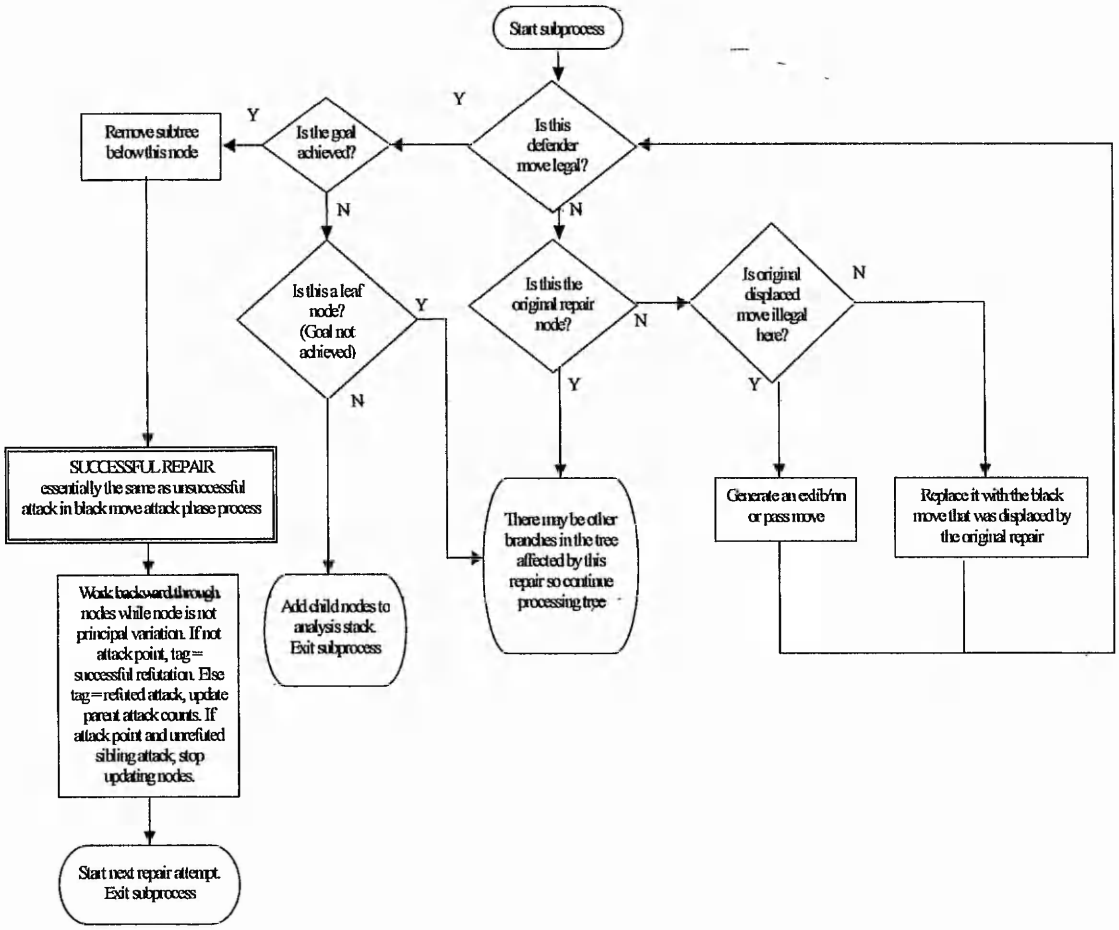


Figure 8.11 Defender Move Repair Process

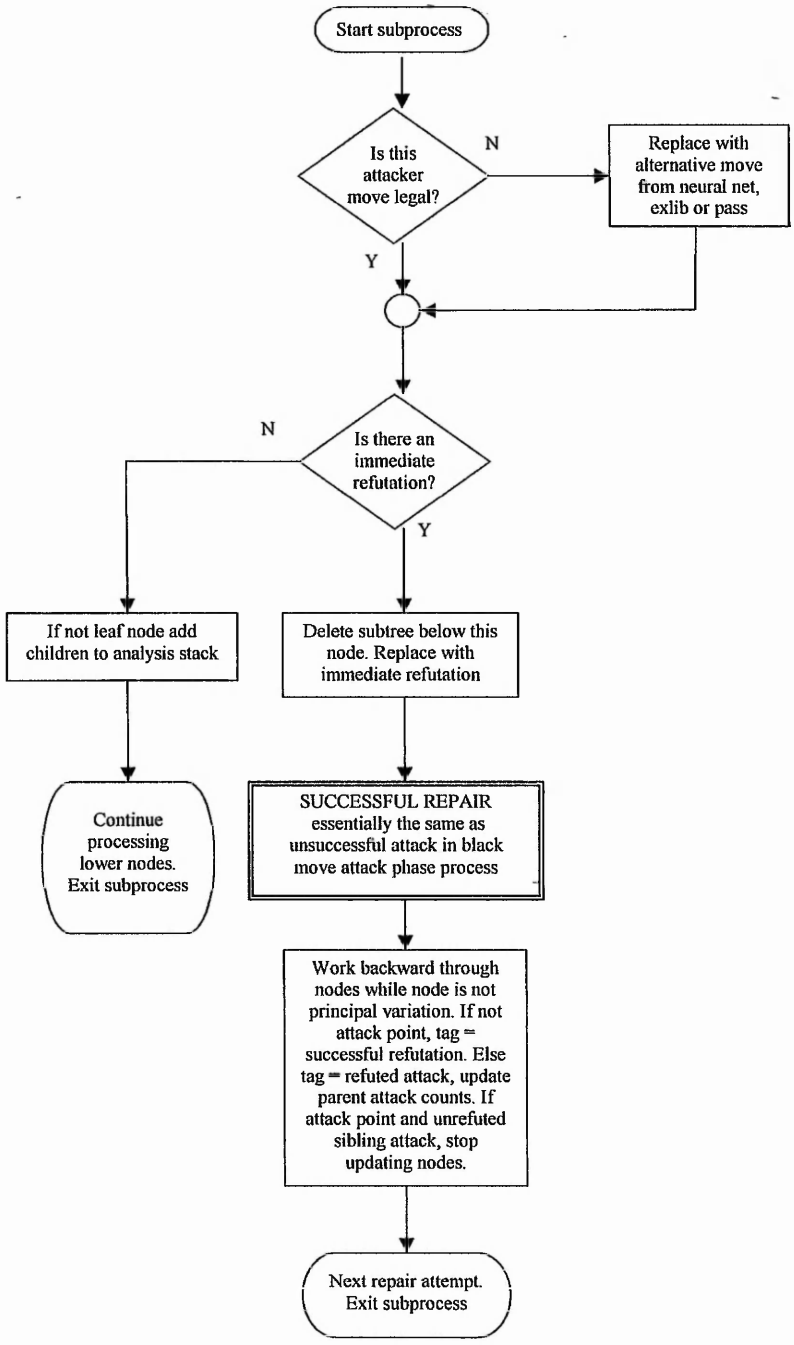


Figure 8.12 Attacker Move Repair Process

8.4.2.2 Repair by Crossover

The crossover operator should be distinguished from reproduction in this algorithm as it can be used as a mechanism of reproduction or repair. The repair phase can only modify existing chromosomes; the reproduction phase can introduce new chromosomes to the population through the mutation and crossover operators.

Crossover refers to the operation of appending a section of one chromosome onto another. For repair this simply means a population search for partial chromosomes that can refute the unrefuted attack. A search through the whole population is carried out to identify parts of chromosomes that could be attached at the repair site to refute the outstanding attack.

8.4.3 Reproduction Phase

The reproduction phase facilitates the explicit recombination of genetic material between chromosomes. Every chromosome in the population has a chance to have either mutation or crossover applied to it. If the product is a chromosome with at least one fault repaired over the original chromosome then it is added to the population. Figure 8.13 shows an overview:

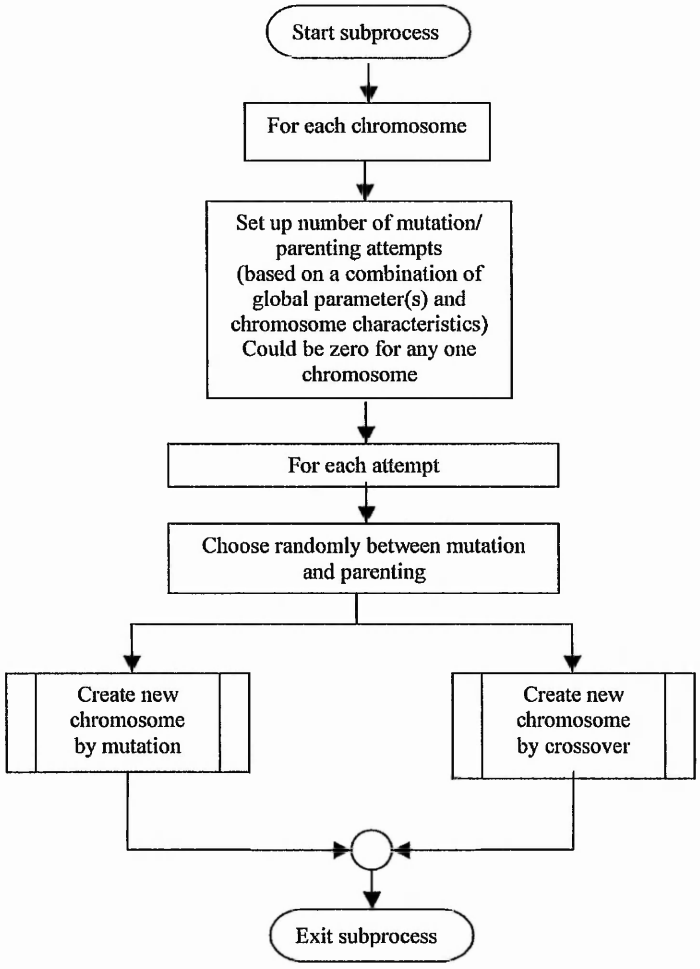


Figure 8.13 Reproduction Phase Overview

Contrary to the repair phase, each method used in the context of the reproduction phase will produce a single new chromosome to be added to the population rather than modifying an existing one. The integration of two subtrees or a mutated node into a chromosome follows the same processes as repair except for the choice of crossover/mutation site, where repair can choose from any node within a completely failed subtree. The reproduction phase is free to select its modification site from any gene in the chromosome except from a completely successful subtree. This prevents the reproduction algorithm from changing sections which currently have no flaws. There is a special case for mutation however, such that reproduction by mutation can only occur in completely failed subtrees to avoid affecting successful subtrees of a partially failed subtree. This distinction is necessary because crossover replaces entire subtrees but mutation only affects single genes.

To help show how the modification sites are chosen within the reproduction algorithm refer to Figure 8.14, which details an example chromosome. B_1 to B_7 is the principal variation with W_8 and W_{12} indicating refuted attacks and completely successful subtrees. $W_{12'}$ is an outstanding unrefuted attack and represents the root of a completely failed subtree, with W_{16} and eventually B_1 being roots of partially failed subtrees due to containing $W_{12'}$.

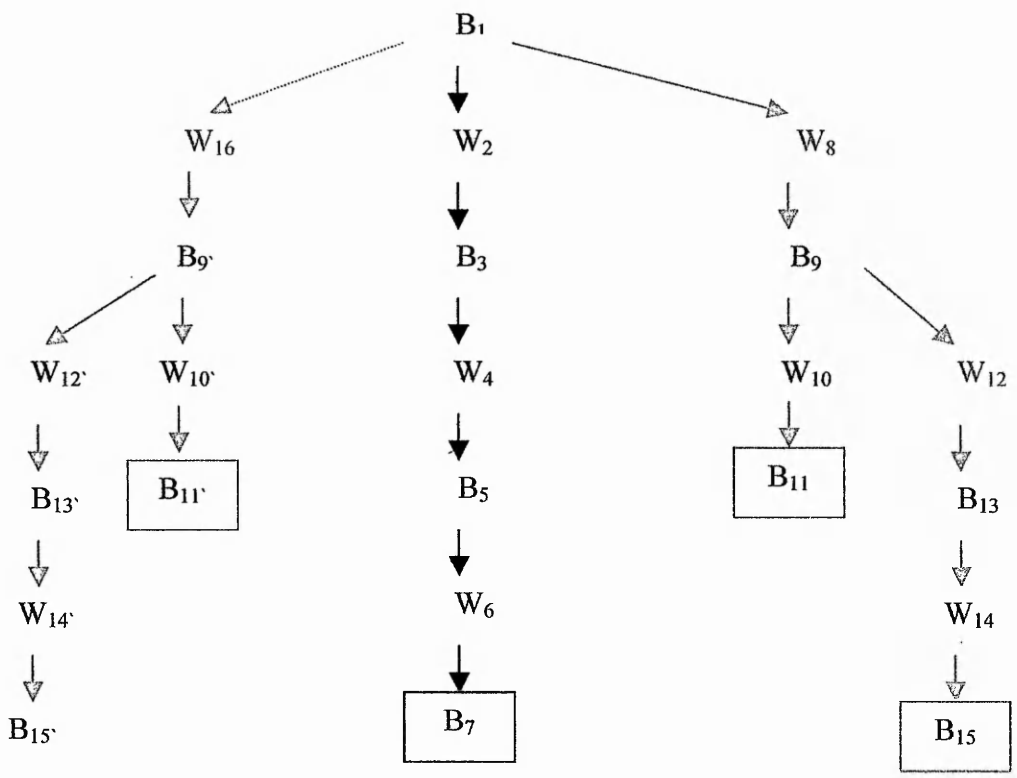


Figure 8.14 – An Example Chromosome DAG

8.4.4 Population Culling

A fitness value for each chromosome is maintained for the purposes of controlling the size of the solution population. If the maximum population size is exceeded, the fitness values are used to decide which chromosomes to remove from the population.

8.4.5 Chromosome Trimming

As a necessary result of limited resources each single chromosome was allowed to contain a maximum of 100 genes. This prevented the development machine's memory from running out during the testing and development of the GA.

8.4.6 Search Termination Conditions

The search is limited by defined confidence criteria and optionally by time. If any chromosome achieves the confidence criteria the search will terminate. The search may also be terminated if a chromosome becomes a completely proven line of play, in which case the equivalent of a complete tree search will have been performed for that chromosome. This may occur in a reasonable time if the problem solution depth is small. An entirely proven line of play consists of a chromosome that has had all potential attacks refuted at every possible point of attack within the chromosome. Usually the confidence criteria will be triggered before this could occur, unless the criteria are particularly high in relation to the problem complexity. In addition to achieving the confidence criteria a chromosome must have no currently unrefuted attacks and must complete the goal to cause the search to terminate.

The confidence criteria may be specified in three different ways as detailed below. In each case a minimum value must be defined that when reached indicates the chromosome has achieved that criteria.

1. Per Chromosome (PC) - The total number of refuted attacks occurring within the chromosome.
2. Per Depth Level (PDL) - The number of refuted attacks occurring at each depth level within the chromosome.
3. Per Node (PN) – The number of refuted attacks occurring at each gene within the chromosome.

8.5 An Example Run

Given the position shown in Figure 8.15, the GA must find a line of play to capture the white stone, with black next to move. At the start of the algorithm a selection of candidate solutions will be generated, an example of which is shown in Figure 8.15 as a capture sequence and in Figure 8.16 as a chromosome which will be used in this section to illustrate the logic of the GA algorithm.

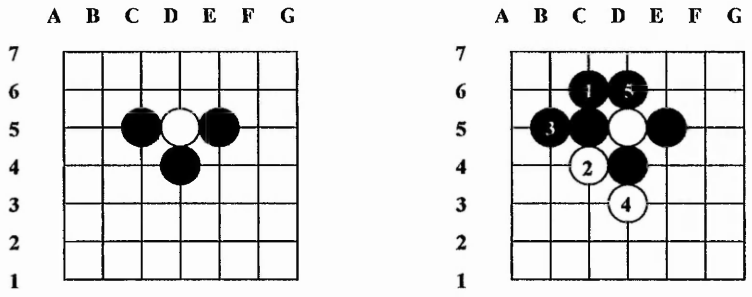


Figure 8.15 One Stone Capture / Capture Sequence

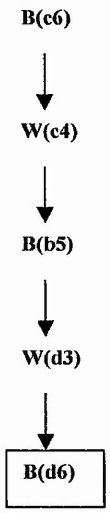


Figure 8.16 A Random Chromosome

8.5.1 The Attack Phase

At this stage the chromosomes are without branches. Let's assume this chromosome has been chosen for attack and the algorithm is about to enter the attack move selection and integration process. First of all an attack site must be randomly picked from the set of valid sites: B(c6) or B(b5). These are genes in the chromosome where an attack gene can be attached. The attacker will be white since it is white that is attempting to stop black's goal being achieved. For this example let's take B(b5) as the attack site and generate a white attack move at W(d6) as shown in figure 8.17. This may have been generated by a goal specific move generator or by a random ANN-biased choice of the available legal moves.

At this point the attack must be analysed and attempts are made to find deterministic refutations, one-shot attempts to catch this attack early. Four refutations are checked; is the attack ineffective and makes no change to the line of play, use domain specific heuristics to refute the attack, use goal specific move suggester sequences and perform a population search for partial chromosome refutations. As can be seen in the Figure 8.17 a white move at D6 will stop the black move at D6 completing the capture sequence. This attack is effective and cannot be ignored. The only domain specific move heuristic used in experiments with this algorithm was the 'mirror my enemy's move' rule. In terms of the tree structure of the line of play this means attempting to play an attack sibling as a defender move immediately after the attack. In this case the only attack sibling is W(d3). Playing this as black after W(d6) does not complete the capture so this refutation has been unsuccessful.

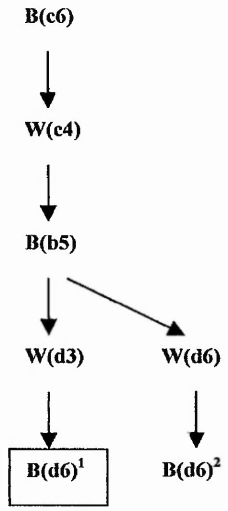


Figure 8.17 The First Attack

The next option is to try a sequence of moves generated by a goal specific move suggester such as the extended liberty generator mentioned earlier in this chapter. Due to the simple nature of this example the attack would be refuted by a sequence created in this manner as can be seen in Figure 8.18. If this hadn't worked and no partial chromosome was found in the population either we would still have the chromosome structured as in Figure 8.17. It would be noted within the chromosome that at the gene representing the move at B(b5) there was one outstanding unrefuted attack. This is important as it allows the repair phase to select chromosomes with the least amount of

work necessary to make them into viable solutions again. This attack site is now excluded from further attack until the unrefuted attack has been refuted. The site at B(c6) is still open for attack. At this point the attack phase ends and the repair phase begins.

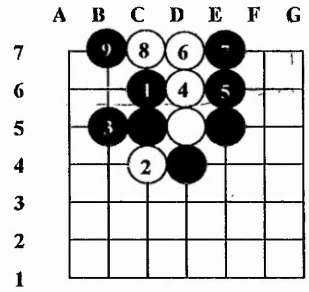


Figure 8.18 An Extended Liberty Capture Sequence

8.5.2 The Repair Phase

Since we only have a single chromosome and one unrefuted attack the choice of repair site is simple. If there where other chromosomes a random choice would be made between them biased by the total number of unrefuted attacks on each chromosome. Those with no unrefuted attacks are not eligible for repair. If there were further unrefuted attacks in the chromosome the choice between which to repair would be made randomly without bias. The repair site is also chosen at this stage from the set of defender genes in the failed subtree which has it's root node as the unrefuted attack. This only leaves one potential repair site - $B(d6)^2$.

Next a method of repair must be selected, either mutation or crossover. With mutation a new gene is randomly selected biased by ANN scores and inserted into the chromosome at the repair site. The algorithm then analyses this move to find sequence continuations that will complete the goal. Crossover is different in the fact that it searches the population for partial chromosomes that can be attached at the repair site to refute the attack. If mutation is used at this stage we may end up with a chromosome as shown in Figure 8.19 showing a successful mutation leading to an extended liberty count sequence completion rendering the attack useless.

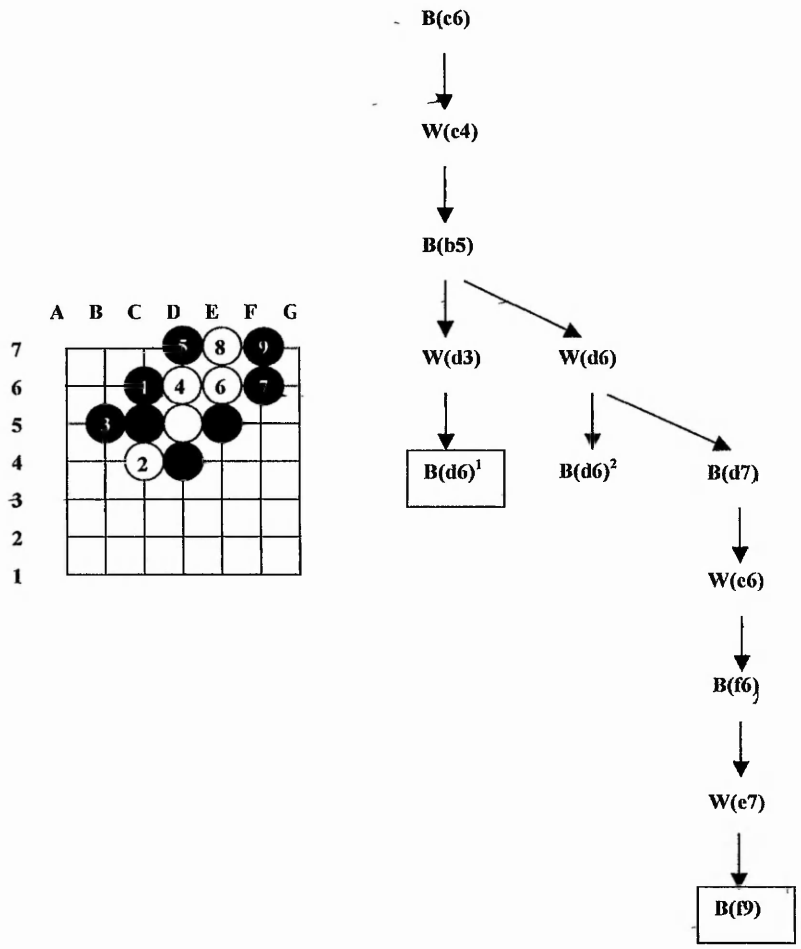


Figure 8.19 A Mutation Based Repair

8.6 Summary

This chapter has described the design of a Hybrid Genetic Algorithm formulated to handle lines of play, partial tree structures, as chromosomes and to evolve them through a process of attack. Rather than basing the evolution simply on a measure of fitness – in the case of the capture problems used here this would be a far to coarse method to produce useful solutions – the core of the algorithm revolves around a repeated process of attack to identify weak spots in the chromosome, and then repair to correct the weaknesses identified. The information found by attacking the chromosomes in this way can be used to generate crossover and mutation sites and to focus on areas that require improvement and investigation. Additionally the statistics gained from attacking can be used to estimate the quality of chromosomes and this in

turn can be used to determine when the algorithm should present a chromosome as a solution that is robust enough to return as a solution to the problem being analysed.

Hybrid Genetic Algorithm Results

9.1 Introduction

In this chapter the experiments created and performed on the hybrid GA described in the previous chapter are discussed and the results presented. Various aspects of the GA are analysed including the balance between mechanisms such as repair, reproduction and mutation, the confidence criteria measurement methods by which chromosomes are evaluated and integration with work discussed earlier in this thesis involving ANNs. This chapter then ends with some discussion contrasting the hybrid method to classic GA techniques and Minimax type game tree searches using Go problems to demonstrate the domain knowledge that has been gained by the hybrid technique.

9.2 Testing the Hybrid Algorithm

During development of the hybrid algorithm tests were run to determine the contribution of the repair and reproduction phases to the total performance. The affect of mutation and crossover within each phase was also investigated and a brief performance analysis of the search termination criteria completes this section.

All experiments were conducted on a system running Windows 2000 with an AMD Athlon 1.67Ghz processor and 768Mb of RAM. Where the configuration required it a finalised ANN using the default training and architecture parameters was used, as described in earlier chapters. All tests were performed on a set of 19 capture problems, 16 taken from Graded Go Problems for Beginners 1 and 2 [Kano 1990], and 3 problems created by the authors. The problems taken from Graded Go Problems Volume 1 were numbers 39, 87 and 88 and from Volume 2 were problems 2, 3, 7, 10, 13, 32, 35, 41, 43, 68, 70, 79 and 86. Ascii diagrams of the problems are provided in Appendix A.

Unlike the deterministic tree search algorithms, the GA required multiple runs to obtain an average due to the random element in the hybrid algorithm leading to a non-deterministic search. All GA results were averaged over 10 test runs and a 60 second time limit was allowed per problem unless otherwise specified. The maximum chromosome size was set to 100 genes. Appendix F contains some detailed results tables.

9.2.1 Assessing the Genetic Algorithm Configuration

The presence of a repair phase and a reproduction phase, each relying on the shared mechanisms of mutation and crossover, gives rise to the question of optimum configuration. Assessing various combinations of these phases and mechanisms would allow redundant processes in the algorithm to be identified. In this section results are presented from experiments comparing a selection of configurations. For these tests a confidence criteria of PN4 was chosen, which requires an acceptable solution to have at least 4 refuted attacks at each gene. This value was established from preliminary experiments.

Key	Configuration	Success Rate
All	Include all phase options (+RmcXmc)	0.7474
-Rm (+RcXmc)	Without reproduction by mutation	0.7947
-Rc (+RmXmc)	Without reproduction by crossover	0.7895
-Xm (+RmcXc)	Without repair by mutation	0.7895
-Xc (+RmcXm)	Without repair by crossover	0.7789
+Rm	Only reproduction by mutation	0.8
+Rc	Only reproduction by crossover	0.8158
+Xm	Only repair by mutation	0.6789
+Xc	Only repair by crossover	0.6632
-RX	Without reproduction or repair	0.6579
+RmXm	Only mutation for repair and reproduction	0.7842
+RcXc	Only crossover for repair and reproduction	0.8105
+X	Only repair	0.7053
+R	Only reproduction	0.7737
+RmXc	Reproduction by mutation and repair by crossover	0.8
+RcXm	Reproduction by crossover and repair by mutation	0.8316

Table 9.1 - GA Configurations and Success Rates

For presentation, the configurations have been labelled using a simple convention and are listed in Table 9.1. A minus indicates the features are not present in the configuration, a plus indicates they are the only features. An 'R' refers to the reproduction phase and an 'X' refers to the repair phase. Use of mutation and crossover within each phase is indicated by the presence of the letters 'm' or 'c' respectively. For example '-RmXc' indicates the configuration excludes reproduction with mutation and repair with crossover, which results in a configuration equivalent to '+RcXm'.

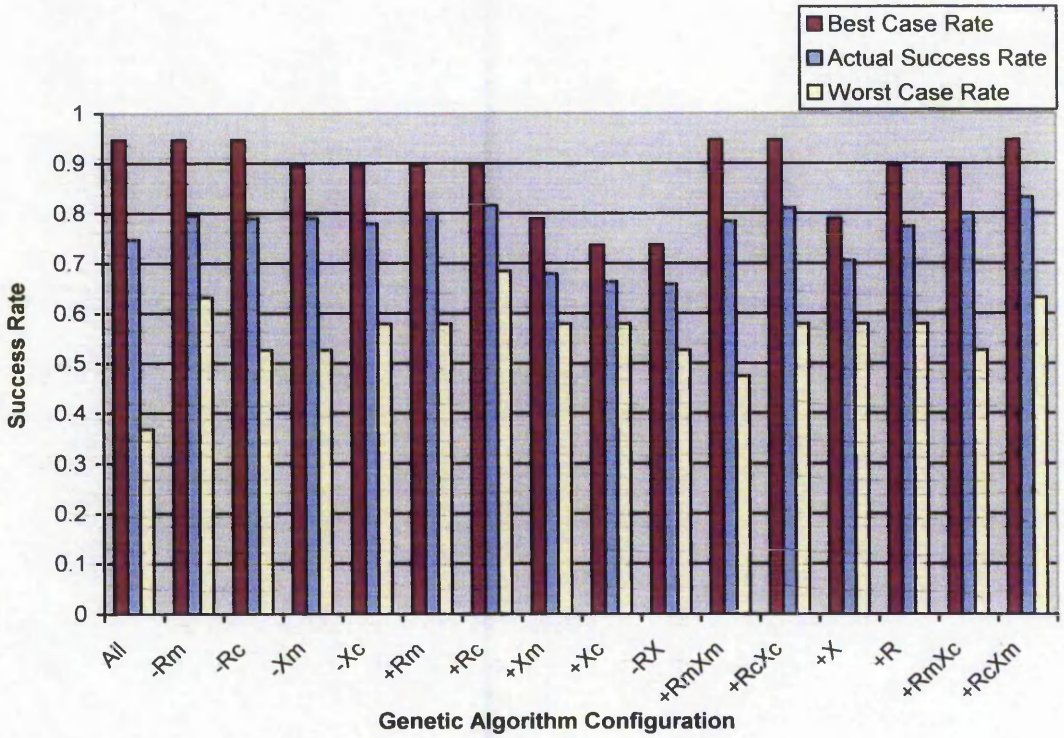


Figure 9.1 - Comparing GA Configurations

The graph in Figure 9.1 shows the best case, worst case and actual success rates for various GA configurations. The first point to note from the graph is that the configuration which includes all the phases and mechanisms (All) achieves a higher best case and actual success rate than the configuration which excludes them (-RX). This indicates that the system can increase performance through the use of one or more phase or mechanism, or some combination of them. The second point is that without reproduction or repair the interval between the best case and the worst case is much smaller showing that both mechanisms introduce elements of instability as well as positive advantages into the search. The discussion below considers the mechanisms and phases separately and is split into comparing the mutation and crossover mechanisms and comparing the reproduction and repair phases.

9.2.1.1 Mutation and Crossover

Comparing the configurations that use only mutation (+RmXm) or only crossover (+RcXc) we can observe that the confidence interval is smaller for crossover, see

Figure 9.2. If we look across the graph we can also see the confidence interval is generally smaller wherever crossover is included as opposed to mutation. This suggests that mutation introduces a greater degree of uncertainty to the algorithm than crossover.

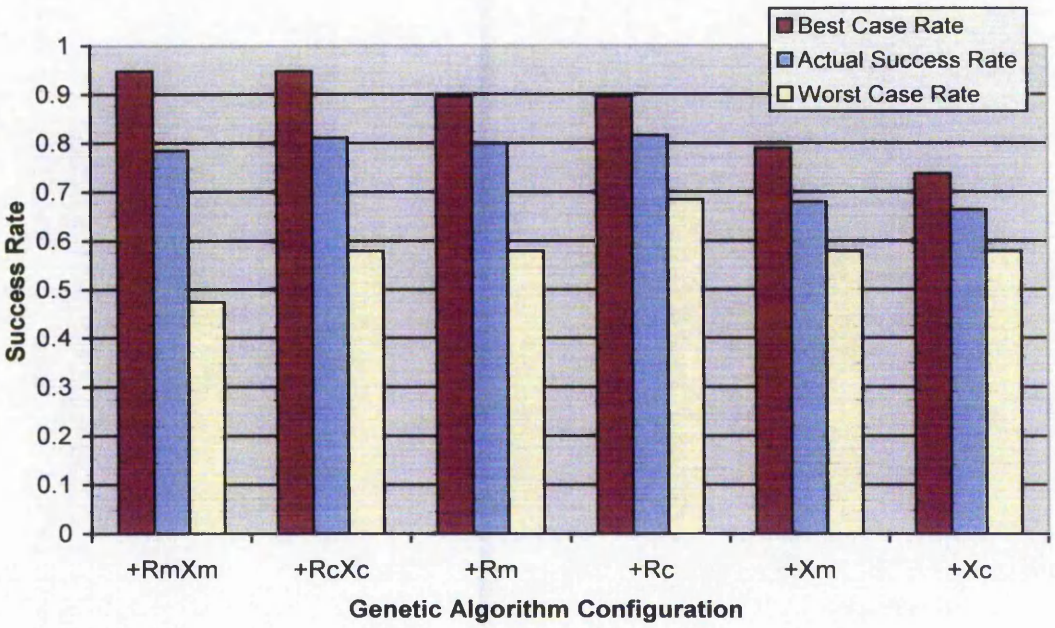


Figure 9.2 - Comparing Mutation and Crossover

9.2.1.2 Reproduction and Repair

If we now consider Figure 9.3 which shows the reproduction only (+R) and repair only (+X) columns we can observe a better success rate for reproduction over repair. The greater contribution of the reproduction phase to the success rate is also indicated by looking at reproduction with mutation only (+Rm) and with crossover only (+Rc) and comparing with the noticeably lower performing repair only configurations (+Xm and +Xc). The interval between the best and the worst case is noticeably larger for reproduction than repair in the cases where reproduction uses mutation (+Rm and +R), however the crossover mechanism coupled with reproduction seems to result in a reduction of this interval. This can be seen in the improved confidence levels for configurations that use reproduction with crossover instead of reproduction with mutation. This can also be seen in the case where reproduction with crossover is left

out (-Rc) giving a larger best to worst case interval than when it is left in (+R, -Rm, +Rc).

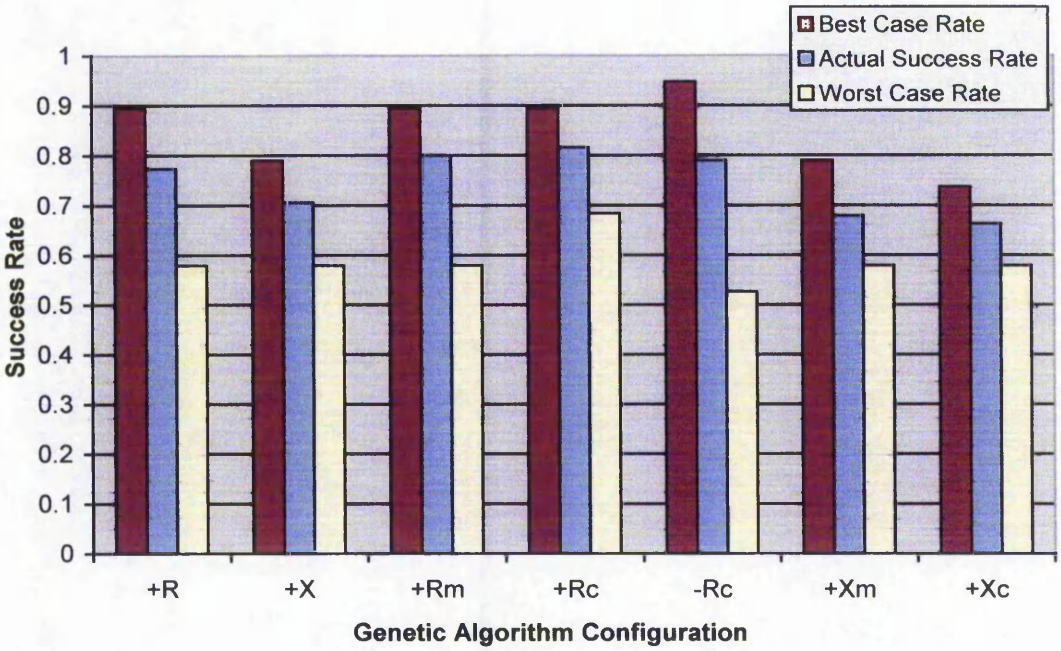


Figure 9.3 – Comparing Reproduction and Repair

9.2.1.3 Configuration Conclusions

From this analysis we can conclude that the GA performs better when both a reproduction and a repair phase are included. Crossover has been shown to be a more stable mechanism for repair and reproduction than mutation and it seems to work best when coupled with the reproduction phase. Conversely the mutation mechanism works best in conjunction with the repair phase. This is probably due to the differences between the phases; mutation in the reproduction phase will always have a low success rate but in the repair phase the chance of finding a valid repair would be much higher due to the repair phase's use of chromosome defect information. This will have been gathered during the attack phase, allowing informed targeting of an appropriate mutation site. Crossover for repair closely approaches the performance of mutation for repair, most likely due to the extra targeting information that the repair phase has available.

The far right columns in Figure 9.1 show the expected worst (+RmXp) and best (+RpXm) combinations of each phase and a single mechanism. Using the most appropriate mechanisms for each phase, crossover for reproduction and mutation for repair, we achieve the highest success rate out of all the configurations, as can also be seen in Table 9.1.

9.2.2 Comparing Confidence Criteria

To compare the confidence criteria the +RmcXmc configuration was used as described in the previous section. This makes the mechanisms of mutation and crossover available to both the reproduction and repair phases. The 3 layer confidence criterion gives the GA the option of terminating the search according to the number of refuted attacks on:

i) The whole chromosome (PC),
ii) Each depth level in the chromosome (PDL),
iii) Each node in the chromosome (PN).

To compare the performance of the hybrid algorithm using each criterion, tests were run using different values for each, including an experimental value assigning mechanism based on a primitive estimate of problem complexity. This automatic value was generated on per goal basis for each criteria above (i, ii and iii). For the capture goal used in these tests and criteria i) the automatic value was found by multiplying the target string's liberties by the total number of liberties (for both sides) in the bounded area of the problem. For the capture goal and ii) and iii) the automatic value was simply the number of liberties of the target string.

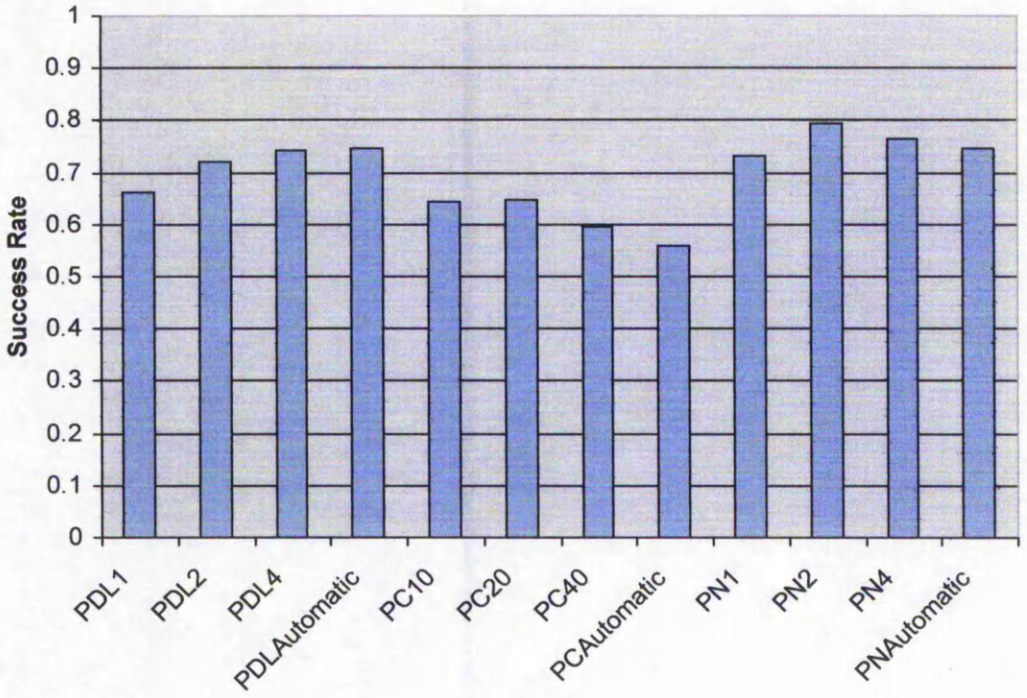


Figure 9.4 - Comparing Confidence Criteria

Figure 9.4 shows the results from the confidence criteria comparison tests. The value associated with each criterion is stated in the x-axis labels. The group of Per Chromosome (PC) tests appears to perform the worst, with Per Depth Level (PDL) and Per Node (PN) making a significant improvement over PC. This success rate increase is most likely due to the stricter requirements of PDL and PN resulting in better chromosome coverage and therefore a more reliable and accurate answer.

The success rate of the PDL criteria improves as the requirements become larger. This is probably because the PDL criteria approaches statistical emulation of the PN criteria as the value increases. From the graph it can be seen that PN2 performs the best. The actual value will be specific to the problem set and the complexity of the problems therein. Further research into generating problem specific criteria based on a measurement of refuted attacks per node would probably result in improved performance.

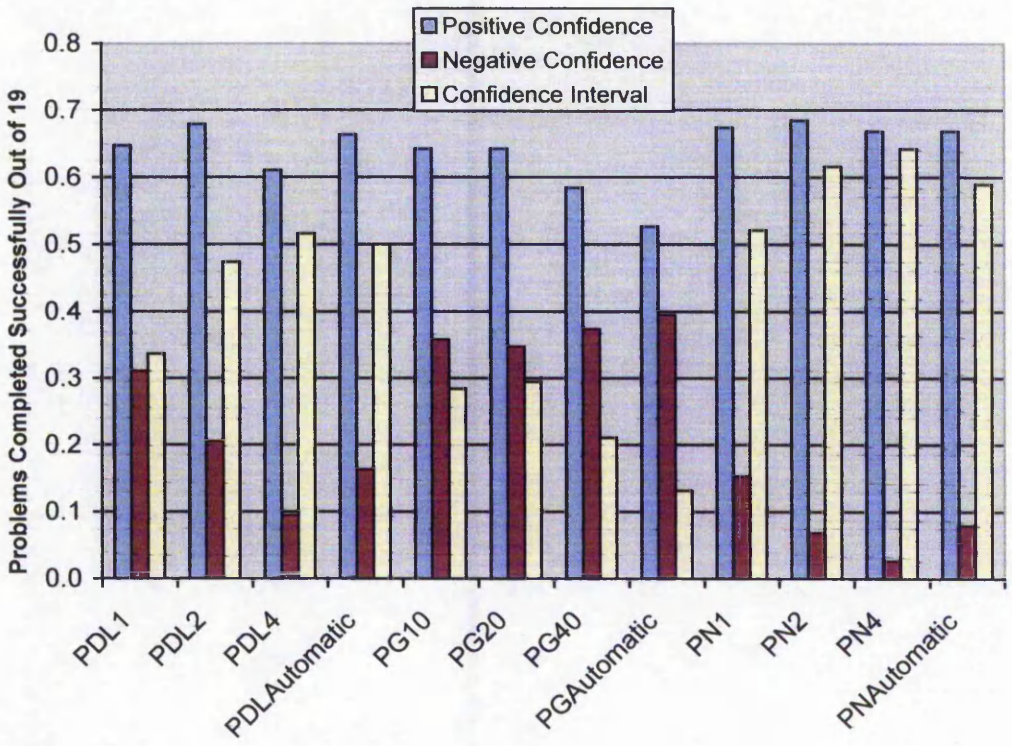


Figure 9.5 - Comparison of Confidence Levels of Morgoth Configurations

Figure 9.5 shows the calculated confidence levels for each confidence criteria option. The positive confidence figure represents the proportion of tests where the algorithm thought it had a correct answer and actually did. The negative confidence figure shows the proportion of false positives – when the algorithm thought it had the correct answer but did not. The confidence interval (the difference between positive and negative confidence levels) is also shown to highlight the stability of each option. If measured on that value then PN4 appears to be the most reliable confidence criterion.

9.2.3 Comparing To A State of the Art Minimax Implementation

To assess the GA's problem solving capabilities it was compared to a high performance Minimax based search algorithm named MTD(f) [Plaat 1996]. This Minimax variant used a zero-window alpha-beta based search, within an iterative deepening framework to maintain efficiency. The iterative depth step was set to 1 and a transposition table was used, along with best move first and enhanced transposition cut-off extensions [Plaat et al 1996]. Included in the results presented here are figures

for MTD(f) configurations that don't use the ANN and configurations that use it only as a move orderer and additionally as a forward pruning mechanism. It is important to note that these results are different from those presented in Chapter 6 due to some modifications to the tree search detailed below.

9.2.3.1 Making The Test Fair

To make the comparison as fair as possible the GA and MTD(f) searches shared certain search dependent factors. For each problem the maximum depth of search and maximum length of chromosome were the same and both methods used the same goal definition and search problem boundary. The boundary was included to restrict the search to a manageable size and it expanded recursively from the target string to encompass any strings within 2 points of any other string inside the boundary. The GA used the extended liberty mechanism to generate single attacks, choosing the highest valued, unseen, legal attack available. The GA also used this move suggester to generate a single chromosome for the initial population, using a sequence of the highest valued extended liberty moves. To balance this in the MTD(f) configurations, the extended liberty move suggester was used to promote the highest valued extended liberty move to the top of the expanded children of each node for MTD(f).

As well as the above factors the breadth of the search needed to be regulated for both methods. This was controlled in MTD(f) by the forward pruning ANN, using an arbitrary value of 0.35 as the threshold score for candidate moves to be judged by when pruning. The GA search termination conditions, or confidence criteria, were used to control the breadth of search for the GA. A PN4 (per node) criteria was chosen for these tests based on the results above, requiring at least 4 refuted attacks per gene in a successful chromosome with no other unrefuted attacks for the search to terminate. This didn't restrict the effective branching factor to 4 child nodes per gene as might be thought because the successful attacks (unrefuted ones) must also be included in the branching factor. The average branching factor for a GA search could vary dramatically depending on the direction the evolution took and must be born in mind when comparing to an MTD(f) configuration with some form of branching control, such as pruning.

Both the repair phase and the reproduction phase were enabled and the mutation and crossover mechanisms were available to each phase. Since the GA relies on a random element the results for it were averaged over 30 runs. Additionally the maximum chromosome size parameter was observed to be hindering the GA's progress for complicated problems, since some of the proofs required multiple, long refutations and these could not be stored. For this reason the maximum chromosome size was increased to 400 for these tests. Also the time limit per problem was raised to 120s to allow as much information to be gathered as possible in a reasonable amount of time.

Table 9.2 shows the performance of each search method over the test set. The bracketed value in the success column gives the confidence bounds for the tests, where the first value indicates the percentage of false positives, problems that the configurations believed were correct answers, but were actually wrong. The second figure indicates the percentage of tests the algorithm believed it had got correct and were correct. This figure only includes answers which the configurations returned due to being proven by their own criteria. This excludes unproven answers that were correct but were returned due to the time limit being reached. The improvement factor column shows the improvement over 'MTD(f) only' for each configuration and for each of the success rate, positive confidence bound, time taken, nodes traversed and nodes evaluated.

Configuration	Success Rate	Time Taken	Nodes Traversed	Nodes Evaluated	Improvement Factor
MTD(f) only	79% (0-47%)	1606	11,685,789	9,611,659	1.0 (1.0) / 1.0 / 1.0 / 1.0
MTD(f) + ANN orderer	79% (0-32%)	1971	1,599,835	1,280,488	1.0 (0.68) / 0.81 / 7.3 / 7.51
MTD(f) + ANN orderer and pruner	74% (5%-47%)	1311	791,187	541,940	0.94 (1.0) / 1.23 / 14.77 / 17.74
NN only	26%	-	-	-	-
GA – PN4 – 400 max nodes, 30 runs	85% (3%-68%)	673	445,455	400,363	1.08 (1.45) / 2.39 / 26.23 / 24.0

Table 9.2 - Comparing the GA and MTD(f) Search Algorithms (120s)

9.3 Conclusions

Looking at Table 9.2 we can see that the GA achieved the highest success rate. As well as this it did it in half the time it took the ANN enhanced MTD(f) search algorithm. The node counts are lower for the GA and the lower time and higher success rate show that the GA covers the search space faster and more efficiently. This is a common property of evolutionary algorithms, making them attractive solutions for problems with very large search spaces.

Considering the confidence bounds for the GA we see a large improvement in the upper bounds over MTD(f). This is an important statistic as it allows us to see a major difference between the algorithms. Firstly, the total percentage of tests that the GA believes it has found a suitable solution for is the sum of the upper and lower confidence bounds: 71%. Therefore problems where the time ran out and the GA was on the right path, but was not confident enough to halt the search, account for the remaining 14% of the main success rate value. Out of the problems answered with confidence by the GA (71% of all the tests) its confidence is merited 96% of the time. This implies that for time limited problems, where there is not enough time to perform a tree search to the full solution depth, the GA will perform better than MTD(f) combined with an ANN orderer and pruner.

The lower confidence bound is 0 for the first two tree search algorithms since they can only return a definite answer or unknown, although the forward pruned search has a lower confidence bound of 5%. This non-zero negative confidence bound for the configuration 'MTD(f) + ANN orderer and pruner' is unusual for tree search as it would usually be expected to produce no false results. However, using the ANN to forward prune the tree allows the opportunity for mistaken pruning to occur and has given rise to the false positive result shown here.

The GA returns a solution when its confidence criteria have been satisfied and so if the chosen criteria are not strict enough there is always a possibility of returning a wrong answer. This is reflected in the lower confidence bound of 3%. In this case the bound is reasonably low but it must be taken into consideration when applying the

algorithm. This also reflects the fact that the confidence criteria should be decided on a per problem basis to produce the optimum results.

9.4 Summary

This chapter has described a novel hybrid algorithm that has been developed using a combination of evolutionary computation methods inspired by human play and as an attempt to more closely model human Go playing strategy. It has been tested against a state of the art Minimax based game tree search implementation and found to be a viable alternative to more traditional, rigid tree search methods for capture problems.

There are drawbacks to the GA method, primarily due to the non-deterministic properties of the algorithm, which whilst allowing the algorithm to traverse the search space more quickly than a deterministic algorithm such as Minimax, also introduces an element of uncertainty such that the algorithm cannot guarantee to get the same answer at every attempt. A deterministic tree search on the other hand, assuming no forward pruning is done, will always deliver a proven solution given enough time.

Both the traditional game tree search and the GA can be easily expanded to handle different problem types, requiring only goal definitions and optionally goal-specific move suggesters to be created. Generalising the GA to other problem domains should be comparable to adapting Minimax type game tree search algorithms to new problem domains.

Conclusions

10.1 Introduction

This thesis has presented work concerning two closely linked investigations into applying Machine Learning to the game of Go. The first involved the design, implementation and testing of an Artificial Neural Network (ANN) created to assign a value to a potential Go move. This was produced with the aim of extracting knowledge about Go so that it could be used in a flexible manner to enhance game tree search. The combination of a state of the art Minimax based game tree search algorithm and the ANN, used firstly as a node ordering mechanism and secondly as a forward pruning device, resulted in better performance on a set of Go problems than using either the search algorithm or the ANN alone.

The second investigation proposed an alternative to the classical model of game tree search represented by the Minimax algorithm, from which a more human-like hybrid Genetic Algorithm (GA) was developed and shown to be a feasible alternative. As an extension of the first investigation the ANN developed was an integral part of the new GA. This facilitated the potentially powerful combination of flexible domain-specific knowledge with a dynamic, evolutionary based search method.

10.2 Contribution to Knowledge

This section discusses the main contributions to knowledge that have been developed from this research. It is split into two parts, firstly conclusions concerning the ANN investigation and secondly a discussion about the development of a novel hybrid GA.

10.2.1 Applying ANN Techniques to Go

Several ANN designs were constructed for the task of assigning a value to a move given the local situation and were described in Chapter 5, while the results of the experiments with the designs were discussed in Chapter 6.

The use of non-repeating training data, originating as a response to the excess of data available for this task, was compared to fixed size training sets and was found to allow the networks to achieve a higher quality than with limited size, repeated training sets. Use of non-repeating training data also had the advantage of avoiding the common ANN problem of over fitting since no repetition was taking place. Combining this technique with training algorithms that rely on repeated sets and batch weight update operations proved more difficult and would provide an interesting avenue of research for the future.

Various input schemes were formulated to convert the raw local board representation to the most favourable interpretation for the ANN, considering in the process pre-calculated features, such as the number of liberties of surrounding strings and the distance to the last move. The largest performance improvement over any other input scheme modification was gained by separating the possible board point states so that each was represented by a single neuron. This scheme ended up with 4 neurons per board point – *ourColour*, *theirColour*, *empty* and *offboard*. It was also found that the network was capable of implicitly encoding the *offboard* state if absent from the input scheme. This was shown through experiments to determine the importance of the target move's distance to the edges. When this factor was included as a feature in the input scheme the performance level achieved was very similar to the input schemes where it was not explicitly encoded. It is reasonable to conclude that the *offboard* state neurons were able to encode at least the same amount of relevant information about the board edges that dedicated distance neurons could. Taken a step further, when the *offboard* state was not encoded, the high performance levels were still maintained. This implies an implicit encoding of the *offboard* state created by not activating any of the other neurons. It also showed that this produced a strong enough signal to allow learning to achieve the same high standard.

As well as the input scheme variations the experiments were concerned with finding the optimum training parameters, such as learning rate, and the most appropriate architecture. A set of theoretically optimum parameters was found, however due to the complicated interactions between them training an ANN to as high a standard as the other networks used in the experiments was not possible. Amongst the parameters investigated were the input field size and shape, the weight update method – batch or incremental, the learning rate, the momentum rate, the number of hidden layers and the number of hidden neurons. The best performance settings found when looking at individual parameters were low learning and momentum rates, a square input field of size 9x9, a good human playing level for the training data (1kyu - 5kyu) and 2 hidden layers with a large number of hidden neurons.

To satisfy the first two aims of this thesis, a comparison of a state of the art implementation of the Minimax algorithm and an ANN enhanced version of the same implementation was made. The MTD(f) algorithm was used, a highly efficient, zero-window based algorithm used to drive an alpha-beta implementation [Plaat 1997]. The ANN was used in two different ways; firstly as a tree node orderer – to improve the efficiency of the alpha-beta implementation and secondly as a forward pruning mechanism. Four configurations were compared over a series of capture problems:

i) MTD(f) only
ii) MTD(f) + ANN orderer
iii) MTD(f) + ANN orderer and pruner
iv) ANN only

Configurations ii) and iii) showed a reduction in the number of tree nodes traversed and evaluated over i) by a factor of 13. The success rate of iii) over i) improved by nearly 50%. The overhead of using an ANN resulted in a 15% increase in the time taken when using the ANN enhanced search configurations, however the cost of using an ANN was shown to reduce compared to the success rate increase, as the search space grew larger. This trend was expected to continue as the search space size increased.

10.2.2 Developing a Hybrid Genetic Alternative to Minimax

In pursuit of the third aim of this thesis a hybrid Genetic Algorithm (GA) was developed with the dual goals of emulating a human player's thought process and the incorporation of domain specific knowledge into the process through the use of the ANNs developed in previous chapters. In Chapter 8 the motivations behind the hybrid GA and the design of the algorithm were described. Experiments to test the components of the algorithm and a comparison with the ANN enhanced MTD(f) Minimax implementation was made over a set of capture problems in Chapter 9.

The hybrid GA was designed with the primary purpose of acting as an alternative to classical game tree search. An adaptive, evolutionary method modelled the non-deterministic human process better than Minimax, with the added benefit of a more efficient traversal of the search space than an exhaustive tree search would allow. Additionally, the clear realisation that exhaustive tree searches are completely impractical for Go led to the investigation of an intelligent non-deterministic search method.

The GA introduced a method of measuring solution quality by relating it to the robustness of the chromosome rather than solely to the goal function. A process of repeated 'attack' upon each population member allowed the robustness measure to be calculated for individual genes within a chromosome, effectively performing a statistical sampling of the potential solutions. The identification of the robustness of particular genes facilitated the intelligent selection of genes for repair, a new mechanism using both crossover[1] and mutation to correct flaws in potential solutions.

The GA involved the use of ANNs described in previous chapters to bias the random selection of attack moves during chromosome testing, as well as producing possible repair sequences and refutations for the attacks.

---

[1] Crossover - the act of combining two different chromosomes, must be distinguished here from reproduction - which may be sexual or asexual.

As with the ANN experiments the same set of Go capture problems was used for all the GA tests. The hybrid algorithm was tested to assess the contribution of its components towards the overall success rate, looking at the affect of the mechanisms of mutation and crossover and how they performed within the repair and reproduction phases of the algorithm. As might be expected mutation was found to introduce more instability to the system than crossover, hence the common use of a low mutation rate in traditional GAs. It was also found that the crossover mechanism was most suited to the reproduction phase and mutation to the repair phase, each mechanism's properties complimenting their associated phase. For instance mutation has a low chance of generating a legal, successful chromosome but this chance is increased when used in conjunction with the statistical analysis gained from the attack phase, which the repair phase uses to direct its attention.

Finally the hybrid GA was compared to a state of the art ANN enhanced MTD(f) Minimax implementation developed in previous chapters. The GA showed considerable improvement over the ANN enhanced search, improving the success rate whilst reducing the time taken to search by nearly 50%. The non-deterministic nature of the hybrid GA does mean that this performance is not guaranteed every time, however the figures presented here were averaged over 30 runs to allow a fair comparison. The GA has been shown to be a viable alternative to traditional Minimax based tree search for the set of problems presented in this thesis.

10.2.3 Using the Techniques Outside the Problem Domain

There are several practical benefits to be noted about the work in this thesis. The ANN work brought up the question of whether repeated training sets are useful when an effectively unlimited amount of training data is available. Experiments with a non-repeating training set showed that for some circumstances when compared to a repeated training set, a non-repeated set may result in ANNs trained to a higher degree of quality. In addition the effort spent on selecting an appropriate, representative collection of data from the problem domain for the purposes of training

is minimised. These results could be transferred to any domain where an abundance of data is available.

The use of an ANN as a receptacle for domain specific knowledge and its integration into a tree search algorithm can equally be transferred to other search problems, where suitable training can be arranged.

The hybrid GA was shown to be a competitive alternative to a state of the art Minimax implementation, displaying the flexible reuse of learnt domain knowledge, in the form of an ANN, and the benefits of hybridisation to create an algorithm that exposed the most successful elements of each component. The novel use of non-deterministic statistical sampling, through a process of repeated attack, to determine the general robustness of population members and to identify weak spots within their genetic structure could be applied in more generic terms to most GA suitable problems.

10.3 Future Work

This section gives some suggestions for future work.

10.3.1 GA Meta-Goals

Any reasonable evaluation function for the game of Go, or indeed any complex system, will have multiple sub-goals that can be defined. Using Go as an example a player attempts to maximise their territory, capture opponents stones, secure their own stones, maintain connections between groups of their stones, break connections between their opponents stones and keep the initiative amongst other aims. Some of the goals listed may be a component of other goals or will be in some way related. The deconstruction of the top-level goal of Go – to win the game – is a non-trivial task, however should a method be created to implement this task then the hybrid GA described here might be as suitable a search method for meta-goals as would be a Minimax based game tree search. Further investigation into an appropriate representation for meta-goals and their internal relationships would be required but

could lead from the current state of Go programs consisting of multiple task-specific modules to creative, adaptive programs driven by their interpretation of a meta-goal.

10.3.2 Measuring the Robustness of Chromosomes in Generic GAs

The transfer of the novel GA method presented in this thesis to the classic GA domain could be achieved by reformulating the process of determining robustness and identifying weak spots in the genetic structure of population members, through non-deterministic statistical sampling as a result of repeated genetic attacks. This in turn would allow directed repair attempts using GA mechanisms such as crossover or mutation once the repair site is located. Further expansion of this idea would be necessary to realise the range of possibilities which a measure of robustness as well as the usual measure of fitness might lead to.

The combination of the robustness measure for focussing evolutionary improvements should be combined with an expansion of the confidence criteria definition and generation as a success indicator. This should include research into generating on-the-fly problem specific criteria based on a measurement of refuted attacks per node. This might be extended and contrasted with dynamic depth and width schemes for classic tree search.

This thesis has presented research using these novel ideas with Go capture problems that provide a very simple evaluation function – success or failure. Use of these methods with a scalar evaluation function, perhaps an influence based territory estimation, might lead to interesting avenues of research.

10.3.3 Performance Extensions

The following are suggestions for minor projects that could expand on the ANN training research presented in this thesis and improve the performance of the GA algorithm.

10.3.3.1 Refining the ANN Design

Work by other researchers [Van Der Werf 2002] has shown some success with alternative training algorithms such as RPROP [Reidmiller and Braun 1993] and input pre-processing techniques. Investigating the application of such techniques to the move scoring network might yield increased performance and perhaps give an insight into the complicated interactions between the non-repeated training set and batch updating, and RPROP and the momentum term. It was unclear during the experiments described in Chapter 6 why a non-repeating data set should appear to perform slightly better with batch update compared to incremental but not translate to the use of an algorithm that relies on batch update operations such as RPROP.

Three uses for ANNs were described in Chapter 5 but were not explored in detail. The first was the use of different ANNs trained for to apply to certain game stages. This leads to the issue of determining the stage in which a game currently exists, and for Go this is not a trivial problem. Moves characteristic of the end game can be played early on making it difficult to determine the global state of a game from either move numbers or move types alone. It may be beneficial to segment a board and find the local stage of sub-games, allowing specialist networks to be applied in the most appropriate areas of the board. Work by Müller and Enzenberger would be of interest concerning the division of a board position into sub-games and segmenting that position [Enzenberger 2003] and [Müller 1995].

In section 5.3.3 safety estimating networks are discussed and it is noted that some success has already been achieved by the authors of WinHonte [Dahl 1999] and Golem [Enderton 1991]. This is an area that could be closely linked to tree search to generate useful input to an ANN and is one problem that demonstrates the need to have close inter-working between the methods that determine local tactics and global strategy.

One flaw in the move evaluating network that was the main focus of research was the difficulty with which two move scores could be compared to each other. The scores related directly to the board position and the proposed move but didn't necessarily allow an accurate ordering between moves. In response to this issue Chapter 5,

section 5.3.5 discussed some modifications to the network design to allow two moves to be directly compared. The brief experiments yielded some interesting possibilities and leaves room for future development.

References

[Allis et al. 1991]
L.V. Allis, H.J. Van Der Herik and I.S. Herschberg, 1991. "Which Games Will Survive?" In *Proceedings of Heuristic Programming in Artificial Intelligence 2 - The Second Computer Olympiad*, pages 232 - 243. Ellis Horwood, ISBN 0133826155

[Allis et al 1994]
L. V. Allis, M. van der Meulen and H. J. van den Herik, 1994. "Proof-Number Search" *Artificial Intelligence*, Vol. 66, No. 1, pp 91-124. Elsevier Science Ltd., ISSN 0004-3702

[Berlekamp and Wolfe 1994]
E. Berlekamp and D. Wolfe, 1994. "Mathematical Go: Chilling Gets the Last Point", A K Peters Ltd, Wellesley, MA. ISBN 1568810326

[BGA 2004]
British Go Association (BGA), 2004. "Comparison of some Go rules" Available on the Internet at <http://www.britgo.org/rules/compare.html> [Accessed 4th of February 2005]

[Bouzy et al. 2001]
B. Bouzy and T. Cazenave, 2001. "Computer Go: An AI-oriented Survey" In *Artificial Intelligence*, Vol. 132, 2001, pp 39-103. Elsevier Science Ltd., ISSN 0004-3702

[Bouzy 2003a]
B. Bouzy, 2003. "Mathematical Morphology Applied to Computer Go" In *International Journal of Pattern Recognition and Artificial Intelligence*, Vol. 17, No. 2, March 2003, pp257-268. World Scientific Publishing, ISSN 0218-0014

[Bouzy 2003b]
B. Bouzy, 2003. "The Move Decision Strategy of Indigo" In *International Computer Games Association Journal*, Vol. 26, No. 1, March 2003, pp 14-27. ISSN 1389-6911

[Callan 1999]
R. Callan, 1999. "The Essence Of Neural Networks", Prentice Hall. ISBN 013908732

[Cazenave 2002]
T. Cazenave, "Gradual Abstract Proof Search" *International Computer Games Association Journal*, Vol. 25, No. 1, pp 3-15, 2002. ISSN 1389-6911

[Cazenave 2004]
T. Cazenave, "Golois" A Go playing program, Information available on the Internet at <http://www.ai.univ-paris8.fr/~cazenave/Gogol.html> [Accessed 1st February 2005]

[Chikun 1997]
C. Chikun, 1997. "Go: A Complete Introduction To The Game" Kiseido Publishing Company. ISBN 4906574505

[Churchill 2001]
J. Churchill, 2001. "A Neural Network Approach to the Game of Go". Dissertation completed as a requirement for the Masters of Science degree in Industrial Computing Systems, Nottingham Trent University.

[Computer Go Bulletin]
D. Erbach ed. "Computer Go Bulletin" Available on the Internet at <http://www.daogo.org/> [Accessed 1st February 2005]

[Computer Go Ladder]
Computer Go Ladder, See <http://www.cgl.ucsf.edu/go/ladder.html> [Accessed 1st February 2005]

[Computer Go Mailing List]
Computer Go Mailing List, See <http://computer-go.org/mailman/listinfo/computer-go> [Accessed 1st February 2005]

[Conway 1976]
J. H. Conway, 1976. "On Numbers and Games", Academic Press, London/New York, ISBN 0121863506.

[Cook 1999]
A. Cook, 1999. "Neural Network Evaluation of the Effectiveness of Rendering Algorithms for Real-Time 3D Computer Graphics". *PhD thesis*, Nottingham Trent University.

[Dahl 1999]
F. Dahl, 1999. "Honte, a Go-Playing Program Using Neural Nets" From *Workshop Notes: Machine Learning in Game Playing*. 16th International Conference on Machine Learning (ICML-99), Bled, Slovenia, 1999, available on the Internet at <http://www.ai.univie.ac.at/icml-99-ws-games/papers/dahl.ps.gz> [Accessed 1st February 2005]

[Dahl 2005]
F. Dahl, 2005. "WinHonte 1.03" Available on the Internet at <http://www.jellyfish-go.com/> [Accessed 6th February 2005]

[Donnelly et al. 1994]
P. Donnelly, P. Corr, D. Crookes, 1994. "Evolving Go Playing Strategy in Neural Networks" *AISB Workshop on Evolutionary Computing*, Leeds, England. Available on the Internet at <http://citeseer.ist.psu.edu/donnelly94evolving.html> [Accessed 1st February 2005]

[Enderton 1991]
H. Enderton, 1991. "The Golem Go program" *Technical Report CMU-CS-92-101*, School of Computer Science, Carnegie-Mellon University. Available on the Internet at <ftp://ftp-igs.joyjoy.net/Go/computer/golem.sh.Z> [Accessed 1st February 2005]

[Erbach 1992]
D. W. Erbach, 1992. "Computers and Go" In *The Go Players Almanac*, Chapter 11, pp 205-217, R. Bouzlich, editor, The Ishi Press, ISBN 4871870405.

[Enzenberger 1996]
M. Enzenberger, 1996. "The Integration of A Priori Knowledge into a Go Playing Neural Network" Available on the Internet at <http://www.markus-enzenberger.de/neurogo.html> [Accessed 1st February 2005]

[Enzenberger 2003]
M. Enzenberger, 2003. "Evaluation in Go by a Neural Network Using Soft Segmentation" In proceedings of *10th Advances in Computer Games*, pp 97-108. Available on the Internet at <http://www.markus-enzenberger.de/neurogo/neurogo3.pdf> [Accessed 11th June 2006]

[Enzenberger 2004]
M. Enzenberger, 2004. "Computer Go Bibliography" Available on the Internet at <http://www.markus-enzenberger.de/compgo_biblio.html> [Accessed 1st February 2005]

[Fausett 1994]
L. Fausett, 1994. "Fundamentals of Neural Networks: Architectures, Algorithms and Applications" Chapter 6, Prentice-Hall, ISBN 0-13-334186-0.

[Fogel 2002]
D. B. Fogel, 2002. "Blondie24" Morgan Kaufmann Publishers, ISBN 1-55860-783-8

[Fotland 1993]
D. Fotland, 1993. "Knowledge Representation in the Many Faces of Go" Available on the Internet at <http://www.smart-games.com/knowpap.txt> [Accessed 1st February 2005]

[Fotland 2002]
D. Fotland, 2002. "The Many Faces of Go 11.0", A computer Go program. Information available on the Internet at http://www.smart-games.com/ [Accessed 2nd February 2005]

[Fürnkranz 2003]
J. Fürnkranz, 2003. "Bibliography on Machine Learning in Strategic Game Playing" Available on the Internet at <http://www.ai.univie.ac.at/~juffi/lig/lig-bib.html> [Accessed 6th February 2005]

[GNUGo]
GNU Go latest version can be found at <http://www.gnu.org/software/gnugo/gnugo.html> [Accessed 1st February 2005]

[Hafner 1984]
J. Hafner, 1984. "Matchbox Tic-Tac-Toe", In *Antic* Vol. 3, No. 1, pp 53. Article available on the Internet at <http://www.atarimagazines.com/v3n1/matchboxttt.html> [Accessed 6th February 2005]

[Heitkoetter and Beasley 2001]
J. Heitkoetter and D. Beasley, eds. 2001. "The Hitch-Hiker's Guide to Evolutionary Computation: A list of Frequently Asked Questions (FAQ)", USENET: comp.ai.genetic. Available via anonymous FTP from rtfm.mit.edu/pub/usenet/news.answers/ai-faq/genetic/ About 110 pages. Also available on the Internet at <http://www.faqs.org/faqs/ai-faq/genetic/part1/> [Accessed 1st February 2005]

[Holland 1975]
J. Holland, 1975. "Adaptation in Natural and Artificial Systems", Ann Arbor, The University of Michigan Press, ISBN 0262581116.

[Hornik et al. 1989]
K. Hornik, M. Stinchcombe and H. White, 1989. "Multilayer Feedforward Networks are Universal Approximators" In *Neural Networks*, Vol. 2, Issue 5, pp. 359-366. Elsevier Science Ltd. ISSN 0893-6080

[IBM 1997]
IBM, 1997. "The Making of Deep Blue" Available on the Internet at <http://www.research.ibm.com/deepblue/meet/html/d.3.1.html> [Accessed 4th February 2005]

[Junghanns et al. 1995]
A. Junghanns, C. Posthoff and M. Schlosser, 1995. "Search with Fuzzy Numbers" In *Proceedings of the 4th IEEE International Conference on Fuzzy Systems*, Yokohama, Japan, 1995.

[Kano 1990]
Y. Kano, 1990. "Graded Go Problems for Beginners" Vols 1-2, The Nihon Ki-in, ISBN 4-906574-47-7 (Vol. 1), 4-906574-47-5 (Vol. 2).

[Kendall 2000]
G. Kendall, 2000. "Genetic Algorithms" Lecture notes from the AI Methods Course (G5AIM) at the University of Nottingham.

[Koza 1992]
J. R. Koza, 1992. "Genetic Programming: On the Programming of Computers by Means of Natural Selection" MIT Press, ISBN 0-262-11170-5.

[Lubberts and Miikkulainen 2001]
A. Lubberts and R. Miikkulainen, 2001. "Co-Evolving a Go-Playing Neural Network" In "Coevolution: Turning Algorithms upon Themselves", *Birds-of-a-Feather Workshop, Genetic and Evolutionary Computation Conference, GECCO 2001.*

[Meijer and Koppelaar 2003]
A. B. Meijer and H. Koppelaar, 2003. "Towards Multi-Objective Game Theory – with Application to Go" In *Proceedings of the 4th International Conference on Intelligent Games and Simulation, Game-On 2003*, Eurosis, ISBN 9-0773-8105-8.

[Minsky and Papert 1969]
M. Minsky, S. Papert, 1969, "Perceptrons: An Introduction to Computational Geometry" MIT Press, Cambridge, Mass, ISBN 0262130432.

[Moriarty and Miikkulainen 1997]
D. E. Moriarty and Risto Miikkulainen, 1997. "Forming Neural Networks Through Efficient And Adaptive Coevolution" In *Evolutionary Computation,* Vol. 5, pp.373-399, MIT Press, ISSN 1063-6560. Also available on the Internet at <http://www.cs.utexas/edu/users/nn/downloads/papers/moriarty.ec98.ps.gz> [Accessed 6th February 2005]

[Müller 1995]
M. Müller, 1995 "Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory" *PhD thesis*, ETH Zürich. Available on the Internet at <http://www.cs.ualberta.ca/~mmueller/publications.html> [Accessed 1st February 2005]

[Müller 2000]
M. Müller, 2000. "Explorer", Information relating to Explorer available on the Internet at <http://www.cs.ualberta.ca/~mmueller/cgo/explorer.html> [Accessed 2nd February 2005]

[Müller 2002]
M. Müller, 2002. "Computer Go", *Artificial Intelligence*, Vol. 134, pp. 145-179, Elsevier Science Publishers Ltd., ISSN 0004-3702.

[von Neumann and Morgenstern 1944]
J. von Neumann, O. Morgenstern, 1944. "Theory of Games and Economic Behaviour" Princeton University Press, Princeton, ISBN 0691003629.

[Newman 1994]
W. H. Newman, 1994. "Wally" A simple Go-playing program available on the Internet at <http://www.britgo.org/downloads/wally.zip> [Accessed 6th February 2005]

[NNGS]
No Name Go server, Additional information can be found at <http://nngs.cosmic.org/> [Accessed 1st February 2005]

[Plaat 1996]
A. Plaat, 1996. "Resarch Re:Search & Re-Search" *PhD thesis*, Erasmus University, Rotterdam, The Netherlands.

[Plaat et al. 1996]
A. Plaat, J. Schaeffer, W. Piljs, A de. Bruin, 1996. "Exploiting Graph Properties of Game Trees" *In Proceedings of the American Association for Artificial Intelligence (AAAI) National Conference*, pp. 234-239, AAAI Press, ISBN 0-262-51091-X.

[Plaat 1997]
A. Plaat, 1997. "MTD(f), A Minimax Algorithm Faster than NegaScout" Available on the Internet at <http://www.cs.vu.nl/~aske/mtdf.html> [Accessed 1st February 2005]

[Riedmiller and Braun 1993]
M. Riedmiller and H. Braun, 1993. "A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm" In *Proceedings of the IEEE International Conference on Neural Network*, IEEE Computer Society, ISSN 1098-7576.

[Reinefeld 1983]
A. Reinefeld, 1983. "An Improvement to the Scout Tree Search Algorithm" *International Computer Chess Association (ICCA) Journal* (Now ICGA), Vol. 6, No. 4, pp. 4-14.

[Reiss]
M. Reiss. "Go4++" Information can be found on the Internet at <http://www.reiss.demon.co.uk/webgo/compgo.htm> [Accessed 1st February 2005]

[Richards et al. 1998]
N. Richards, D. Moriarty and R. Miikkulainen, 1998. "Evolving Neural Networks to Play Go" *Applied Intelligence*, Vol. 8, No. 1, pp. 85-96., Springer Netherlands, ISSN 0924-669X. Available on the Internet at <http://www.cs.utexas.edu/users/nn/pages/publications/neuro-evolution.html> [Accessed 1st February 2005]

[Rosin and Belew 1995]
C. D. Rosin and R. K. Belew, 1995. "Methods for Competitive Co-evolution: Finding Opponents Worth Beating", In *Proceedings of the Sixth International Conference on Genetic Algorithms*, L. J. Eshelman, editor, Morgan Kaufmann, ISBN 1558603700.

[Rumelhart et al. 1986]
D. E. Rumelhart, G. E. Hinton and J. L. McClelland, 1986. "Learning Internal Representations By Error Propagation" In Parallel Distributed Processing, D.E. Rumelhart and J. L. McClelland eds., pp. 318-362, Cambridge, Massachusetts, 1986. MIT Press, ISBN 0262181207.

[Russell and Norvig 1995]
S. Russell and P. Norvig, 1995. "Artificial Intelligence: A Modern Approach" Prentice Hall, ISBN 0-13-103805-2.

[Schaeffer 1989]
J. Schaeffer, 1989. "The History Heuristic and the Performance of Alpha-Beta Enhancements", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, No. 11, pp. 1203-1212, IEEE Computer Society, ISSN 0162-8828.

[Schaeffer and Plaat 1996]
J. Schaeffer and A. Plaat, 1996. "New Advances In Alpha-Beta Searching" In the *Proceedings of the ACM 24th Annual Conference on Computer Science*, pages 124-130, ISBN: 0-89791-828-2.

[Schraudolph et al. 1994]
N. Schraudolph, P. Dayan, and T. Sejnowski, 1994. "Temporal Difference Learning of Position Evaluation in the Game of Go" In *Advances in Neural Information Processing Systems 6*, Morgan Kaufmann. Available on the Internet at <http://www.gatsby.ucl.ac.uk/~dayan/papers/sds94.pdf> [Accessed 1st February 2005]

[Scott 2002]
J. Scott, 2002. "Machine Learning in Games" Available on the Internet at <http://satirist.org/learn-game/> [Accessed 6th February 2005]

[SGF 1997]
Smart Game File Format FF[4], Specification by A. Hollosi, Additional information can be found at <http://www.red-bean.com/sgf/> [Accessed 1st February 2005]

[Spofford]
J. Spofford. "The Primordial Life Notebook" Available on the Internet at <http://www.io.com/~spofford/> [Accessed 8th March 2005]

[Stoutamire 1991]
D. Stoutamire, 1991. "Machine Learning Applied to Go" *MSc thesis*, Case Western Reserve University. Available on the Internet at <http://david.stoutamire.com/publications.html> [Accessed 6th February 2005]

[Sutton 1988]
R. S. Sutton, 1988. "Learning To Predict By The Method Of Temporal Differences" *Machine Learning* Vol. 3, Issue 1, pp. 9-44, Kluwer Academic Publishers, ISSN 0885-6125..

[Sutton and Barto 1998]
R. S. Sutton and A. G. Barto, 1998. "Reinforcement Learning: An Introduction", MIT Press, Cambridge, MA, ISBN 0262193981.

[Tesauro 1994]
G. Tesauro, 1994. "TD-Gammon, a self-teaching backgammon program, achieves master level play" *Neural Computation*, Vol. 6, No.2, MIT Press, ISSN 0899-7667.

[Thomsen 2000]
T. Thomsen, 2000. "Lambda-search in Game Trees – With Application to Go" *International Computer Games Association Journal*, Vol. 23, No. 4, pp 203-217, December 2000.

[Tucker 1950]
A. W. Tucker, 1950. "A Two-Person Dilemma", memo, Stanford University. Published under the heading "On Jargon: The Prisoner's Dilemma", *UMAP Journal*, Vol. 1, 1980, pp. 101, COMAP Inc, ISSN 0197-3622.

[Van Der Werf 2002]
E. Van Der Werf, 2002. "Local Move Prediction in Go" In the *Proceedings of the 3rd International Conference on Computers and Games*, Edmonton. Available on the Internet at <http://www.cs.unimaas.nl/~vanderwerf/publications.html> [Accessed 1st February 2005]

[Watkins 1989]
C. J. C. H., Watkins, 1989. "Learning From Delayed Rewards" *PhD Thesis*, Univeristy of Cambridge.

[Wilcox 1988]
B. Wilcox, 1988. "Computer Go" In *Computer Games*, D.N.L. Levy ed., Vol. 2, pages 94-135, Springer-Verlag, New York.

[Winands 2001]
M. H. M. Winands, J. W. H. M. Uiterwijk and H. J. van den Herik, 2001. "Combining Proof-Number Search with Alpha-Beta Search" *Proceedings of the Thirteenth Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2001)*, eds. B. Krose, M.de Rijke, G. Schreiber and M. van Someren, pp 299-306.

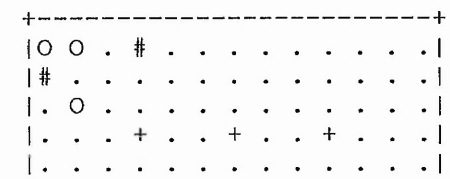
[Woodcock 2004]
S. M. Woodcock, 2004. "Game AI" Available on the Internet at <http://www.gameai.com/> [Accessed 6th February 2005]

[Zobrist 1969]
A. Zobrist, 1969. "A Model of Visual Organization for the Game of Go" In the *Proceedings of the AFIPS Spring Joint Computer Conference*, Boston, AFIPS Press, Montvale, NJ, pages 103-111.

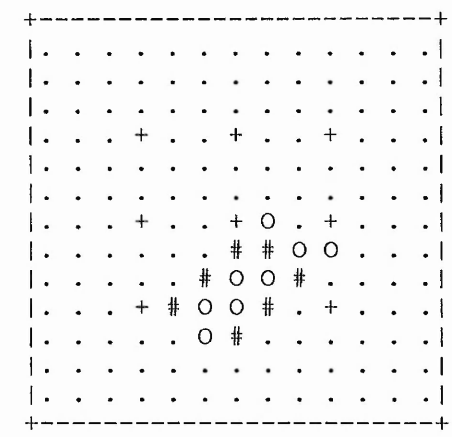
Appendix A
Search Test Diagrams

In each problem black ('#') is attempting to capture white 'O' stones.

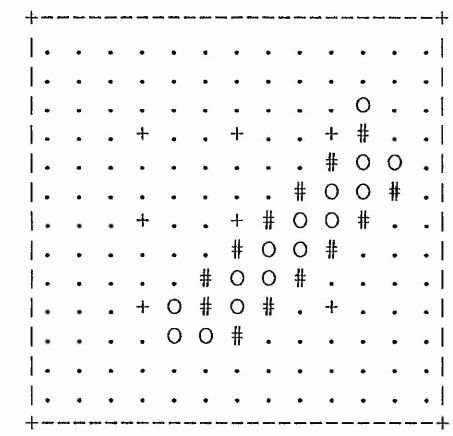
Problem 1 - Custom



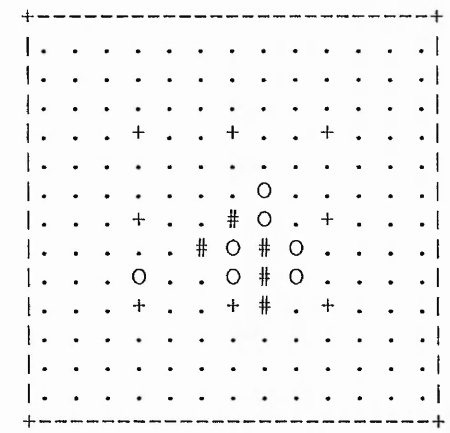
Problem 2 - Graded Go Problems
1 - 39



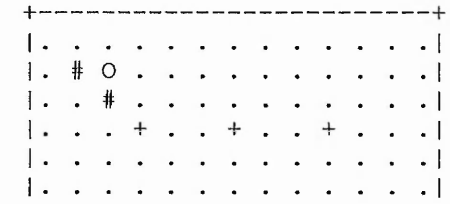
Problem 3 - Graded Go Problems
1 - 87



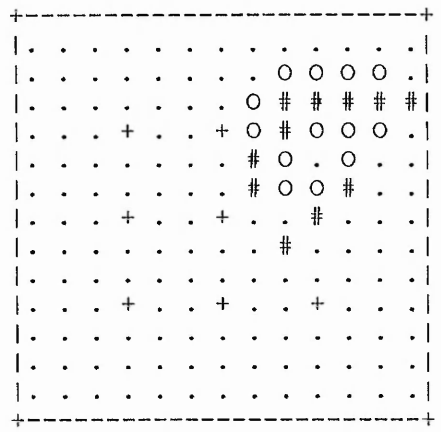
Problem 4 - Graded Go Problems
1 - 88



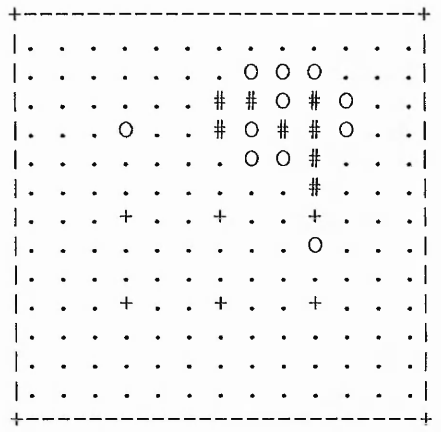
Problem 5 - Custom



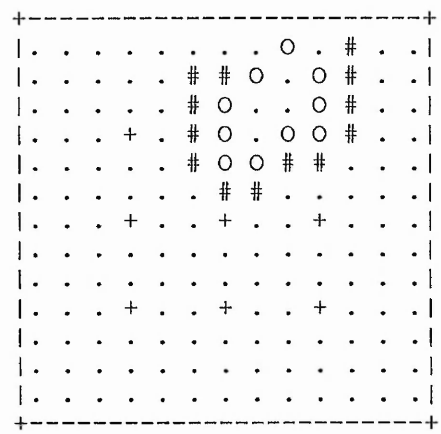
Problem 6 - Graded Go Problems
2 - 3



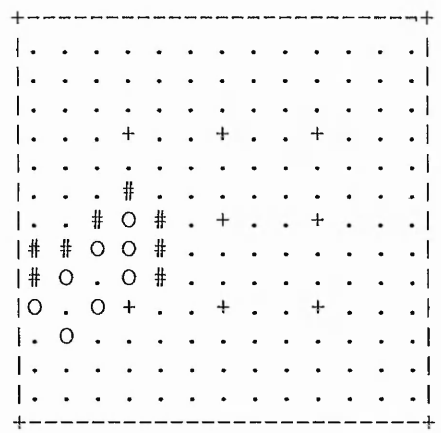
Problem 9 - Graded Go Problems
2 - 7



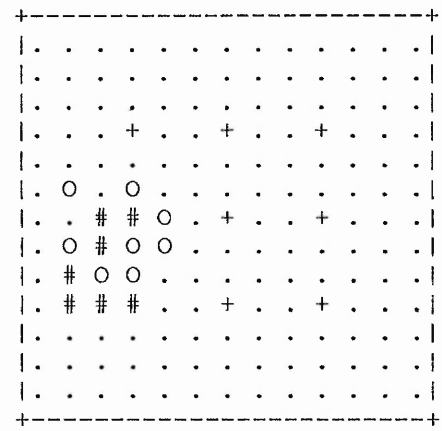
Problem 7 - Graded Go Problems
2 - 43



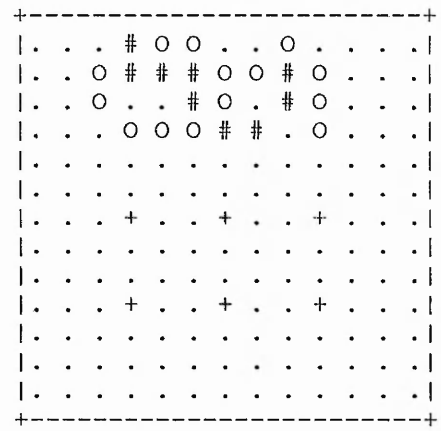
Problem 10 - Graded Go Problems
2 - 10



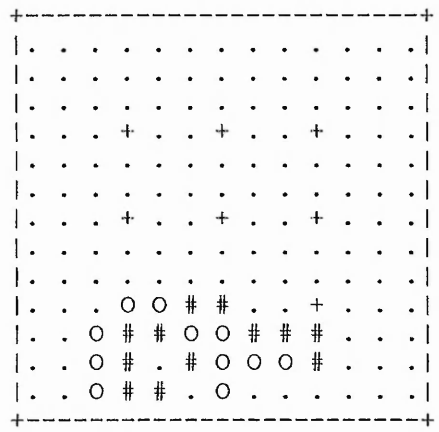
Problem 8 - Graded Go Problems
2 - 2



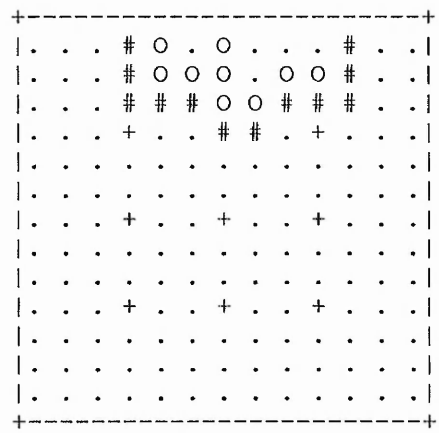
Problem 11 - Graded Go Problems
2 - 13



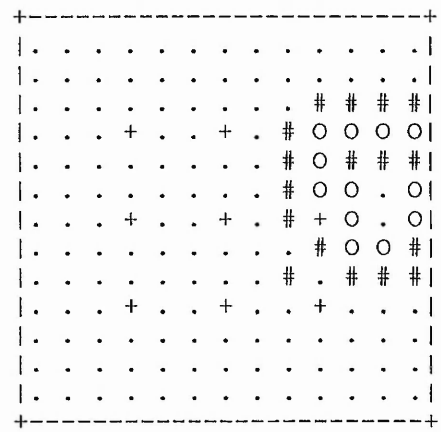
Problem 12 - Graded Go Problems 2 - 32



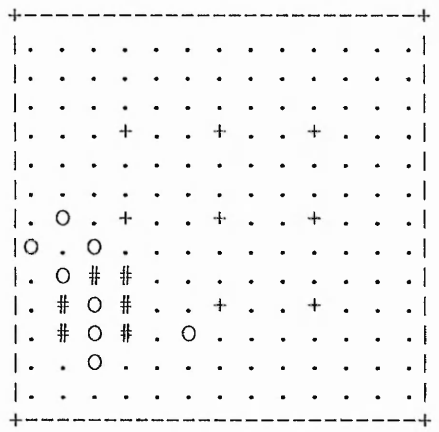
Problem 13 - Graded Go Problems 2 - 35



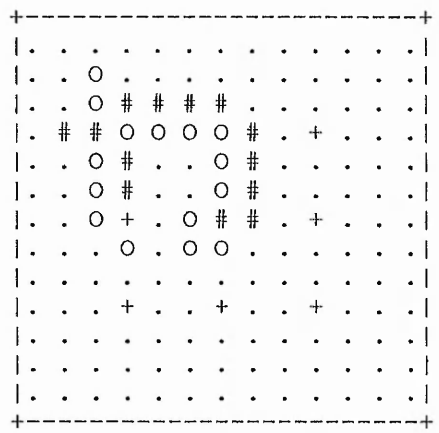
Problem 14 - Graded Go Problems 2 - 41



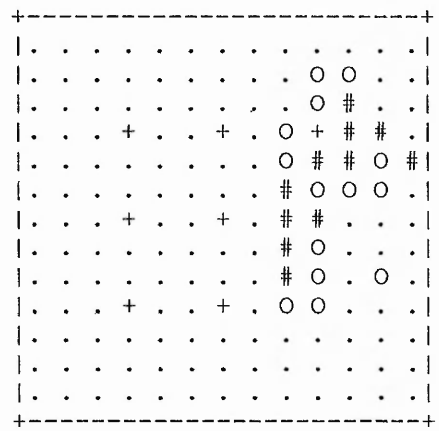
Problem 15 - Graded Go Problems 2 - 68



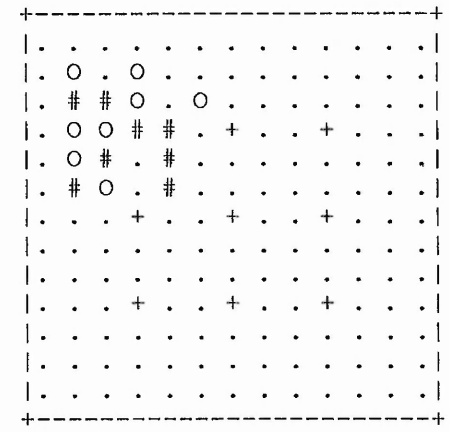
Problem 16 - Graded Go Problems 2 - 70



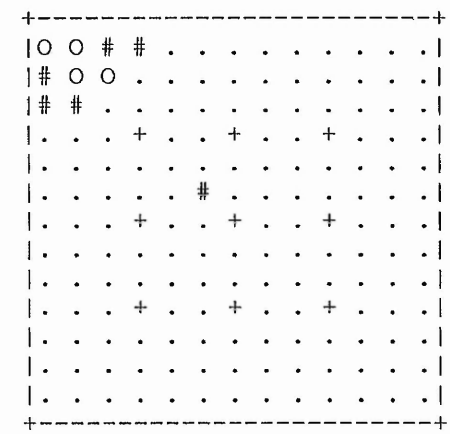
Problem 17 - Graded Go Problems 2 - 79



Problem 18 - Graded Go Problems
2 - 86



Problem 19 - Custom



Appendix B

Software Design – Modules and Classes

The software system was built using the Object Oriented Design methodology and the C++ programming language. Every effort was made to maintain a platform independent design avoiding OS dependent constructs or compiler extensions. This allows the development environment to be relocated as necessary in the future however some concessions were made concerning the use of the Visual Studio 6 IDE under Windows to ease code development.

This rest of this appendix describes the software system in terms of its major components or modules, a description of helper projects and a module-by-module breakdown highlighting the critical classes and their relationships. The main Go playing engine is contained in the Morgoth module. Extensions to this involving Artificial Neural Networks are supplied in the Neural Network module and the GA subsystem is located in the Genetic Algorithm module. UML diagrams are supplied in the following appendix to clarify the relationships between these modules and classes.

Modules

Since Visual Studio was used to organise the software by the author, project files (.dsp) are are supplied with each module where a buildable component such as a DLL or executable exists. An overall Visual Studio workspace file (.dsw) is provided with the NeuralGo module, which contains all of the relevant C++ based projects.

Morgoth

Morgoth is the Go playing engine which can be run standalone with a text menu interface or through the Go Text Protocol (GTP).

GeneticAlgorithm

Encapsulates the generic Genetic Algorithm code, plus some experimental SANE inspired classes. The most important GA classes are GeneticSearch and Genome. These are used by the Morgoth module to implement an experimental GA designed to be an alternative to Minimax based game tree search.

Goal

The Goal module contains generic Goal classes that can be used to define the acceptance criteria for Go problems. The goals are used by the MTD(f) minimax search implementation in the Morgoth module and by the Genetic Algorithm module. All goals implement GoalInterface which in turn implements GoalFunctions. The Goal class is a container for multiple GoalInterface derived classes, allowing each goal to have a weight assigned to it and various functions to be applied to all the goals within a Goal object, i.e. calling the evaluateForGameTreeSearch() function and testing the types of goal that make up a Goal object.

NeuralNetwork

This module handles all generic Artificial Neural Network code and allows the use of various types of Neural Network including BPNs and temporal difference based networks. Importantly this includes the generic Matrix class, which is used by all Neural Network implementations and throughout the Morgoth module. Various Go adapter classes are in this module that provide the interface between the generic NN implementations and Go specific code.

Patterns

This module contains the beginnings of a pattern matcher module, including eye space matching for life and death analysis.

SGF

The SGF module contains code to allow reading of SGF version 4 files – with the exception of handling branched or complex files. Includes preliminary work on writing SGF files but needs testing.

Util

Contains various pieces of source code that are shared by many of the aforementioned modules including a Go board representation, a Go class that provides rule checking for the game amongst other functions, a logging class and an optimised hash map implementation.

Helper Projects

Project titles in *italics* are deprecated.

MorgothDLL

MorgothDLL provides Visual Studio project files and additional source code to compile Morgoth as a Windows DLL including definitions of which functions are publicy available to the DLL user.

MorgothNETGUI

This program provides a graphical front end to MorgothDLL using the .NET language.

MorgothVBGUI

This program provides a graphical front end to MorgothDLL but has been deprecated in favour if MorgothNETGUI.

MorgothGUI

The successor project to NeuralGo - this project is also deprecated in favour of a .NET or VB based GUI using MorgothDLL to provide functionality.

NeuralGo

NeuralGo provides a Windows based GUI which can load SGF version 4 files and be used to play against a GTP program or to play two GTP programs against one another. NeuralGo also understands some custom GTP commands specifically designed to be used with Morgoth. This GUI has been deprecated in favour if MorgothNETGUI.

CreateNet

CreateNet is used to create Neural Networks, specifying number of layers, neurons per layer and other training and architecture specific variables. The NN specification is stored in a .bpn file.

NetInfo

NetInfo is used to print out various pieces of information about a Neural Network.

TestNet

TestNet is a program that is used to invoke a selection of performance tests on a specified Neural Network. It can be used to monitor the training progress given a set of SGF files to assess the NN against.

TrainNet

Trainnet is a program that is used to train Neural Networks on a repository of SGF games. Limits on the duration of the training can be specified such as a target quality level or number of epochs to train and a variety of training algorithms can be selected to use during training. This program does not restrict training to the use of backpropagation based algorithms.

TrainEvalNet

TrainEvalNet is used to play through a set of SGF games and train a Neural Network based on those games with the goal of correctly evaluating a board position. This program is currently unused.

Module Breakdown

This section gives a breakdown of the classes within each module including a brief description of functionality for each class. The classes are presented in alphabetical order and the most important classes in each module are highlighted in **bold**.

Morgoth

Class Name	Description
GoMorgoth	**This class provides functions to control a number of user interfaces, a text UI, a GTP based client and a legacy mode to allow the NeuralGo GUI to interact with Morgoth. Several functions are available here, for example functions to get the best move according to Morgoth with or without a game tree search. Internal references are held within this class for relevant NNs.**
GTPFunctions	This class provides the functionality to use the GTP protocol and supplies a gtp_main_loop() function to read and interpret input from stdin (or optionally a file) as GTP commands. Custom GTP commands should have a function per command to implement the required functionality. Helper functions are also supplied to allow
LineOfPlayGenome	A class derived from Genome in the Genetic Algorithm module. This class defines a Genome as representing a sequence of moves as defined in the Move class in the Util module.
LineOfPlayGS	**The class extends the GeneticSearch class supplied in the GeneticAlgorithm module providing implementations of the abstract functions, most importantly the doSearch() function which will literally run a GA based search given a board position and return the best available move and line of play in the form of a LineOfPlayGenome object.**

LogWriter	This class implements the LogWriter abstract class from Util providing a common logging facility throughout Morgoth.
Main	Not a class but a file containing the main function. Handles command line arguments, creates an initial Morgoth object and runs the appropriate mode function on it..
Morgoth	**Provides an entry point for the three operational modes; a text UI (DoMainMenu()), GTP (playGTP()) and a legacy NeuralGo GUI interfacing mode (playNeuralGoMode()). This is also the central object which contains instances of GoMorgoth, MTDFSearch, LineOfPlayGS, Test and GTPFunctions. Also holds global parameters typically set at the command line.**
MorgothDLL	Not a class but a header and a source file containing functions to expose parts of Morgoth as a DLL interface.
MoveSuggester	An abstract class that represents an object that can suggest potential Go moves, optionally supplying them in some order.
MTDFSearch	**An implementation of the SearchAlgorithm interface supplying a basic Alpha-Beta Minimax tree search. Functions are also supplied to extend the search with Iterative Deepening and/or MTD(f). Methods are available to set the node expansion and node evaluation functions for a given search configuration, which can also allow user defined functions to be substituted.**
Node	This class represents a node in an Alpha-Beta based search tree and is used by MTDFSearch.
SearchAlgorithm	A pure interface defining functions that a Move and Goal based search algorithm must implement.
Test	A collection of test functions covering Go rule implementation, SGF reading and navigating, BPN training, Alpha-Beta searches, Genetic Algorithm searches and some Go specific board representation mechanics.

GeneticAlgorithm

Class Name	Description
GASwitches	Not a class at all but a header file containing some preprocessor defines to control how the Genetic Algorithm configuration is compiled.
GenerationHistory	This class provides a container to store a history of successive generations of genomes, including details of how the genomes evolved from generation to generation.
GeneticSearch	**This class contains functions to use a custom genetic algorithm to generate lines of play. Included in this is an inner class definition of ConfidenceCriteria based on the number of refuted attacks at various levels. The history of a search is stored as a member of this class using the GenerationHistory type. Many other variables and functions to change them are supplied and a doSearch() pure virtual function definition is included to provide an entry point to GeneticSearch based searches.**
Genome	This class defines a generic Genome. It uses the DAG and DAGNode classes from Util to store the structure, which may be linear or can take advantage of the properties of DAGs (Directed Acyclic Graph). Several functions are defined to handle reproduction and mutation and the default Genome implementation allows information to be stored about refuted and unrefuted attacks and locations within a Genome.

Goal

Class Name	Description
CaptureGoal	This class represents a goal with the aim of capturing a string of one or more stones at a given point.
Goal	**This class represents an abstract goal which can consist of**

	one or more weighted goals of any GoalInterface derived type.
GoalFunctions	**This class provides a pure virtual interface which both GoalInterface and Goal implement.**
GoalInterface	**This class provides an abstract interface from which concrete goal types can be extended.**
InfluenceGoal	This class represents a goal with the aim of maximising an influence score. This uses the calculateInfluence() function from Misc.
LadderGoal	This class represents a goal with the aim of capturing a string of one or more stones in a ladder, a more specialised form of CaptureGoal.
LibCountCaptureGoal	This class represents a goal with the aim of capturing a string of one or more stones using a count of the liberties of the target string to influence the evaluation. This class is a more specialised form of CaptureGoal.
LibertyCountGoal	This class represents a goal with the aim of maximising the relative number of friendly liberties available.
LifeDeathGoal	This class represents a goal with the aim of capturing a string of one or more stones using Life and Death tactical knowledge such as eye pattern matching and analysis. This class is a more specialised form of CaptureGoal. Note: this is incomplete.
MoveFinderGoal	This class represents a goal with the aim of maximising the relative friendly score of a board position calculated with an ANN.
StoneCountGoal	This class represents a goal with the aim of maximising the relative number of friendly stones present on the board.

NeuralNetwork

Each of the BPNGoAdapter classes (except BPNGoAdapterInterface) implements BPNGoAdapterInterface and they are detailed in Chapter 6. The related input scheme code is listed in italics.

Class Name	Description
BPN	**A class which extends NeuralNetwork for Backpropagation based networks. Provides structure and usage functions – but not training.**
BPNGoAdapter	*S1*
BPNGoAdapterInterface	**An interface to adapt BPN derived classes to be used in a Go context. This includes conversion of data from a board position to BPN input and various performance testing functions.**
BPNTrainer	**This class uses a BPN object a training target providing methods to train the network with the Backpropagation method over time, a number of epochs or until some quality constraint is reached.**
CN	Extends BPN to implement a Competitive Network as described at the end of Chapter 6.
CNGoAdapter	An adapter to allow CN type ANNs to be used in a Go context.
Matrix	**A utility class to provide an efficient 2 dimensional matrix of a template type. Used extensively by the ANN classes.**
NeuralNetwork	**An abstract class to represent a generic Artificial Neural Network.**
newBPN10GoAdapter	This class and the following 11 GoAdapters each implement the BPNGoAdapterInterface and are detailed in Chapter 6. The related input scheme is listed in italic. *S4*

newBPN11GoAdapter	*S4D2*
newBPN12GoAdapter	*S3D18*
newBPN2GoAdapter	*S3D18 duplicate*
newBPN3GoAdapter	*S4D18*
newBPN4GoAdapter	*S4D18Libs*
newBPN5GoAdapter	*S4D18DLM*
newBPN6GoAdapter	*S4D18ELM*
newBPN7GoAdapter	*S2 (OURCOLOUR, THEIRCOLOUR)*
newBPN8GoAdapter	*S2 (BLACK,WHITE)*
newBPN9GoAdapter	*S3*
newBPNGoAdapter	*S3D2*
NNDatabase	**This class represents a common repository for holding training pairs of data to be used to train ANNs.**
ObjectMatrix	This class is like Matrix except it should also be able to handle non-primitive data types, i.e. objects. May be a legacy class – refer to Matrix in the source code.
RPROPBPNTrainer	A BPNTrainer derived class that implements the RPROP ANN training algorithm [Riedmiller and Braun 1993].

Patterns

Class Name	Description
EyeShapePattern	A class to represent an eye shape pattern.
Pattern	An abstract pattern class.
PatternMatcher	**A class to match a given set of patterns to a given board position.**

SGF

Class Name	Description
SGFGameInfo	This class holds the game information for an associated SGFTree.
SGFNode	This class represents a node in an SGFTree.
SGFProperty	This class represents a text SGF property tag.
SGFReader	This class provides functionality to read an SGFTree from an SGF file, format version 4.
SGFTree	This class represents a tree of SGFNodes and provides facilities to search, traverse and edit the tree.

Util

Class Name	Description
BMP	A class to read, write and create 24bpp palette-less bitmap files.
BoardStruct	**The core Go board representation class. Provides a level of abstraction through GoString objects which are kept dynamically up-to-date as the board changes.**
DAG	A representation of a generic Directed Acyclic Graph based on templates.
DAGNode	A node in a DAG. Failrly generic except for a small amountof additional storage for GeneticSearch related information per node.
globals	Not a class at all but a header file containing some preprocessor defines to control various aspects of the build configuration.
Go	**A large helper class which manages the main BoardStruct object within the system. Has knowledge of the handicap system, rules, komi and functions to determine whether a move is legal or not.**

GoString	**A representation of a same colour connected string of stones. Used by BoardStruct to maintain efficient incremental board data.**
hash_map	An implementation of a hash map based on Stroustrup's "The C++ Programming Language" 3rd edition.
hash_set	An implementation of a hash set based on Stroustrup's "The C++ Programming Language" 3rd edition.
hash_shared	Common functionality shared between hash_map and hash_set.
LogWriter	A common logging interface without output sent to stderr by default.
memoryHelper	Utility functions to help track memory leaks.
memoryTracking	Utility functions to help track memory leaks.
Misc	**A helper class containing assorted functions including calculateInfluence(), findAdjacentStrings() and OS independent timing routines.**
Move	An object representing a move with some extra information including alpha-beta score and NN score. Also provides routines to convert between different move formats, i.e. SGF, GTP, GMP and GNUGo.
Point	A simple 2 dimensional coordinate class.
Rect	A simple rectangle class.
SmartPtr	An implementation of the smart pointer idea using reference counting.
SpecialPoint	Extends Point and includes two additional variables to encode user information. Occasionally used to rank moves using light weight objects.

Appendix C

Software Design – Diagrams

This appendix contains UML diagrams describing the important module relationships and object interactions.

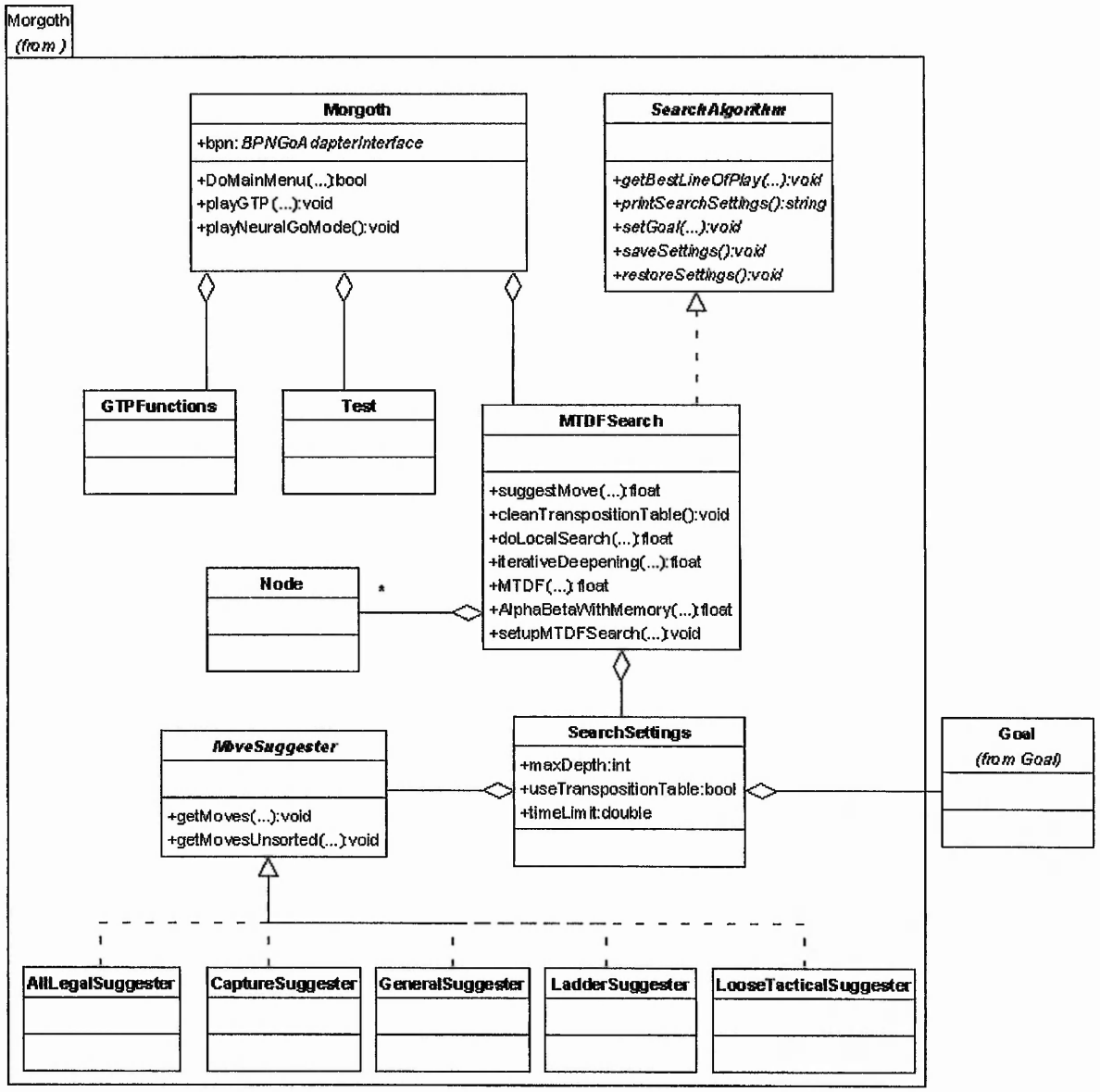


Figure C-1 - Morgoth Package

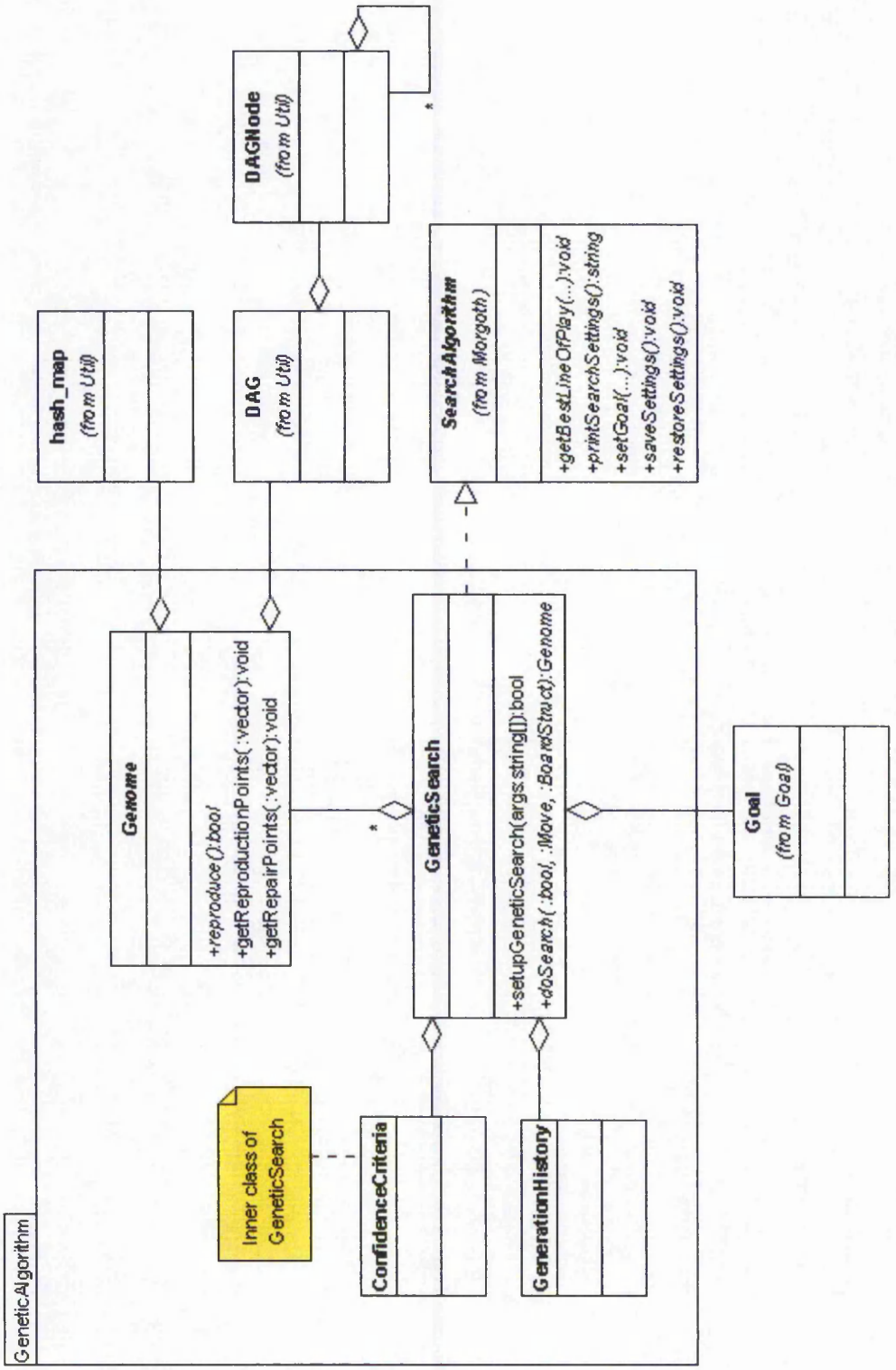


Figure C.2 - Genetic Algorithm Package

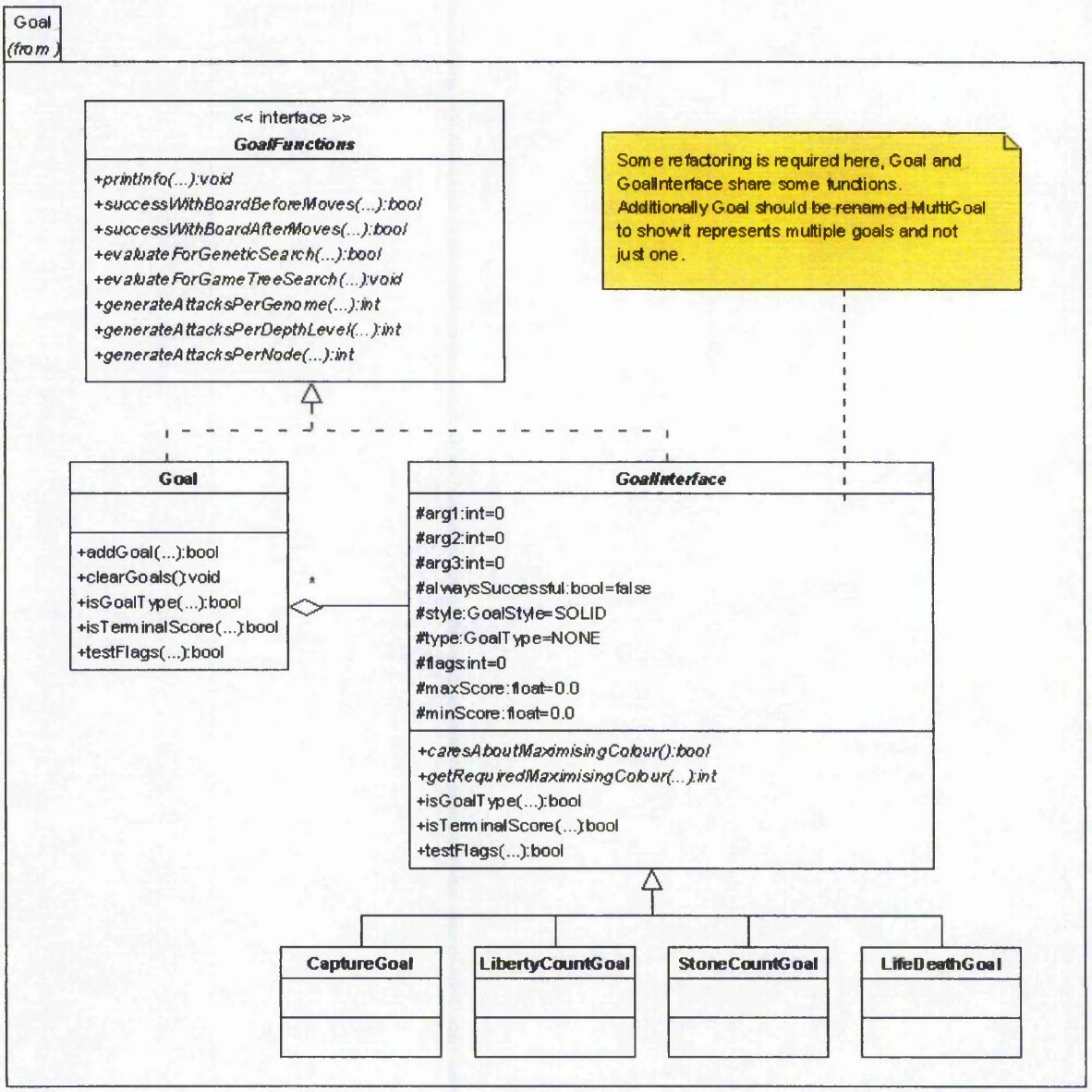


Figure C.3 - Goal Package

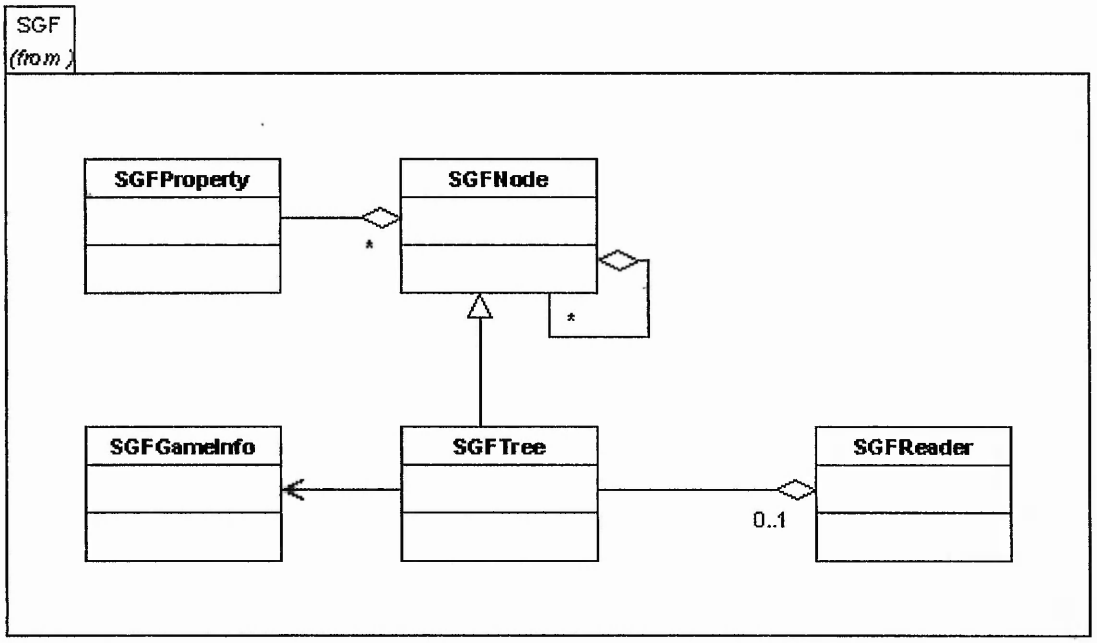


Figure C.4 - SGF Package

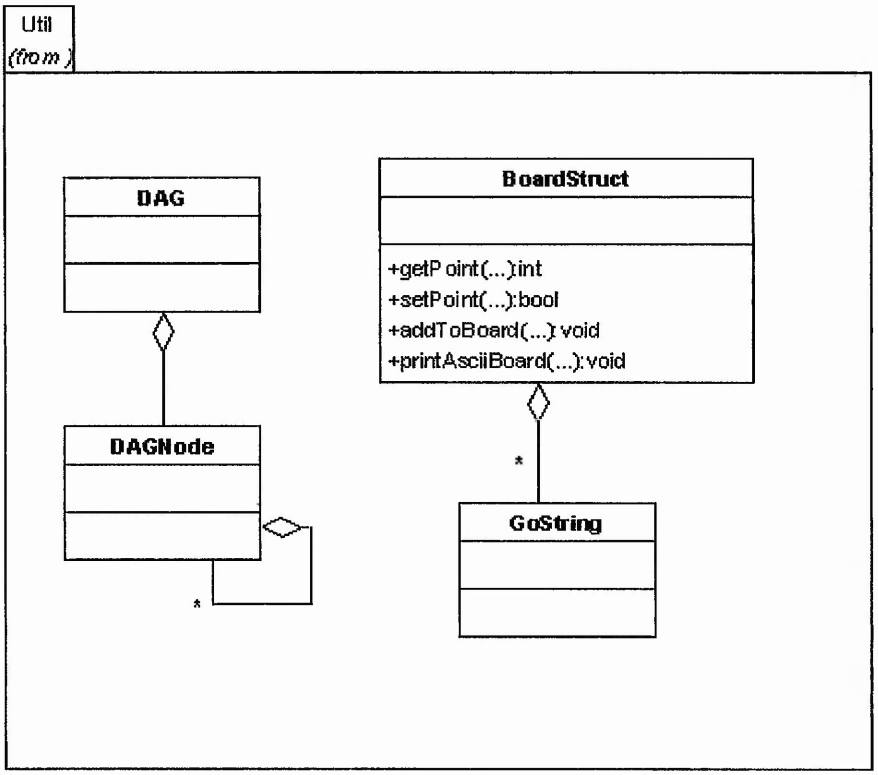


Figure C.5 - Util Package

Appendix D

Genetic Algorithm

This appendix provides a step-by-step account of the control flow within the hybrid Genetic Algorithm. An example chromosome that has been through many of the processes described in the text below is shown at the end of this appendix.

The Algorithm

1. Setup
    a. Generate legal moves for white and black, scored by the ANN given the current board position.
    b. If there are no moves for the next player that score higher than the MTDFSearch prune variable, return a pass. This ensures the GA will return a pass when MTDFSearch does.
2. Create an initial population of chromosomes
    a. Mechanistic generation, the following methods each attempt to generate one chromosome to add to the population which is accepted if it is legal and unique. There is no requirement that the chromosomes achieve the goal at this stage.
        i. Highest scoring moves sequence using the ANN.
        ii. Extended liberty count move sequence.
        iii. As ii) but all enemy moves are passes.
        iv. All moves are passes.
    b. Generate random ANN biased chromosomes to fill up remaining space in the population.
3. Goto 10 if any of these conditions are true:
    a. We have run out of time.
    b. We have found a chromosome with maximum refuted attacks and no unrefuted attacks i.e. we have found a 'good enough' chromosome.

c. We have found maximum unrefuted attacks for all chromosomes in the population.

4. Choose a chromosome.
    a. First choose from the chromosomes with no unrefuted attacks.
    b. Otherwise choose amongst chromosomes with less than the maximum unrefuted attacks. Bias the choice using the number of unrefuted attacks for each chromosome so that we are less likely to pick chromosomes with high numbers of unrefuted attacks.
5. Generate an attack position.
    a. If we can't find a random attack position within a given number of attempts go to 3.
6. Generate an attack. 50/50 chance of:
    a. Extended liberty count move selection.
        i. If we can't find one move on to 6b). Note: it's not necessary to check if this move is the same as the one it's replacing, the exLib generator will do a simple check to make sure this attack has not been played here already.
    b. Or ANN biased move selection.
        i. If this attack has already been tried choose another random ANN biased move.
        ii. If there are no moves available that this chromosome has not been attacked with at this position go to 3.
7. If the attack is immediately refutable (1 move refutation) then:
    a. Increase refuted attack count for all previous enemy colour nodes.
    b. If there are still attacks remaining for this node go to 6 and get the next one.
    c. Otherwise there must be no attacks left at this node. Choose another chromosome, go to 3.
8. Attack move integration. Copy the subtree, analyse and in the process:
    a. Copy subtree and traverse it depth first.
    b. Try to find a post-refutation for the attack.
        i. Try the following refutations in order, if any succeed store the attack and refutation in the chromosome and go to 3.

1. Ineffective move: If the move has no effect, i.e. the goal is still achieved; the refutation is simply the rest of the line of play.
2. Mirror enemy move. Try to play the enemy move replaced by the attack as a friendly move.
3. Extended liberty count move sequence. Generate a series of extended liberty moves until we refute the attack or reach the maximum chromosome size.
4. Search other chromosomes. Search through the population attempting to find a single, consecutive section of a chromosome that can be attached after the attack to refute it.

ii. If a refutation is found, go to 10 if chromosome is 'good enough'.

9. Repair phase.
   a. Attempt to correct failed refutations or refute unrefuted attacks.
   b. Choose a random chromosome biased by the inverse of the number of unrefuted attacks on the chromosome.
   c. Use non-deterministic methods only:
      i. ANN move sequence. Generate a series of moves using the ANN to bias the random selection. Continue until we refute the attack or reach the maximum chromosome size.
      ii. Apply the mutation or crossover operator.
   d. If a refutation is fixed or found for an unrefuted attack and the chromosome is 'good enough', go to 10.
10. Reproduction phase.
   a. For each chromosome choose between mutation and crossover.
   b. Randomly choose a modification site within the chromosome dependent on restrictions due to mutation or crossover mechanism.
   c. Create a new chromosome using the selected mechanism and add to the population if at least one fault is repaired over the original chromosome.
11. Population Culling (elitism).

a. Remove chromosomes based on fitness until population is within maximum size limit.

12. Chromosome trimming. Remove branches for chromosomes that have exceeded the maximum size limit.
13. Go to 3.
14. Select best chromosome from these conditions:
    a. If we have any successful chromosomes (fitness>0) with no unrefuted attacks pick the one with the maximum refuted attacks if single, otherwise choose one of those in order of fitness.
    b. If no chromosomes match a) or we have several with equal fitness's choose the chromosome with the lowest unrefuted attack count.
    c. Else choose the chromosome with the highest refuted attack count, if there's more than one, select the first that occurs in the population.

Note – A fitness value is maintained indicating the success and efficiency of the principal variation. In the case of capture goals it is the success (1 or 0 for failure) divided by the length of the solution. Code is included in the system which checks attack counts and fitness values to select the best chromosome. See point 10.

Example Chromosome

The following diagram shows a chromosome with one remaining unrefuted attack (W_{23}). The black path indicates the principal variation, red genes are attacks and purple genes are mutations (they would not be stored in the originator chromosome but used to create a new chromosome). Blue paths show successful refutations and red paths represent unsuccessful refutations. Boxed genes are chromosome terminators and indicate successful completion of the goal.

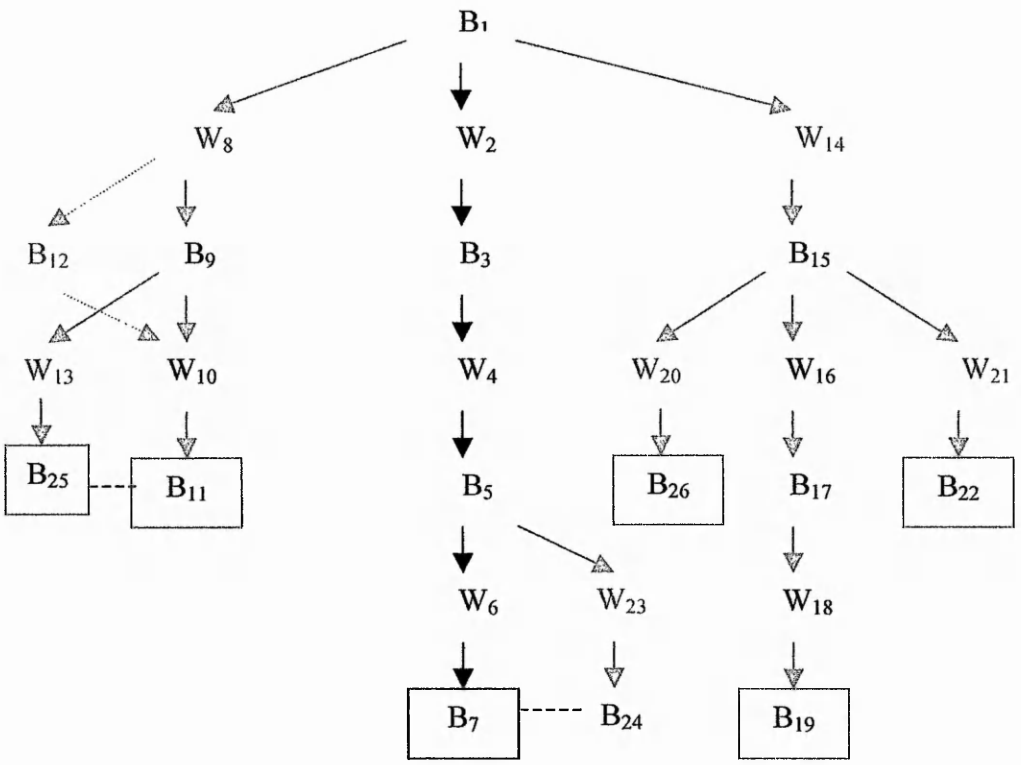


Figure D.1 – Example Chromosome

Appendix E

Neural Network Results

This appendix contains detailed results concerning the Artificial Neural Network training and performance measurements.

Input Field Size	Average Rank	Standard Deviation	Median	Mode
3x3	30.0809	41.3942	13	1
4x4	26.4412	38.3062	12	1
5x5	23.1216	31.2395	12	1
6x6	23.3903	31.2958	12	1
7x7	22.2596	29.2103	12	1
8x8	22.6053	29.4296	12	1
9x9	22.0959	28.1145	12	1
10x10	24.1592	30.9393	13	1
11x11	23.5237	30.6573	12	1
12x12	26.0464	33.4492	14	1
13x13	25.7988	29.9899	15	1

Figure F-1 - Input Field Size

Learning Rate	Average Rank	Standard Deviation	Median	Mode
0.001	21.8135	29.1219	11	1
0.01	19.3955	25.5659	9	1
0.15	21.0740	28.1627	11	1
0.25	22.0959	28.1145	12	1
0.5	24.7545	31.154	14	1
0.7	26.5301	32.9944	15	1
Dynamic	125.6050	81.4568	114	86

Figure F-2 - Learning Rate

Momentum	Average Rank	Standard Deviation	Median	Mode
0.1	21.2021	27.5558	11	1
0.3	21.4954	27.228	11	1
0.5	21.8132	28.1135	11	1
0.7	22.0959	28.1145	12	1
0.9	72.7441	58.0405	60	78

Figure F-3 - Momentum

Hidden Neurons	Average Rank	Standard Deviation	Median	Mode
20	24.5925	31.1252	13	1
40	22.0959	28.1145	12	1
60	21.4390	29.1404	11	1
80	20.8547	28.9074	10	1
100	20.1642	27.5976	10	1
120	20.4543	28.9683	10	1
140	19.9058	28.4617	9	1

Figure F-4 - Hidden Neurons

Training Range	Average Rank	Standard Deviation	Median	Mode
20k-10k	28.2212	35.617	15	1
10k-5k	25.1546	34.9757	12	1
5k-1k	23.4672	29.22	13	1
1d-5d	23.8196	31.0043	12	1

Figure F-5 - Rank Range for Training Data

Number of Hidden Layers	Average Rank	Standard Deviation	Median	Mode
1	22.0959	28.1145	12	1
2	20.7209	28.6342	10	1
3	20.9931	28.6673	10	1
4	22.9122	29.8486	12	1

Figure F-6 - Number of Hidden Layers

Input Field Shape	Average Rank	Standard Deviation	Median	Mode
Square	22.0959	28.1145	12	1
Diamond	22.1856	30.3781	11	1

Figure F-7 - Input Field Shape

Training Data Source	Average Rank	Standard Deviation	Median	Mode
NNGS	22.0959	28.1145	12	1
KGS	23.7128	30.3734	12	1

Figure F-8 - Training Data Source

Appendix F

Genetic Algorithm Results

This appendix contains detailed results concerning the Genetic Algorithm development and performance. The tables below show the results of running various GA configurations against the set of test Go Problems illustrated in Appendix A. The tests were all averaged over ten runs and the tables include columns to show the success rates of each available refutation method and the two repair methods on a per problem basis.

							Refutations				Repair	
No.	Name	Time(s)	Success Rate	Nodes Created	Evaluation Calls	Algorithm Iterations	NoEffect	Mirror Success / Total	ExLib Success / Total	SearchGenomes Success / Total	Mutation Success / Total	SearchGenomesRepair Success / Total
1	Short capture	0.2	1	77.9	61.4	1.7	0.0	0.0 / 0.2	0.0 / 0.2	0.0 / 0.2	0.8 / 4.3	0.3 / 4.5
2	GG1-P39	1.4	0.9	740.5	709.0	19.6	0.0	0.7 / 7.5	5.9 / 6.8	0.2 / 0.9	4.1 / 102.8	0.8 / 106.6
3	GG1-P87	1.3	1	598.1	572.4	12.2	0.0	0.1 / 6.4	3.4 / 6.3	0.6 / 2.9	3.2 / 63.8	1.1 / 63.4
4	GG1-P88	18.4	0.5	22043.9	11006.8	566.3	6.2	0.1 / 309.5	132.3 / 309.4	157.2 / 177.1	3.7 / 3130.3	1.1 / 3095.4
5	2nd line trap	0.2	0.9	240.5	227.3	9.1	0.0	0.1 / 4.9	3.8 / 4.8	0.3 / 1.0	2.9 / 46.1	0.4 / 46.0
6	GG2-P3	0.5	1	279.3	273.0	9.1	0.0	0.1 / 4.9	3.4 / 4.8	0.6 / 1.4	4.1 / 46.4	0.5 / 46.9
7	GG2-P43	47.2	0.5	114996.9	103842.3	1013.8	67.5	11.5 / 771.4	259.5 / 759.9	396.2 / 500.4	3.6 / 5579.2	1.3 / 5575.1
8	GG2-P2	0.4	1	230.6	208.7	8.0	0.0	0.0 / 4.8	3.2 / 4.8	0.0 / 1.6	2.8 / 38.6	0.5 / 41.4
9	GG2-P7	59.0	0.1	47737.1	36866.0	1383.5	69.9	65.3 / 382.4	81.9 / 317.1	171.3 / 235.2	5.7 / 7620.0	1.4 / 7604.0
10	GG2-P10	0.7	1	515.4	497.7	19.7	0.2	0.0 / 8.8	5.00 / 8.8	1.4 / 3.8	3.5 / 106.2	1.0 / 103.6
11	GG2-P13	0.6	0.9	726.4	663.8	26.4	0.0	0.0 / 10.3	2.9 / 10.3	4.6 / 7.4	3.7 / 143.3	1.6 / 141.1
12	GG2-P32	59.0	0.5	153039.5	139510.2	1301.2	32.1	27.3 / 1136.8	105.3 / 1109.5	946.4 / 1004.2	6.6 / 7187.0	1.8 / 7130.2
13	GG2-P35	53.4	0	53378.1	41483.1	1125.4	36.4	0.8 / 821.7	100.3 / 820.9	661.6 / 720.6	4.9 / 6176.6	1.7 / 6207.9

14	GG2-P41	42.2	0.8	46378.6	38464.0	878.9	125.2	7.7 / 316.3	26.5 / 308.6	215.4 / 282.1	5.1 / 4867.6	1.8 / 4795.6
15	GG2-P68	48.0	0.4	28834.0	19671.2	1232.4	33.5	222.4 / 563.8	174.8 / 341.4	146.7 / 166.6	5.6 / 6777.0	0.8 / 6782.6
16	GG2-P70	18.9	0.9	13363.9	10691.7	410.8	3.7	17.7 / 119.6	2.6 / 101.9	84.5 / 99.3	4.8 / 2260.5	1.8 / 2256.6
17	GG2-P79	1.9	1	1200.7	1034.6	75.5	1.1	2.4 / 19.2	7.5 / 16.8	4.3 / 9.3	3.4 / 414.0	1.3 / 408.8
18	GG2-P86	0.6	1	493.7	489.9	18.5	0.0	0.1 / 8.0	5.3 / 7.9	1.3 / 2.6	4.9 / 101.3	0.5 / 95.8
19	Ladder3	0.5	0.8	914.7	882.3	29.5	0.2	0.4 / 9.4	5.0 / 9.0	1.1 / 4.0	3.1 / 157.2	0.6 / 159.9
Totals		**354.4**	**14.2**	**485789.8**	**407155.4**	**8141.6**	**376**	**356.7 / 4505.9**	**928.6 / 4149.2**	**2794 / 3220.6**	**76.5 / 44822.2**	**20.3 / 44665.4**
Success Rate			**0.7474**					**0.0792**	**0.2238**	**0.867**	**0.0017**	**0.00045**

Table F-1 - 'All' Configuration

							Refutations				Repair	
No.	Name	Time(s)	Success Rate	Nodes Created	Evaluation Calls	Algorithm Iterations	NoEffect	Mirror Success / Total	ExLib Success / Total	SearchGenomes Success / Total	Mutation Success / Total	SearchGenomesRepair Success / Total
1	Short capture	0.3	1	61.1	38.2	1.4	0.0	0.0 / 0.3	0.0 / 0.3	0.0 / 0.3	N/A	N/A
2	GG1-P39	0.8	0.9	130.4	95.4	12.5	0.0	0.2 / 4.1	3.7 / 3.9	0.0 / 0.2	N/A	N/A
3	GG1-P87	0.9	1	136.9	103.0	4.6	0.0	0.0 / 1.5	0.8 / 1.5	0.2 / 0.7	N/A	N/A
4	GG1-P88	6.6	0.7	12013.7	5172.3	284.8	0.0	99.9 / 249.7	103.6 / 149.8	40.9 / 46.2	N/A	N/A
5	2nd line trap	0.1	1	73.9	47.0	7.1	0.0	0.1 / 3.1	2.7 / 3.0	0.0 / 0.3	N/A	N/A
6	GG2-P3	0.5	1	56.6	34.6	4.0	0.0	0.1 / 1.2	1.0 / 1.1	0.0 / 0.1	N/A	N/A
7	GG2-P43	47.3	0.1	219130.5	186719.2	1265.3	0.0	11.6 / 1109.9	401.2 / 1098.3	474.2 / 697.1	N/A	N/A
8	GG2-P2	0.4	1	55.0	22.7	6.0	0.0	0.1 / 2.6	2.5 / 2.5	0.0 / 0.0	N/A	N/A
9	GG2-P7	59.0	0	42988.6	27069.3	3308.2	0.0	58.8 / 350.5	165.2 / 291.7	99.7 / 126.5	N/A	N/A
10	GG2-P10	0.5	1	81.2	50.4	8.0	0.0	0.0 / 2.2	2.0 / 2.2	0.0 / 0.2	N/A	N/A
11	GG2-P13	0.2	1	350.3	281.5	20.1	0.0	0.4 / 6.2	1.9 / 5.8	2.4 / 3.9	N/A	N/A
12	GG2-P32	53.4	0	87849.1	63672.3	1683.9	0.0	126.9 / 1579.1	383.1 / 1452.2	1013.0 / 1069.1	N/A	N/A
13	GG2-P35	35.9	0	57475.4	38104.1	1161.5	0.0	0.0 / 966.7	197.5 / 966.7	698.1 / 769.2	N/A	N/A
14	GG2-P41	41.3	0	113907.9	98110.6	1551.6	0.0	0.7 / 563.6	11.2 / 562.9	391.4 / 551.7	N/A	N/A

15	GG2-P68	59.0	0	47446.1	23026.4	2790.7	0.0	619.6 / 1268.6	449.4 / 649.0	188.1 / 199.6	N/A	N/A
16	GG2-P70	12.7	0.8	3807.1	1919.4	444.6	0.0	10.8 / 23.1	0.2 / 12.3	4.3 / 12.1	N/A	N/A
17	GG2-P79	1.2	1	445.7	324.6	43.7	0.0	0.5 / 9.1	5.4 / 8.6	0.8 / 3.2	N/A	N/A
18	GG2-P86	0.3	1	122.2	95.0	9.6	0.0	0.0 / 3.3	2.4 / 3.3	0.3 / 0.9	N/A	N/A
19	Ladder3	0.5	1	256.2	218.8	20.2	0.0	0.0 / 4.4	3.2 / 4.4	0.1 / 1.2	N/A	N/A
Totals		**320.9**	**12.5**	**586387.9**	**445104.8**	**12627.8**	**0**	**929.7 / 6149.2**	**1737 / 5219.5**	**2913.5 / 3482.5**	N/A	N/A
Success Rate			**0.6579**					**0.1512**	**0.3328**	**0.8366**	N/A	N/A

Table F-2 - 'Without Reproduction or Repair' Configuration

							Refutations				Repair	
No.	Name	Time(s)	Success Rate	Nodes Created	Evaluation Calls	Algorithm Iterations	NoEffect	Mirror Success / Total	ExLib Success / Total	SearchGenomes Success / Total	Mutation Success / Total	SearchGenomesRepair Success / Total
1	Short capture	0.4	1	72.6	44.9	1.1	0.0	0.0 / 0.1	0.0 / 0.1	0.0 / 0.1	0.0 / 0.1	0.2 / 1.2
2	GG1-P39	1.1	1	2408.3	2373.9	18.1	0.0	0.8 / 6.7	4.3 / 5.9	0.2 / 1.6	0.5 / 17.1	2.8 / 173.8
3	GG1-P87	1.3	1	1306.2	1273.6	12.0	0.0	0.4 / 5.3	2.2 / 4.9	0.3 / 2.7	0.4 / 11.0	2.8 / 112.8
4	GG1-P88	1.2	0.8	2051.3	1962.3	36.8	0.0	1.4 / 13.6	1.3 / 12.2	5.0 / 10.9	0.5 / 35.8	3.6 / 361.6
5	2nd line trap	0.3	0.9	558.9	528.4	9.7	0.0	0.1 / 3.5	2.7 / 3.4	0.2 / 0.7	0.2 / 8.7	1.9 / 88.9
6	GG2-P3	0.8	1	664.5	630.9	12.7	0.0	0.7 / 6.2	4.0 / 5.5	0.7 / 1.5	0.3 / 11.7	2.7 / 119.7
7	GG2-P43	35.6	0.6	59504.1	48371.7	774.1	0.0	4.6 / 649.0	188.4 / 644.4	384.4 / 456.0	0.4 / 773.7	4.4 / 7741.4
8	GG2-P2	0.5	1	666.0	627.3	8.5	0.0	0.2 / 4.1	2.5 / 3.9	0.0 / 1.4	0.6 / 7.5	2.4 / 77.4
9	GG2-P7	43.4	0	80757.3	71986.4	947.4	1.3	23.4 / 335.2	64.9 / 311.8	170.6 / 246.9	1.0 / 947.1	6.4 / 9477.4
10	GG2-P10	0.6	1	1821.4	1785.6	15.3	0.0	0.0 / 5.8	3.9 / 5.8	0.2 / 1.9	0.5 / 14.3	3.3 / 146.3
11	GG2-P13	0.6	1	964.9	867.4	38.0	0.0	0.3 / 13.0	3.0 / 12.7	4.9 / 9.7	0.5 / 37.0	4.3 / 374.3
12	GG2-P32	59.2	0.5	111345.5	93182.5	1332.4	3.5	48.3 / 1194.8	144.6 / 1146.5	922.9 / 1001.9	4.1 / 1332.3	4.5 / 13327.5
13	GG2-P35	47.7	0.2	40501.7	32200.9	924.4	4.4	7.4 / 721.2	65.3 / 713.8	588.2 / 648.5	1.0 / 924.2	5.9 / 9247.9

14	GG2-P41	42.3	0.8	44476.7	37761.5	875.0	16.7	9.8 / 453.1	17.6 / 443.3	364.9 / 425.7	0.4 / 874.4	5.6 / 8749.6
15	GG2-P68	31.5	1	31856.2	26970.7	772.0	0.0	126.2 / 401.8	109.5 / 275.6	140.7 / 166.1	1.2 / 771.5	4.8 / 7719.8
16	GG2-P70	8.1	1	9598.9	8165.3	170.6	0.4	23.5 / 61.8	3.4 / 38.3	24.0 / 34.9	0.8 / 169.6	5.4 / 1701.4
17	GG2-P79	1.6	1	1637.7	1475.2	62.4	0.0	2.2 / 19.0	7.3 / 16.8	3.8 / 9.5	0.7 / 61.4	4.1 / 618.1
18	GG2-P86	0.4	1	702.8	680.2	13.4	0.0	0.0 / 5.5	3.2 / 5.5	1.0 / 2.3	0.4 / 12.4	3.3 / 127.3
19	Ladder3	0.9	1	9275.6	9195.8	34.9	0.0	0.2 / 10.6	3.9 / 10.4	1.5 / 6.5	1.1 / 33.9	2.4 / 341.4
Totals		**277.5**	**15.8**	**400170.6**	**340084.5**	**6058.8**	**26.3**	**249.5 / 3910.3**	**632 / 3660.8**	**2614 / 3028.8**	**14.6 / 6043.7**	**70.8 / 60507.8**
Success Rate								**0.06381**	**0.17264**	**0.863**	**0.002416**	**0.0012**

Table F-3 – With 'Reproduction By Crossover and Repair By Mutation' Configuration

Appendix G

Publications

This appendix contains selected publications by the author from conferences and journals directly relating to the topics discussed in this thesis. References to the titles are presented below.

Page 174:

R. Cant, J. Churchill and D. Al-Dabass, 2002. "A Hybrid Artificial Intelligence Approach With Application to Games" In *Proceedings of The World Congress on Computational Intelligence* (2002-WCCI), IEEE Int Joint Conf on Neural Networks (IJCNN02), 12-17 May 2002, Honolulu, Hawaii, pp1575-80, ISBN 0-7803-7278-6, and ISBN 0-7803-7281-6 (CD-Rom).

Page 182:

J. Churchill , R. Cant and D. Al-Dabass, 2004. "Using Neural Network Techniques Within A Go Playing Program" In *Proceedings of CGAIDE 2004 International Conference on Computer Games: Artificial Intelligence, Design and Education*, 8-10 November 2004, Reading, England, pp187-192, ISBN 09549016-0-6.

Page 189:

J. Churchill , R. Cant and D. Al-Dabass, 2004. "A Hybrid Genetic Alternative to Game Tree Search in Go" In *Proceedings of United Kingdom Simulation Society Conference*, 29-31 March 2004, St Catherine's College, Oxford, England.

A HYBRID ARTIFICIAL INTELLIGENCE APPROACH WITH APPLICATION TO GAMES

RICHARD CANT, JULIAN CHURCHILL, DAVID AL-DABASS

Department of Computing and Mathematics
The Nottingham Trent University
Nottingham NG1 4BU.
Email: richard.cant/david.al-dabass@ntu.ac.uk

Abstract: We describe a hybrid Artificial Intelligence (AI) approach combining soft AI techniques (neural networks) and hard AI methods (alpha-beta game tree search), in an attempt to approximate human play more accurately, in particular with reference to the game of Go. The program is tested and analysed by play against another Go playing program and it is shown that the use of hard AI enhances the performance of the soft AI system and vice-versa.

Keywords: Neural Networks, Alpha beta search algorithms, Computer Go.

INTRODUCTION

This paper investigates the combination of hard and soft Artificial Intelligence (AI) techniques and the application of such a combination to games. This paper concerns itself with applying the problem to the game of Go. Current Go playing programs have been markedly less successful than their chess playing counterparts. Whilst success in playing chess has come from a move away from attempting to copy human play, this approach has failed in the field of Go. In the present paper we explore an idea inspired by human modes of thought. We attempt to combine neural network techniques and traditional game tree search with some degree of success, allowing experienced based move selection to be used with rigorous analysis in an efficient and effective way.

What is Go?

Go is a relatively simple game the complexity of which emerges as you become familiar with the ideas presented. A comparison with Chess is often made, as these are both board-based games of zero-chance [2]. The rules are simpler in Go, however the board is larger and due to the unrestrictive nature the rules there are many more moves available for the Go player to consider.

The game is played on a board, which has a grid of 19x19 intersections. Two players, black and white, take turns to place a single stone on any unoccupied intersection, with the aim of surrounding as much territory as possible. A player can pass at any turn (giving one point to his opponent) instead of placing a stone. Capturing the opponent's stones is also used to increase a player's score. A stone is captured when the last of its liberties is removed. A liberty is an empty intersection directly next to the stone. Suicide is not allowed unless it is to capture some opponent's stones.

The end of the game is usually reached by mutual agreement between the players, when they both pass consecutively. Stones which are effectively dead and territory points are then totalled up and the winner declared.

CURRENT RESEARCH

Neural Networks

The inspiration behind the neural network idea is the simulation of neurons in a biological brain. Biological neurons receive stimulus signals from other neurons and when a certain activation level is reached the neuron fires signals to all the other connecting neurons. The change in the strength of the connections is dynamic and it is this particular feature that means networks of neurons simulated on a computer can be trained to recognise patterns of input and give appropriate patterns of output [3].

Hard And Soft AI

For the purposes of this project it is necessary to define two ideas to allow us to discuss the hybrid approach used.

Hard AI refers to the more traditional artificial intelligence techniques such as the various tree search methods, pattern matching and rule based expert systems. The term 'hard' is used to emphasise the predictable and exact nature of such methods.

Soft AI techniques on the other hand deal with methods that rely on statistical processes and may produce results with a probabilistic interpretation. This is one of the benefits of these methods since often in real life problems there is not a single 100% correct answer to be found. Methods in this category include neural networks and other machine learning processes.

State Of The Art

The computer Go programming community is relatively small compared to computer Chess but interest in the topic is building now that computer chess has reached such a high level of success. The central hub of this community can be found at the computer Go mailing list and the computer Go ladder within which programmers can enter their attempts in an ongoing tournament with most of the best programs available, including commercial programs [4,5]. The low number of entrants, around 20, perhaps reflects the difficulty of completing a Go playing program that can at least play competently at each stage in a game of Go.

Many Faces Of Go

One of the best Go programs around is called Many Faces Of Go. This program uses traditional, hard AI techniques in combination with specialist knowledge databases and rule-based expert systems which are used with a deep game tree searching mechanism and a highly tuned evaluation function. The weakness of this approach is that it is totally dependent on the accuracy and completeness of its internal database. Should a flaw be discovered by its opponent then manual reprogramming would be required to correct it.

NeuroGo

One program that has been developed using neural network techniques is called NeuroGo [6]. The method used by NeuroGo when using the neural network is to reduce the board position into strings and empty intersections. The relationships between these units are used to construct a neural net. The training target was determined by the Temporal Difference learning algorithm [15].

Several experts concerning relations between points and features on the board, relevant to determine the next move, are used in conjunction with the neural network. This is effectively feeding knowledge about important concepts in Go directly into the network, rather than attempting to make the neural net 'discover' these concepts itself.

A NEW APPROACH

The foundation idea for this project was to use a neural network (NN) to suggest plausible moves and then pass these to a game tree analyser to search the moves using a traditional game tree search method. The neural network would thus act as a replacement for the expert systems used in Many Faces Of Go. The use of a NN for selecting possible moves should be fast and also allows the opportunity to expand the NNs knowledge in an automated fashion.

Combination Of Hard And Soft AI Techniques

By attempting to use neural networks with game tree search this project is bringing together hard and soft AI. The main purpose behind this is to take the best features of each component and combine them to produce something that performs better than either of the two components separately. In the case of game tree search the advantage of using it comes from its ability to look ahead into the probable results of playing a particular move. The disadvantage is that it can be very resource intensive and inefficient.

The advantage of using neural networks is that they can provide some knowledge, learnt from experience, of actually playing the game. The disadvantage is exactly the benefit that game tree search provides, in that neural

networks work with the position of the board as it is. They aren't directly able to handle looking ahead and considering the consequences of playing a particular move explicitly.

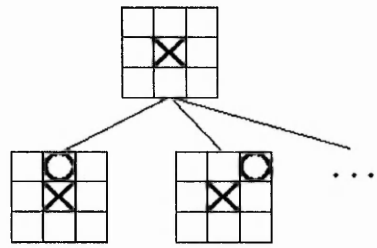


Figure 1: A Partial Game Tree for Tic-Tac-Toe

To summarise, the use of hard and soft AI techniques combined in a carefully thought out manner can negate the disadvantages of both and accent the advantages to produce an overall benefit to the system that it would not have been able to get through the use of only one or other of the classes of methods.

Application To Standard Game Tree Search

A game tree search works by taking a root node, usually the current board position and then generating all possible lines of play from the root until a desired depth is reached or until some other cutoff criterion comes into play. Sometimes all possible moves are used, but it is common to cut this down by removing illegal and obviously bad moves from the tree. The terminal nodes in a game tree are assigned a score based upon some static analysis of the board position at that node. The scores from all the terminal nodes are filtered up the tree toward the root node, using a technique known as Minimax [12]. This way the best move, the one that maximises the player's score, regardless of the opponent's play, can be found. However the terminal node board scoring method must be fairly accurate else the minimax values will be irrelevant, and for many games the computational power required to generate a game tree large enough to produce a conclusive result is enormous. For instance the search space size of Go has been estimated at around 10^{170}, compared to Chess, which is about 10^{40} [1].

In a standard game tree search much processor time will be spent on generating nodes for the next level of the tree but a neural network can be used to quickly and efficiently suggest the most plausible moves given a board position and this can be used at every level of the game tree, assuming the neural network is trusted enough to suggest reliable and high enough quality moves. Ideally around 6 moves would be used from the suggestions of a neural net interfaced to a minimax game tree algorithm, allowing a depth of about 7 or 8 ply to be reached with out too much stress on the system resources.

SOFTWARE/HARDWARE DEVELOPMENT

The software system was developed with the following aims in mind:

- To provide facilities for the program to communicate with other Go playing programs. A generic neural network program should be developed to create and train neural nets.
- The generic application should be expanded and applied to Go.
- To process training data for the network an automated procedure should be developed.
- Utilities must be provided for board position evaluation, standard game tree search and methods for gathering Go specific information about the board position such as the number of liberties a string of stones may have.
- A graphical user interface should be provided for ease of use.

For communications the Go Modem Protocol was used [17]. An alternative to GMP which is currently under debate is called the Go Text Protocol, GTP [9].

The game tree search algorithm implemented is specifically a version of the classic Minimax search with alpha-beta pruning, called MTD(f) [11]. This is currently one of the best performing and most reliable Minimax type search algorithms developed. It was intended that this search algorithm should use the neural network module to supply moves for expanding the tree. Iterative deepening was also used, as this appears to be the most efficient method of expanding search trees. Several enhancements to the Minimax algorithm were also used

including a transposition table, best move first and Enhanced Transposition Cutoffs, ETC [14]. Best move first is straightforward and means that, through the use of the transposition table, the best move at a branch in the search tree, from a previous search, will always be checked before other candidate child nodes. ETC is also simple, involving a simple check when a node is generated to see if the same node had previously caused a cutoff in another search and if so to check the cutoff condition again. Both of these additions allow for a considerable performance boost to the standard Minimax algorithm, allowing much larger and faster searches to be performed. A simple evaluation function for the Minimax search was used, just a liberty count and comparison for both sides.

Design Issues

After some experimentation with different sized neural networks I found that a 3-layer network with 81 input neurons, representing 9x9 sections of a Go board centred on the proposed move, appeared to be the most useful. Only 1 output neuron was required to give a plausibility score for the move fed into the network and an arbitrary number of hidden neurons (in this case 45 neurons) were included. The highest scoring moves were selected to be used in the game tree search.

The back propagation algorithm for training was decided upon, mostly because of its general applicability to a wide range of problems [3]. Training data was acquired as a collection of professional tournament games in SGF format from the Internet [16].

An idea that occurred later in the project was of varying the granularity of consideration of the networks. Just as a human player might, the idea was to look at the board and narrow down an area in stages and eventually identify the best move to play. Until now the program had been designed to look at as many points on the board, and their neighbours in relation, as time and resources allowed. This new method would allow a succession of networks, trained to varying levels of detail to be used. And will be given some consideration later in this paper.

IMPLEMENTATION

A simple GUI was developed which allows the user to control the neural network training and development. A Go board is present and games can be played against the program with it. The GUI is also intended to be used so the user can watch a game in progress between the program and another program via GMP or another protocol.

A useful test feature developed in the program was the game walkthrough test, which allowed the user to play through a professional game of Go that has been stored in Smart Game Format, SGF, the format the training data was initially found in [10]. The program itself makes suggestions and comparisons are made between the program and the actual moves made. Lots of information is generated at each move and outputted in a log window. Graphically, the board is shown and potential moves are coloured according to the plausibility score assigned by the neural network. This allows the user to get a general idea of the level of skill the network has learnt to. Code to create and maintain GoString information was included, which monitors which stones belong to each string, what colour the string is and how many liberties the string has. This greatly increases the speed when detecting and removing captured groups.

The distilled experience of 36 professional tournament games was used in constructing a training database, producing over 50,000 training pairs, which pushed resource constraints to the limit. Due to the training set actually being fairly small in experience content, the networks were in danger of over training and becoming too specialised on the set of games used to produce the training set. Several issues concerning resources still have to be addressed before these problems can be overcome.

A coarser grained Area Finder network was trained for a reasonable amount of time and started to show some signs of sensible suggestions concerning the area of board to play in. Two restricted move range networks were trained, both showing promising results. The first covered the first five ply moves and second covers from five ply to ten ply. An advantage of using restricted ranges is that the training is much quicker and the networks gain a higher degree of skill in their area than a more generalised network ever would.

The MTD (f) variation of the alpha-beta minimax search algorithm was implemented along with the aforementioned enhancements. The expand function which creates child nodes given a position uses the 9x9 neural network to suggest a specified number of the most plausible moves as child nodes. An alternative evaluation function for the Minimax search that was not implemented but could be a future path of research involves using a TD (λ) trained neural network as an evaluation function. In fact this approach appears to be the

most promising method of implementing an effective evaluation function, judging from research done by other parties [13].

RESULTS & DISCUSSIONS

Looking at how long it takes various configurations to reach a move decision shows us some important points and in combination with some quality of result analysis can tell us to what degree the combination of soft and hard AI has been successful and if it is worth pursuing in the future.

Configuration	Average Time Taken Per Move (seconds)
1. 9x9 Neural Network	0.784
2. 9x9 Neural Network + Area Finder	0.136
3. Alpha-Beta (Liberty Count)	40.712
4. 9x9 Neural Network + Alpha-Beta (Liberty Count)	9.02
5. 9x9 Neural Network + Alpha-Beta (Liberty Count) + Area Finder	1.56

TABLE 1 – SYSTEM CONFIGURATION SPEEDS

First of all, looking at table 1, the difference between configuration 1, which used just a 9x9 network and configuration 2 which used a 9x9 network with an Area Finder network is easily explainable. Using the coarser grained Area Finder network divides the board into 9 sectors and given the full 19x19 board selects the most appropriate sector out of the 9. Then the 9x9 network looks at all legal moves within that sector. For the first configuration all legal moves in the entire 19x19 board must be considered so we see a logical time difference of around a factor of 9. A similar affect is seen when comparing configurations 4 and 5. The use of the Area Finder network gives a substantial speed boost without adding any unreasonable overheads. If the quality of suggestions presented by the Area Finder network can be measured and built upon then this could be an effective and efficient method of incorporating neural network technology within a Go playing program. The use of alpha-beta search added considerable computational overheads, however that is expected considering the nature of the algorithm. The liberty count evaluation function was used in all cases.

To assess and compare the quality of the neural networks and different configurations involving either or both soft and hard AI two approaches were taken. The first method was used to determine the extent and level of training achieved by the neural networks. Several measures were used and information gathered about 6 different networks is displayed in table 2. The statistics were collected as each network played through the same professional game; the average time to select a move was recorded, as was the average rank of the actual move within all the moves considered by the neural networks. To give a more detailed look at the quality of the moves being selected the percentage of time that the actual move was ranked in certain percentiles was also noted, for example for the 5x5 network the actual move was tanked in the top 10% of moves 19.2% of the time and was tanked in the 20% of moves 36% of the time. For comparison sakes a newly created, hence untrained, network was tested also, so we should expect, if training has worked at all, that the new network should have the lowest scores. Looking at the results the first thing of note is that the larger the network the longer it takes to suggest a move. This is quite expected and reminds us that even though neural networks are fast compared to other AI techniques they can still easily build up a substantial overhead that must be kept in mind and minimised wherever possible. The best performing network appears to be the 5x5 with the actual move being ranked in the top 30% of moves just over half of the time.

Neural Network	Average Time Per Move	Percentage of time that actual move is in top n percent				
		10%	20%	30%	40%	50%
5x5	0.576	19.2%	36%	50.4%	59.2%	64.8%
7x7	0.736	16.8%	29.6%	41.6%	60%	65.6%
9x9	0.936	12%	30.4%	39.2%	52.8%	64%
Copy of 9x9	0.912	18.4%	34.4%	47.2%	56%	68.8%
11x11	1.256	9.6%	19.2%	30.4%	35.2%	44%
13x13	1.664	18.4%	28%	36%	51.2%	61.6%
Random 9x9	0.824	4.8%	20.8%	28.8%	35.2%	44.8%

TABLE 2 – NETWORK PERFORMANCE STATISTICS

Comparing all of these figures to the random 9x9 network shows that they all improved after training and reveals an interesting and very important point about over training. The 'copy of 9x9' network was an earlier version of the current 9x9 and has much better figures. This does suggest rather strongly that the current 9x9 has been over trained and the quality of its output has been degraded as a result. This also implies that a peak of training can be reached and by considering the figures they also suggest that the peak does not mean getting the very best move at rank number one. Rather it suggests, perhaps viewing it optimistically, that the network realises there may not be one perfect move but maybe lots of good moves and using a neural network allows the moves to be ranked effectively as opposed to selecting a single best move. This may be the strength of using neural networks in this instance. A rather large hindrance that should be kept in mind is the time it takes to train a network. The 9x9 took around 2 months to reach a point where it started to over train. Larger networks and bigger training databases will take proportionately more time. This meant there was not a lot of spare time for experiments and trying alternatives, so I think this will be a stumbling block for quite a while in the future.

With the second quality measure the configurations used for timing an average move were used to play proper games of Go against GNUGo 26b [7]. The results are presented in table 3. It is important to play actual games since this was the original intention of creating such a program and is really the best way of judging its success in its intended environment. The program itself still unfortunately has a few problems and bugs that were given special provision. The program did not have any method to decide when to pass, so a game would continue until GNUGo passed or until a crash occurred.

Configuration	Game Score (we play black), (J= Japanese, C= Chinese)
1. 9x9 Neural Network	J: B-8, W-49 C: B-106, W-146
2. 9x9 Neural Network + Area Finder	J: B-3, W-21 C: B-54, W-71
3. Alpha-Beta (Liberty Count)	J: B-9, W-12 C: B-35, W-38 * Not Compete
4. 9x9 Neural Network + Alpha-Beta (Liberty Count)	J: B-9, W-25 C: B-107, W-124
5. 9x9 Neural Network + Alpha-Beta (Liberty Count) +Area Finder	J: B-7, W-18 C: B-55, W-66

TABLE 3 – GAME SCORES

Where a crash occurred it is marked in the results table as N.C. (not completed). When a game has reached an end, either on purpose or by fault, the board was scored by Jago, which functioned as arbiter between the

programs [8]. A final point of note is that the program had no knowledge of the Ko rule and as such could have broken it and forfeited the game, however such a situation did not occur in any of the test games played.

If we look at the results of the test games against GNUGo we can observe several things from the scores. Both Chinese and Japanese scores are presented, with our program playing black for each game. The first thing to note is that where the alpha-beta algorithm was included the score gap has been considerably narrowed. This would imply an overall improvement in defence and offence by the program thanks to the lookahead facilities provided by the alpha-beta routine. It also shows that although only a simple and occasionally probably inhibitory evaluation function was used, the liberty counter, a general improvement in play was fairly easy to achieve. Currently the alpha-beta is limited to a 6 move look ahead but it would probably be beneficial to make this a dynamic factor based on the line of play and resource availability. Unfortunately the plain alpha-beta configuration would not complete an entire game so the scoring is pretty inaccurate and may not reflect the final state of the game had it reached its conclusion. This means we can see that the soft AI, neural networks, benefits from using hard AI, alpha-beta, techniques but we cannot be certain vice versa. It is possible that the neural networks may actually hinder the optimum play of the alpha-beta routine although this seems unlikely. What more we can tell is that the addition of the Area Finder not only gives a speed advantage as discussed earlier but also increases the quality of suggestions. From observation of the games I would suggest that this is partly because a wider area of the board was played across when using the Area Finder than without, so the opponent found it harder to establish solid territories. When the Area Finder was not used the play tended to a single area and usually stayed there throughout the game, allowing the opponent to establish unchallenged territories.

All these results show that soft AI has something to offer the problem area of Go, the limitations and extent of which require further investigation, however we have made some connections, various ideas have been tried and tested and a system that can support further research has been developed and implemented. More than anything else I think, questions have been raised and pointers for promising future investigations have been found. There could be any number of ways to combine soft and hard AI. The trick is to do it in such a way as to maximise the strengths of each and minimise the weaknesses. If in doing so the combination is greater than the parts then the job is a success. From the results it seems clear that hard AI benefits soft AI and it is a pity the reverse cannot be concretely induced from the results collected but it would be reasonable to assume so.

CONCLUSIONS AND FUTURE WORK

Many ideas cropped up throughout this project and given time and appropriate attention there are lots of avenues of research that warrant further investigation. Also the game of Go is such an expansive problem that there are many places where the system could be improved and alternative approaches could be taken. Amongst the best ideas that were considered are using various grain networks, how to combine them and which to use, limited range networks which tend to specialisation and of course other methods of combining soft and hard AI. There are many other soft AI techniques worth looking at, such as genetic algorithms and evolutionary programming and seeing how they could be combined efficiently and effectively with the more traditional hard AI methods. Also there are lots of hard AI techniques not considered here such as rule based systems that could benefit the soft methods, to give direction and guidance when faced with such a vast and chaotic wealth of information, which may be essential to give encourage the system to develop global strategies. Interesting extensions to the ideas could include a closer look at the actual choice of neural network architecture and construction, and a more thorough review of the choices available. A deeper understanding of the way humans perceive and play Go could be developed from further work, perhaps leading to a better understanding of human pattern recognition. Go is an excellent environment that provides many opportunities for developing knowledge, not just of artificial intelligence and how to use it, but also of human intelligence and how the two compare, and it may also provide some impetus for the acceleration of development of soft AI techniques such as neural networks and other machine learning processes.

REFERENCES

[1] Allis, L.V., Van Der Herik, H.J., Herschberg, I.S., "Which Games Will Survive?", Heuristic Programming in Artificial Intelligence 2 - The Second Computer Olympiad, pages 232 – 243, Ellis Horwood, 1991.

[2] Burmeister, J, "An Introduction to the Computer Go Field", and Burmeister, J., Wiles, J., 1995, "Associated Internet Resources", Available on the Internet at http://www2.psy.uq.edu.au/~jay/go/CS-TR339.html

[3] Callan, R., "The Essence Of Neural Networks", Prentice Hall, 1999.

[4] Ladder 2001, "Computer Go Computer Go", See http://www.cgl.ucsf.edu/go/ladder.html

[5] Computer Go Mailing List, 2001, See

http://www.cs.uoregon.edu/~richard/computer-go/index.html.

[6] Enzenberger, M, "The Integration of a Priori of Knowledge into a Go Playing Neural Network", 1996, Available on the Internet at http://www.uni-muenchen.de.

[7] GNU Go, latest version can be found at http://freedom.sarang.net/software/gnugo/beta.html

[8] Grothmann, R, "Jago", 2001, available on the internet at http://mathsrv.ku-eichstaett.de/MGF/homes/grothmann

[9] "GTP2001", Go Text Protocol information can be found 2/3 down the page at http://freedom.sarang.net/software/gnugo/beta.html.

[10] Hollosi, A, "SGF User Guide", 1999, available on the Internet at http://www.red-bean.com/sgf/user_guide/index.html.

[11] Plaat, A, "MTD(f), A Minimax Algorithm Faster than NegaScout", 1997, available on the Internet at, http://www.cs.vu.nl/~aske/mtdf.html.

[12] Owsnicki-Klewe, B, "Search Algorithms", 1999, available on the Internet at http://www.informatik.fh-hamburg.de/~owsnicki/search.html

[13] Tesauro, G, "TD-Gammon, a self-teaching backgammon program, achieves master level play", Neural Computation, Vol. 6, No.2, 1994.

[14] Schaeffer, J and A. Plaat, "New Advances In Alpha-Beta Searching".

[15] Schraudolph, N, Dayan, P, Sejnowski, T, "Temporal Difference Learning of Position Evaluation in the Game of Go", Neural Information Processing Systems 6, Morgan Kaufmann, 1994, available on the Internet at ftp://bsdserver.ucsf.edu/Go/comp/td-go.ps.Z

[16] Van Der Steen, J, "Go Game Gallery", 2001, available on the Internet at http://www.cwi.nl/~jansteen/go/index.html

[17] Wilcox, B, "The Standard Go Modem Protocol – Revision 1.0", available on the Internet at http://www.britgo.org/

USING NEURAL NETWORK TECHNIQUES WITHIN A GO PLAYING PROGRAM

JULIAN CHURCHILL, RICHARD CANT, DAVID AL-DABASS

School of Computing & Informatics
Nottingham Trent University
Nottingham NG1 4BU
E-mail: richard.cant@ntu.ac.uk

KEYWORDS
Go, Neural Networks, Artificial Intelligence, Games.

ABSTRACT

Artificial neural network algorithms represent a valid approach to the exploration of search strategies for computer games systems. The oriental game Go presents one of the most challenging search problems in the games domain. A set of experiments using artificial neural networks are presented in this paper to investigate the application of these techniques to the game of Go in particular and other computer games in general.

INTRODUCTION

This paper presents experiments with neural network techniques, designs and methods of use, within a Go playing framework. The idea behind this was to develop a neural network that could be used with a game tree search algorithm to provide a selective full board search for the best move available. The network would be used to assign values to given moves allowing efficient node ordering within a game tree and a method to select branches to prune if required.

What is Go?

Go is a board game with its origins in the Far East. It is a relatively simple game the complexity of which emerges as you become familiar with the ideas presented. A comparison with Chess is often made, as these are both board-based games of zero-chance (Burmeister and Wiles 1995). The rules are simpler in Go, however the board is larger and due to the unrestrictive nature of the rules there are many more moves available for the Go player to consider.

A board of 19x19 intersections is used onto which the two players, one placing black stones the other white stones, take turns to play a single stone of their colour onto an empty intersection. Black plays the first move in a non-handicap game. The main aim of the game is to surround as much territory as possible whilst confining your opponent to a minimum amount of space on the board. Territory is defined as empty intersections that one side or the other is clearly in uncontested control of. A player can pass at any turn instead of placing a stone. Stones may be captured anytime after they have been placed on the board and are removed if they are captured. The captured stones are added to the players score at the end of the game. The concept of liberties play an important part in the process of keeping stones alive or killing them. A liberty is any empty intersection directly adjacent to a stone and can be thought of as breathing space for the stone or stones it is adjacent to. A stone is captured when the last of its liberties are removed. Connected stones of the same colour share liberties and will live or die together. Suicide counts as an illegal move unless it captures some opponent's stones in the process.

The end of the game is reached when both players pass consecutively signalling a mutual agreement that the game has gone as far as it can. Stones that are effectively dead, captured stones and territory points are then totalled up and the winner declared. A small amount of points is usually added to whoever is playing white, since as black plays first in an even game a small advantage is assumed.

The board edges and corners have an important role in the game as they are treated as an imaginary wall of opposing colour stones. This means a lone stone on the edge will have only 3 liberties, as opposed to 4 away from the edges, and in the corner will have only 2.

NEURAL NETWORKS

The inspiration behind the neural network idea is the simulation of neurons in a biological brain. Biological neurons receive stimulus signals from other neurons and when a certain activation level is reached the neuron fires signals to all the other connecting neurons. The change in the strength of the connections is dynamic and it is this particular feature that allows networks of neurons simulated on a computer to be trained to respond to patterns

of input, giving appropriate patterns of output. More information can be found in (Heitkoetter and Beasley 1994; Fausett 1994).

Artificial neural networks can be constructed from simple processing units, artificial neurons, and can be trained to learn complex functions. This seems an appropriate method to use in a Go playing program, since these techniques offer a possible way to capture some of the intuitive move recognition required to play at more advanced levels.

OUR IDEAS

Many uses for neural network methods can be imagined for application to the problem of playing Go. The authors original intended use for a neural network was to score proposed moves given a local board situation, for instance a square area of 9x9 intersections centred on the move to be score. For a complex task such as this there is also the matter of interpreting the output of the neural network. Ideally it would be a score relative to all other possible moves at that point in the game but if it was then it would be easy to write an excellent if not perfect Go playing program.

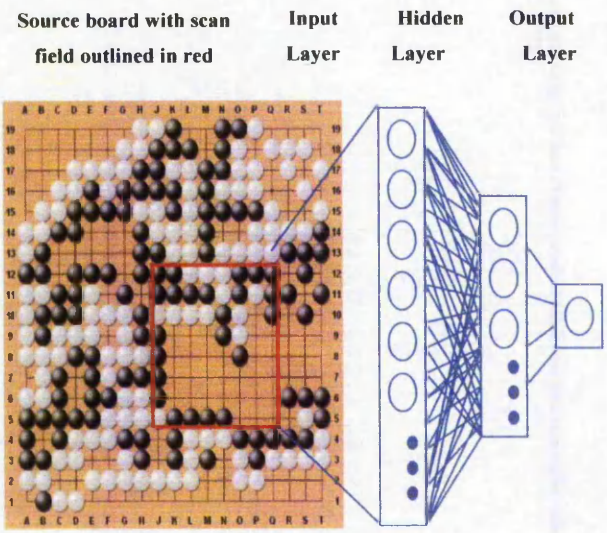


Diagram 1 – Using the neural network

The neural network output must represent some measure of the usefulness of the move at this point in the game or the training would not improve, so we choose to call this output a plausibility score, to distinguish it from the idea of a global absolute 'one move is better than another' score. Diagram 1 illustrates how the network would be used in practice.

A Neural Network as a Move Ordering Mechanism

Game tree search algorithms allow the construction of a tree of possible moves and responses. When coupled with an evaluation function to assess tree nodes the best available move can be deduced within a given depth. This is a good solution for problems that have a very shallow depth and narrow branching factor but as soon as we get into games any more complicated than Tic-Tac-Toe, the limits of game tree search become apparent. To know the very best move at any position in a game of Go would require the search to cover an enormous search space, estimated at around 10^{170} positions (Allis et al. 1991). The number of moves available at each position in a game of Go averages at around 180, combined with a game length of around 200 moves makes a straightforward game tree search impractical.

The order in which moves are generated and processed by the search algorithm can greatly affect the efficiency of the search when the common Minimax algorithm with Alpha-Beta pruning is used. Minimax is a game tree search algorithm for zero-chance two player games. (Russell and Norvig 2003) The efficiency of the Alpha-Beta pruning depends on a good ordering of nodes; the best available moves should be examined as early as possible. One use for the developed neural network would be to enhance a game tree search program by using the network to provide a good estimated ordering.

Input Schemes

Much time was spent on experimenting with different algorithm parameters, input field shapes and sizes, how to use the training data and various network input schemes. The methods for encoding the board area around the move to be scored incorporated and contrasted quite a few different features. Varying input schemes for the neural network were created, to allow comparison of the contribution each feature made towards evaluating a move.

Input Scheme	State Neurons	Distance Neurons	Other Features
0 - Raw	1	0	-
1 - 3 states	3	2	-
2 – Binary distance	3	18	-
3 – Offboard state	4	18	-
4 – Liberties	4	18	Liberties of surrounding strings.
5 – Distance to last move	4	18	Distance to last move.
6 – Stress last move	4	18	Last move emphasised in input.
7 – Binary states	2	0	Binary encoding of states using *ourColour theirColour*
8 – Binary states	2	0	Binary encoding of states using *black* and *white*
9	3	0	-
10	4	0	-
11	4	2	-
12	3	18	-

Table-1 – Input Scheme Details

A lot of information can be extracted from a raw board position. One of the challenges of designing the networks was deciding which features would contribute positively to the network performance and learning capability. Some features impacted negatively on the network, they overbalanced or mislead the network during learning due to the feature being overly strong compared to other features. One such feature found to produce a derogatory effect on the performance of a network trained by other researchers, was the distance to the last move played (Enderton 1991).

It seems an intuitively important feature since play in Go tends to be near the last move played, until one player has built up enough confidence in the local position to play elsewhere on the board. The frequent occurrence of consecutive moves played close to each other in the training data resulted in this feature dominating the network's play, disproportionately scoring moves that were closest to the last move played. Part of the reason for this feature causing such a problem was the fact that these networks had no strategic input (nor indeed was this intended in their design). The networks trained here and by some other researchers were designed to look at a static local position without temporal knowledge, information about the global board situation or therefore any form of tactical or strategic planning. These concepts were to be dealt with by more appropriate mechanisms such as game tree search.

Redundancy within feature sets can easily occur, where one feature implicitly provides the information of one or more other features, so experiments were conducted with networks with varying input architectures to assess the impact of the features. The full set of features available and the actual input schemas used are detailed in Table 1. The experiments mostly focussed on the use of multiple state neurons and the inclusion of distance to the board edge information.

The initial network design had only raw input from the board with a very simple encoding scheme: 1 neuron per point, with the value indicating the board point content as follows: *black*(+1), *white*(-1), *empty*(+2), *offboard*(-2). After this design multiple state neurons per point were used to represent the contents of the board point by selecting only one of these state neurons to be activated at a time. For the input schemes with 3 neurons per board point the states were *ourcolour*, *theircolour* and *empty*. Those with 4 neurons per board point had the addition of an *offboard* state neuron.

The distinction between *ourcolour* and *theircolour* rather than *black* and *white* was made to allow colour reversal invariance to be handled. Just as with rotation and reflection symmetries, colour reversal results in an equivalent position, but from the other colours point of view unlike rotation and reflection. To train the network to deal with this would require presenting the network with a colour reversed pattern for each training pattern, effectively doubling the training time. However solutions are possible to solve this issue during network input pre-processing. One method is to always look at training pairs from blacks point of view, then whenever the network is actually used, training included, the stone colours must be reversed if it is being used to score potential moves for white. The alternative method used here is to always score moves from the point of view of the colour to play – represented by *ourColour*. This is a more appropriate method to use when the input to the network is separated into the possible board point states.

To limit the number of weights and so also the amount of computation time required to use and train the networks, designs 7 and 8 looked at combining the states and encoding them through only 2 neurons. In scheme 7 an encoding used

successfully by other researchers is included (Schraudolph et al. 1994). Using 2 neurons the states are encoded as *ourcolour*(+1,0), *theircolour*(-1,0), *empty*(0,+1), *offboard*(0,-1). For scheme 8 the states *black* and *white* replace *ourcolour* and *theircolour*.

Input schemes 4, 5 and 6 introduce some simple extra features to the input. Scheme 4 adds the liberties of the surrounding strings, providing 4 neurons per direction, north, south, east and west. One of the set of 4 neurons per direction is activated to indicate 1, 2, 3 or >3 liberties for the adjacent string if present. Scheme 5 adds an extra neuron to encode the Euclidean distance to the last move played. Scheme 6 handles this slightly differently by emphasising the input signal of the point that was last played.

ALTERNATIVE IDEAS

The main aim for developing the network was to aid the ordering of moves but other options were also considered, amongst them an area finder network that could identify the most urgent part of the board to play in. Also the problem of moves being scored according to the local situation without considering the relative value of alternative moves was looked at.

The urgency network idea was developed to cope with the problem of having move plausibility scores that related only to the board state with no sense of whether one move should be played before another. A network architecture that made the move scores relative to each other would be much more useful and accurate for the task of ordering the moves. To be able to adjust the scores so the network could output a relative urgency value required it to consider at least two moves at a time. A design was drawn up to train a network that could judge whether one move was more urgent than another. In actual play an appropriate sorting algorithm could be used to order the full set of available legal moves at a given position after using the network.

Two alternative designs were created. The first used the entire 19x19 board as input; introducing two additional neurons per board point to the four board point state input schema and dropping the redundant *offboard* state neurons. The additional neurons were labelled MOVE A and MOVE B, each to be activated for only a single, exclusive board point, the points corresponding to the first and second move to be compared. Two output neurons were specified to allow the network to output a relative score for each MOVE A and MOVE B. This would let the program using the network compare the moves relative to each other. If the scores were not different enough from one another, i.e. both were less than 0.5 or greater than or equal to 0.5, then no confidence in the quality of the relative comparison was assumed. For training, the example games were stepped through taking two consecutive moves of the same colour at each board position and randomising between presenting the first move as MOVE A or MOVE B. This was done so that neither MOVE A nor MOVE B would learn a bias as to which neuron represented the more urgent move. The more recent move was given an output value of 0.9, the later move, 0.1.

The second design took advantage of the previous move scorer network experiments and attempted to use some of the trained weights from those networks that had trained successfully. It used the four state input pattern and so could use a ready trained set of weights from any network that used the same schema. An additional 2 hidden layers of neurons were added to create the urgency network and the first two layers taken from the ready trained networks were frozen, hopefully allowing the network to assimilate information about relative scoring in the last two layers.

The training for this design used a reinforcement scheme. Again two consecutive same colour moves were scored and compared at each ply in the training games. This time if the earlier move didn't score at least 0.2 better than the later move a reinforcement factor of +0.2 was added to the output score and used as the new output training vector for the earlier move. The output score for the later move had a reinforcement of –0.2 added to its score and this was used as the later moves output training vector. Some initial results were gathered from these designs. The first design stabilised quickly with a poor performance score, suggesting that perhaps the network had only learned that the more urgent move was MOVE A 50% of the time and MOVE B the rest of the time. The second design also stabilised very quickly, the extra hidden layers adapting to the ready trained weights fast, however the performance never bettered the performance of the original source network indicating that nothing new was learnt and that the extra hidden layers had merely adapted around the frozen source weights.

EXPERIMENTS AND RESULTS

Unless specified the training data that was used for training all the neural networks was taken from the No Name Go Server (NNGS), a games server where Go players can conduct games in real-time over the Internet. All games played from the year 1995 to 2003 were collected and in all the experiments where the playing rank of the training games was not being varied, games with confirmed player ranks of between 25 Kyu and 5 Dan were

used. The ranks were calculated automatically by the server based on player performance.
Unfiltered this gave a total of 395,972 game records in Smart Game File Format (SGF 1997), giving approximately 80 million training pairs. The results presented here represent some highlights from a larger more comprehensive set of experiments.

Where it is usual to use a fixed size training set, we took the approach of using a non-repeating set of data. Since we had a large number of example games from the NNGS server we felt it was unnecessary to impose this training set size limitation. For comparison, experiments were also conducted with limited sized training sets and the results are compared later in this paper. The validation data set consisted of 10 games in SGF format selected from the Nihon Ki-in Championship, the Tengen Tournament and a few professional level games from NNGS. The networks were tested every epoch, for the purposes of non-repeating training sets an epoch was set to 20,000 training patterns. The value used to monitor the progress of the neural network training was the average rank that the network placed the move actually played, in each game from the validation set, compared to all other legal moves globally available in that position.

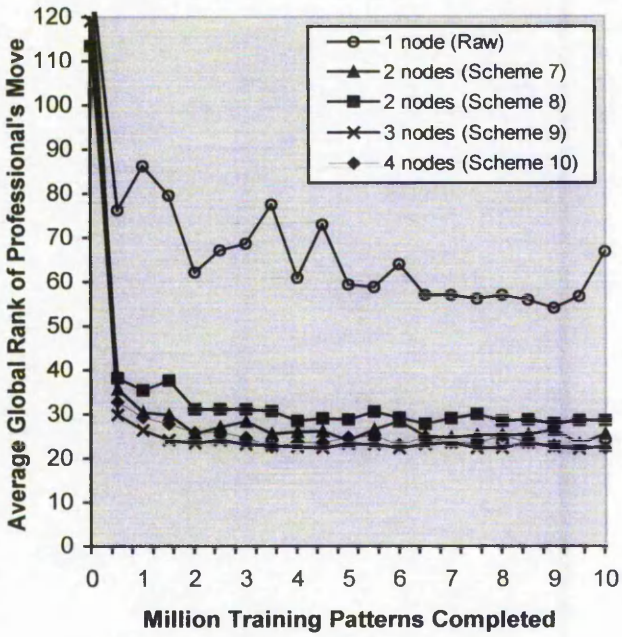


Figure 2 - Comparing the Number of Input Neurons Per Board Point

Figure 1 shows the training progress for network designs with differing state separation policies. The first obvious item to note from these results is that separating the possible states a board point can be allows the neural networks to be trained to a higher quality and a much more stable weight set than by not separating the states. This means that the network saves training time and effort from having to classify the input itself. This is a simple form of input pre-processing which would otherwise require a number of weights or possibly an entire layer of free parameters to be dedicated to this straightforward classification task. Further improvements are apparent as the various states are allocated individual neurons. Of course the extra neurons will cause the network to use more processor time; a balance between accuracy and speed must be found to suit the networks intended use.

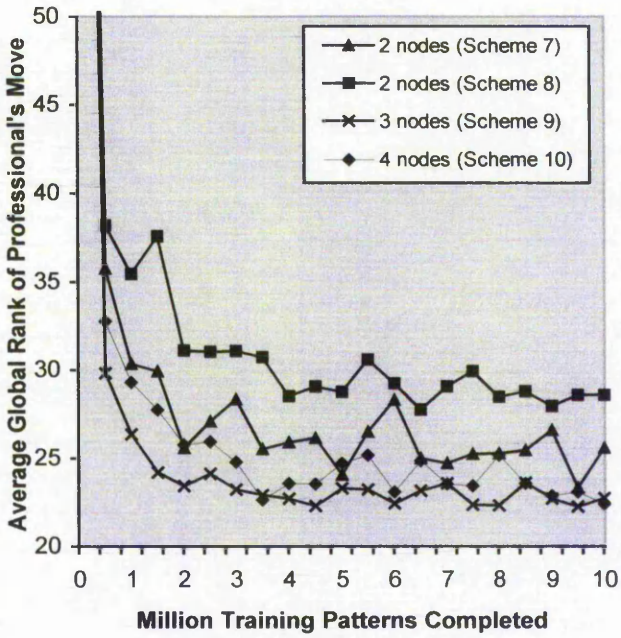


Figure 2 - Detail of Figure 1

Figure 2 shows a detailed view of the more complex input schemes. Scheme 8 with the encoding of the board point contents using 2 neurons, *black*(+1,0), *white*(-1,0), *empty*(0,+1) and *offboard*(0,-1), makes efficient use of the available input neurons and allows properties that some states share to be included, such as both black and white assigning a value to only the first neuron, encouraging the stone present/absent distinction to be learnt for both the black and white states. Surprisingly further improvement is gained from the addition of our method of colour symmetry handling in scheme 7, although it is unclear why this should be so.

The difference between using 3 state neurons or 4 seems to be uncertain, but the similar performance certainly suggests that the extra *offboard* state neuron that scheme 10 provides may be redundant. Without an explicit state neuron the *offboard* state would be implicitly encoded by not activating any

of the other state neurons. From figure 2 it appears that this is enough for the network to take advantage of the properties of the board edges.

The graph in figure 3 displays a comparison between training over fixed size training sets and a non-repeating set. From this it can be observed that the non-repeating set clearly allowed the neural network to train to a higher standard than the fixed size sets. It can also be noted that the size of the fixed size training sets directly correlates with the training performance. A larger training set produces better results. The actual sizes of the sets used were 20,000 patterns for the small set, 100,000 for the large set and 140,000 patterns for the very large set. It also appears that a limit to the performance of a repeated set, given a large enough size, could be reached and that the non-repeating set either exceeded or at least equalled that limit.

Of further note is the appearance of overfitting, a common phenomenon when training neural networks. This may be observed in the gradual performance degradation shown by the small set after about 1 million training patterns have been processed. It might be conjectured that the use of a non-repeating training set prohibits the occurrence of over fitting since the training is not being restricted to an artificial subset of the problem.

The sheer mass of possible positions in Go lends itself towards a non-repeating set where it will be large enough to encompass as many possible situations as possible and as with any learning-by-example problem a well chosen training set should improve performance further.

CONCLUSIONS AND FUTURE WORK

The results presented in this paper compare some different network architectures and training methods relevant to training a neural network to evaluate a potential Go move given some information about the local board area. Several networks were trained that could be used in a Go playing program to increase the efficiency of game tree search mechanisms already in place.

The neural network experiments revealed that under the conditions described in this paper, separating the possible board point states and using individual neurons to indicate the condition of each point could increase the training performance. A performance increase was also noted when using a non-repeating training set, as opposed to a traditional finite sized set. This may be a useful alternative to creating a well-chosen set of training patterns, when a large amount of well spread domain data is available.

The results of further experiments currently being conducted to determine the value of combining this hard, algorithmic, game tree search approach with more flexible machine learning methods represented by neural networks will be presented in a future publication.

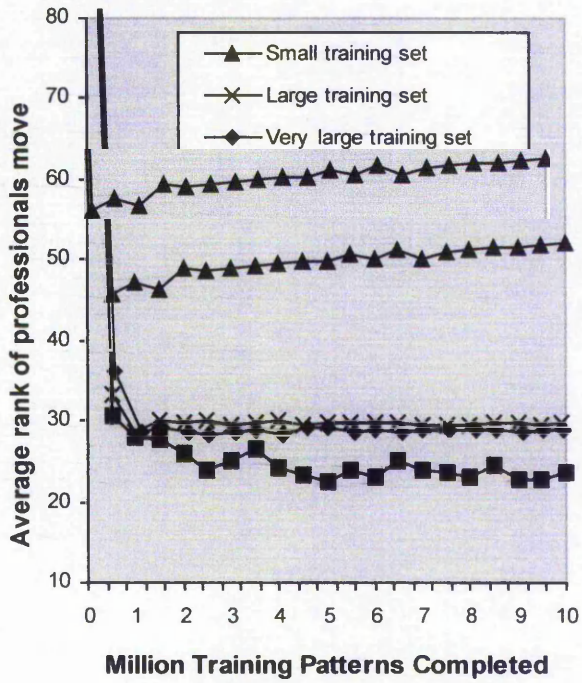


Figure 3 - Using Repeated Training Sets

REFERENCES

Allis, L. V.; H. J. Van Der Herik and I. S. Herschberg. 1991. "Which Games Will Survive?". In *Proceedings of Heuristic Programming in Artificial Intelligence 2 – The Second Computer Olympiad*, pages 232 – 243, Ellis Horwood.

Burmeister, J. and J. Wiles. 1995. *"An Introduction to the Computer Go Field and Associated Internet Resources"* Available on the Internet at http://www2.psy.uq.edu.au/~jay/go/CS-TR-339.html

Churchill, J.; R. Cant and D. Al-Dabass. 2004. "A Hybrid Genetic Alternative To Game Tree Search In Go", *UKSIM2004. Conf. Proc. of the UK Simulation Society*, St Catherine's College, Oxford, 29 - 31 March 2004, pp156-161, ISBN 1-84233-099-3.

Churchill, J.; R. Cant and D. Al-Dabass. 2003. "Genetic Search Techniques for Line of Play Generation in the Game of Go", Game-on 2003, 4th *International Conference on Intelligent Games and Simulation*, IEE Savoy Place, London, November 19-21, 2003, pp233-237, ISBN 90-77381-05-8.

Enderton, H. 1991. "The Golem Go program" *Technical Report CMU-CS-92-101*, School of Computer Science, Carnegie-Mellon University. Available on the Internet at ftp://www.joy.ne.jp/welcome/igs/Go/computer/golem.sh.Z

Fausett, L. 1994. *"Fundamentals of Neural Networks: Architectures, Algorithms and Applications" Chapter 6*, Prentice-Hall, ISBN 0-13-334186-0.

Heitkoetter, J. and D. Beasley, eds. 1994. *"The Hitch- Hiker's Guide to Evolutionary Computation: A list of Frequently Asked Questions (FAQ)"*, USENET : comp.ai.genetic. Available via anonymous FTP from rtfm.mit.edu:/pub/usenet/news.answers/ai-faq/genetic/ About 90 pages.

NNGS, No Name Go Server. Additional information can be found at http://nngs.cosmic.org/

Russell, S. and P. Norvig, 2003. "*Artificial Intelligence – A Modern Approach*", 2nd edition, pp165-171, Prentice Hall.

Schraudolph, N.; P. Dayan and T. Sejnowski. 1994. "Temporal Difference Learning of Position Evaluation in the Game of Go". In *Advances in Neural Information Processing Systems 6*, Morgan Kaufmann. Available on the Internet at http://www.gatsby.ucl.ac.uk/~dayan/papers/sds94.pdf

SGF, Smart Game File Format, FF[4]. 1997. Specification by A. Hollosi. Additional information can be found at http://www.red-bean.com/sgf/

A HYBRID GENETIC ALTERNATIVE TO GAME TREE SEARCH IN GO

JULIAN CHURCHILL, RICHARD CANT, DAVID AL-DABASS

School of Computing & Technology
The Nottingham Trent University
Nottingham NG1 4BU
julian.churchill@ntu.ac.uk, richard.cant@ntu.ac.uk

Abstract: An alternative game tree search method is presented using a novel genetic algorithm. This is enhanced by the inclusion of Go specific knowledge learnt with neural network techniques developed from previous research. This hybrid algorithm is compared to a traditional alpha-beta search method, MTD(f), and a series of tests and results are presented.

Keywords: Go, Genetic Algorithm, Neural Networks, Artificial Intelligence, Games.

1. INTRODUCTION

This paper presents an alternative to traditional game tree search techniques using a combination of genetic methods and neural networks to generate lines of play. In particular, the paper will concentrate on applying the search techniques to the game of Go. Currently, Go playing programs have been markedly less successful than their chess playing counterparts. Whilst success in playing chess has come from a move away from attempting to copy human play, this approach has failed in the field of Go

1.1 What is Go?

Go is a relatively simple game the complexity of which emerges as you become familiar with the ideas presented. A comparison with Chess is often made, as these are both board-based games of zero-chance [Burmeister et al 1995]. The rules are simpler in Go, however the board is larger and due to the unrestrictive nature of the rules there are many more moves available for the Go player to consider.

The game is played on a board of 19x19 intersections. There are two players, one taking the black stones, the other the white stones. They take turns to place a single stone on any unoccupied intersection, with the aim of surrounding as much territory as possible. A player can pass at any turn instead of placing a stone. Capturing the opponent's stones is also used to increase a player's score. A stone is captured when the last of its liberties is removed. A liberty is any empty intersection directly adjacent to the stone. Suicide is an illegal move unless it is to capture some opponent's stones.

The end of the game is usually reached by mutual agreement between the players, when they both pass consecutively. Stones that are effectively dead and territory points are then totalled up and the winner declared. Figure 1 shows a board snapshot during a game of Go.

1.2 Previous Search Techniques

Traditional Alpha-Beta search techniques, as used successfully in other games suffer from two problems when applied to Go.

Firstly the potential depth and breadth of the search is substantially larger than in other games. The number of legal moves available can be 250-300 or more (roughly a factor 10 larger than in chess for example) leading to a very rapid expansion in the number of positions that need to be examined as the depth increases. The depth itself can be substantial, a simple ladder sequence, which even a novice human player can read out, can contain up to 70 moves. It follows that any search algorithm that is to be successful must have a very effective and flexible move selection function, able to narrow the search dramatically when a long forced sequence occurs and yet take account of a wide range of possibilities at other times.

Secondly there is no fast and simple evaluation function that can be applied throughout the game. In Chess it is usually sufficient simply to count material with any positional evaluation being used only as a tiebreaker. In Go the capture of stones is of minor significance unless the number involved is very large (which is rare) or the stones have a particular strategic importance (which requires a positional evaluation to determine).

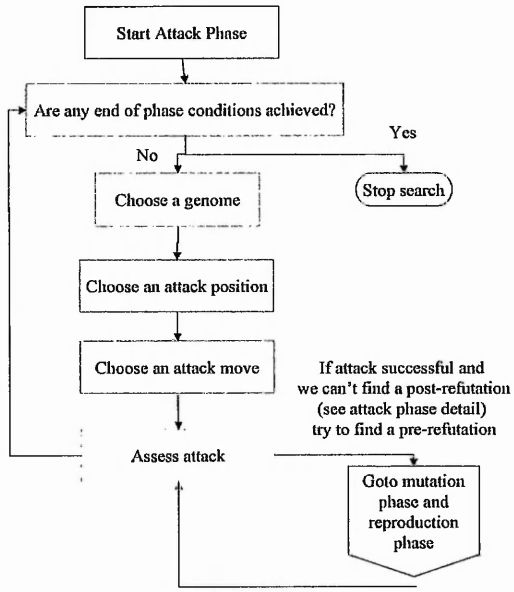


Figure 1 Attack Phase Overview

To make things worse the actual capture of stones is often omitted since once their fate is inevitable it is undesirable for either side to actually play out the final moves in the sequence. The obvious alternative is to count territory but this is an unreliable measure except near the end of the game. At earlier stages the territories are only loosely defined and may often be invaded or reduced. The ability of a player to make such an invasion (or build more territory of his own) depends in turn on the strategic attributes of influence and thickness. Whilst these concepts are usually assessed simply by looking at the superficial pattern of stones on the board this gestalt evaluation needs to be backed up by tactical analysis if it is to be relied on. The judgement of exactly where such analysis is needed is a key skill for human players.

In the present paper we will avoid the complexities of a general evaluation function by confining ourselves to situations in which a specific goal (for example to capture a group of stones) can be used instead. The ability to determine when such a goal is achievable can form an important component of a general evaluation system. Even with this restriction there are problems with evaluation since the actual removal of stones from the board may require many moves beyond the point where their fate is obvious to a human player.

To overcome the limitations of traditional minimax algorithms when faced with such a large search space we propose a genetic algorithm model.

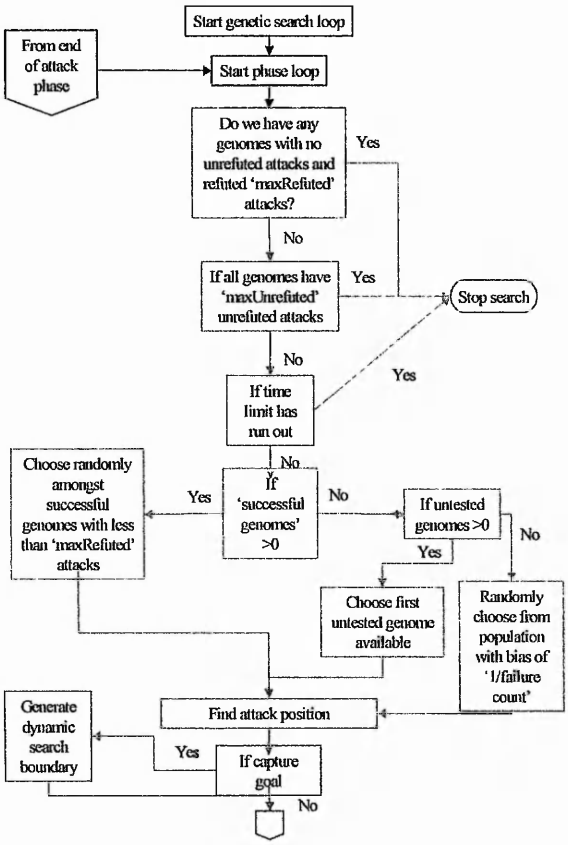


Figure 2 Attack Phase Detail Part 1

2. GENETIC SEARCH ALGORITHM

Human Go players seem to be able to cope with the large search space and find optimal lines of play without exploring more than a tiny fraction of the variations that are possible. Moreover they seem to be able to cope with long sequences combined with a significant number of different branches at certain key points. Furthermore they sometimes try the effect of moving a whole sequence of moves from one context to another. Based on these observations we have devised a genetic algorithm. There have been other attempts to use genetic techniques in the domain of Go amongst them an attempt to combine temporal difference learning [Sutton 88] and genetic algorithms to evolve an evaluation function [Rutquist 00]. This algorithm is designed to

generate lines of play that human Go players might devise, given an initial board position.

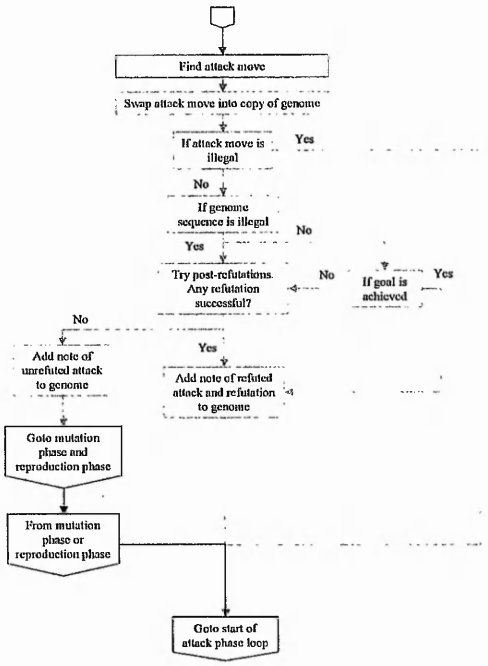


Figure 3 Attack Phase Detail Part 2

We attempt to model the human players move selection and line of play construction using a combination of artificial neural network and genetic algorithm methods. A line of play is represented as a genome in the algorithm and moves in the proposed lines of play as genes. The move selection is aided by previous work into neural networks to score available moves according to how plausible they appear given the local board situation [Cant et al. 2002]. The neural network used during the construction of this genetic algorithm shows signs of having learnt some properties of the abstract concept of shape and of simpler reflex action response moves.

The line of play construction and development parts of the algorithm we have created and modified over some time. The most recent version stores the line of play each genome represents as a Directed Acyclic Graph (DAG). At present the algorithm is optimised for capture problems but could be modified to accept other user-defined goals.

The current version of algorithm first generates lines of play as genomes using various rules, for example reduce the liberties of the opposing string. At present there are 4 different auto-generated genome rules. To these genomes, random genomes are created using the neural network to bias move choice until the population size limit is reached. There is no requirement that the genomes must complete the goal. Some initial experiments were performed and it was found the genome generation stage could easily use up all the problem time without giving any advantages over simply accepting all generated genomes.

After generating the population it is repeatedly subjected to attacks either using the neural network to bias the selection or choosing a move on a liberty using a set of selection rules. Now must attempt to refute the attack. There are several methods to create refutations, if one works then we store the attack and refutation and try another attack. In the DAG version these attacks and if relevant, refutations are stored as branches to the main variation in the DAG. However if the attack is not refuted we attempt to find a refutation through genetic methods.

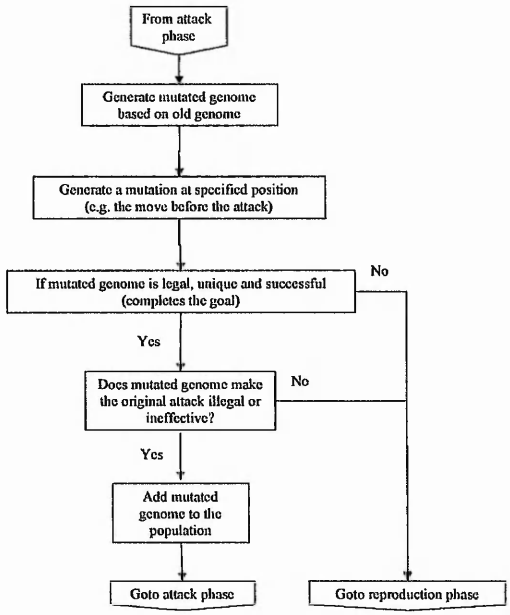


Figure 4 Mutation

The first method to be tried is mutation. The old genome is copied and a random mutation is generated. If the attack is ineffective against the new mutated genome it is added to the population and we continue the attack cycle.

If mutation fails the second method, reproduction is attempted. This is not biased by any fitness calculations as is usual with genetic reproduction. All that occurs is a simple search through the population to find any sections of genome that could combine with the section of the attacked

genome before the attack occurs. This allows genomes to be constructed that could be immune to the attack.

The frequency of mutation or reproduction generating a valid genome is very low for these problem types which is one reason for using more forceful, directed evolutionary techniques. Another reason is that by attacking the genomes at known points in the lines of play we have extra information which we can use to highlight the weak points in the genomes and so direct our efforts to combat these attacks more constructively. This type of information is not usually available in genetic algorithms so a fitness biased mutation/reproduction approach must be used in those cases.

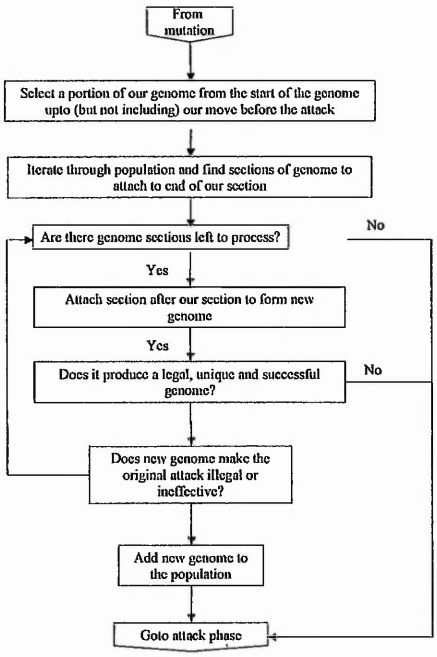


Figure 5 Sexual reproduction of lines of play

This attack phase continues until we must stop the algorithm. This can be due to time constraints or if we have at least one genome with a maximum number of refuted attacks and no unrefuted attacks or until there are no unbeaten genomes left in the population.

The principal variation of each DAG is equivalent to the linear, non-branching line of play stored in previous versions of the genetic algorithm. We chose to use a DAG as the internal representation of the line of play genomes so that future developments to the algorithm could be easily facilitated. It was thought that a DAG roughly represented the structure a human might mentally construct, allowing for multiple branches at key points and the storage of attacks and refutations. This opens up the algorithm to be naturally extended to have attacks against refutations.

A detailed description of the algorithm is shown in the flowcharts in Figures 1 to 5. The flowchart in Figure 1 starts immediately after the initial population has been generated and describes the process of "attacking" the initial lines of play, that is attempting to find enemy moves that prevent the designated goal from being achieved.

Figures 2 and 3 show the detail of the attack and refutation phase. Figures 4 and 5 show the process of creating a new genome to counter an attack. Figure 4 uses mutation whilst Figure 5 introduces "sexual reproduction" to the process.

3. TESTS, RESULTS AND COMPARISONS

We have tested the algorithm using a set of Go problems, mainly taken from "Go Problems for Beginners" [Kano 1990], an elementary instructional text. For comparison we have run the same tests using a state of the art alpha-beta search algorithm known as MTD(f) [Plaat 1997].

Some results are presented in tables 1 and 2. The test names starting with GG refer to the Graded Go Problems for Beginners series of books. GG1 refers to volume 1 and GG2 to volume 2. The remaining tests are simple capture problems devised by the authors. The presence of an asterisk '*' beside pass or fail for each test indicates that the search mechanism believes it has found the correct answer. Otherwise the search will have either run out of time or it has explored the whole available search space and found no answer. Note that the search space is restricted for both mechanisms to equivalent degrees as far as possible. A time limit of 60 seconds was set per problem for both search methods.

Both algorithms use some form of pattern recognition based move pre-selection. In the case of MTD(f) this is used to determine the order in which variations are searched, becoming a tree pruning mechanism where the search breadth must be limited. The genetic algorithm uses pre-selection to build initial genomes as well as for attacks and mutations. To ensure fairness, considerable care has been taken to ensure that exactly the same combination of liberty counting and neural network based pre-selection has been used in each case. The neural network used is one that we have developed earlier and reported in [Cant et al 2002].

Test	Name	Time(s)	Success	Boards Examined	Evaluation Calls
1	Short Capture	1	Pass*	122	79
2	GG1-P39	2	Pass*	154	99
3	GG1-P87 Ladder	2	Pass*	152	79
4	GG1-P88 Ladder	2	Pass*	230	163
5	2nd Line Trap	1	Pass*	174	87
6	GG2-P3	1	Pass*	145	110
7	GG2-P43	2	Pass*	192	82
8	GG2-P2	1	Pass*	168	97
9	GG2-P7	60	Fail	2700	2646
10	GG2-P10	1	Pass*	128	72
11	GG2-P13	3	Fail*	400	359
12	GG2-P32	7	Fail*	409	583
13	GG2-P35	1	Fail*	113	49
14	GG2-P41	60	Fail	1459	779
15	GG2-P68	0	Pass*	119	61
16	GG2-P70	2	Pass*	116	45
17	GG2-P79	1	Fail*	125	65
18	GG2-P86	2	Pass*	235	134
19	Ladder3	1	Pass*	152	112
Totals		**150**	**13/19**	**7293**	**5701**

Table 1 - Genetic Algorithm Results

Test	Name	Time(s)	Success	Boards Examined	Evaluation Calls
1	Short Capture	0	Pass*	126	82
2	GG1-P39	68	Pass	12118	8679
3	GG1-P87 Ladder	14	Pass*	1513	1082
4	GG1-P88 Ladder	35	Pass	3780	2043
5	2nd Line Trap	1	Pass*	470	261
6	GG2-P3	4	Pass*	475	269
7	GG2-P43	45	Fail	9356	6382
8	GG2-P2	4	Pass*	678	416
9	GG2-P7	38	Fail	4472	2463
10	GG2-P10	26	Pass*	3395	1828
11	GG2-P13	7	Pass*	1246	754
12	GG2-P32	8	Pass*	1053	628
13	GG2-P35	55	Fail	11439	7707
14	GG2-P41	96	Pass	17258	9979
15	GG2-P68	51	Pass*	8142	5004
16	GG2-P70	11	Pass*	1157	672
17	GG2-P79	35	Pass*	4266	2283
18	GG2-P86	1	Pass*	71	44
19	Ladder3	98	Pass	16185	9636
Totals		**597**	**16/19**	**97200**	**60212**

Table 2 - MTD(f) Results

4. CONCLUSIONS

Although the genetic algorithm gets fewer tests correct, it does take considerably less time than MTD(f) to decide on answers for the tests it and MTD(f) get right. However time may not be a very good indicator as to the performance of one algorithm over another as it always depends greatly on implementation and hardware. Whilst every effort has been made to keep both these factors as constant as possible between the two search algorithms, the implementation will always be difficult regulate. A better test of performance is the number of calls to the evaluation function, a function shared between the two algorithms, and the number of effective game tree nodes visited. In this case game tree nodes are equivalent to board positions.

Now if we compare the performance of the genetic algorithm and MTD(f), we can see that the genetic search visits a fraction of the game tree nodes that MTD(f) does and the number of evaluation calls for both methods shows a similar pattern. This does look encouraging but it must also be noted that while both methods use forward pruning techniques in the form of the neural network move scorer, the genetic method relies on random biased move selection. This in turn means that the algorithm may solve a problem successfully during one run but on a consecutive run could, with unfavourable random number generation fail to solve it. At present the results seem to be fairly consistent and the results presented above are certainly representative of a typical run.

The purpose of creating this genetic algorithm was to approximate the process a human player might go through and hopefully achieve some of the speed, sacrificing the odd mistake. For the purposes of playing Go a game tree search is usually too cumbersome to be useful for more than severely restricted searches. Hopefully a hybrid genetic algorithm may allow deeper and wider searches in Go than deterministic game tree search methods can currently cope with.

The results indicate that, at present, the genetic algorithm is still on average, slightly inferior to MTD(f) but does outperform it in certain tests. This seems to be particularly true where long move sequences are involved.

A hybrid genetic algorithm incorporating previous neural network research was proposed and tested against MTD(f) and shown to be competitive, although not superior. This is understandable given that MTD(f) is a highly developed example of its type whilst the GA is still in a developmental stage. We hope to demonstrate the usefulness of the GA compared to standard game tree search techniques after further development. On the upside problems with long sequence solutions seem to be solved more consistently by the GA. There is also a clear general speed benefit over exhaustive game tree search even when equivalent forward pruning techniques are used by MTD(f).

REFERENCES

1. Burmeister, J., Wiles, J., 1995, An Introduction to the Computer Go Field and Associated Internet Resources, Available on the Internet at http://www2.psy.uq.edu.au/~jay/go/CS-TR339.html

2. Kano, Y., 1990, Graded Go Problems for Beginners Vol 1-2, The Nihon Ki-in.

3. Plaat, A., 1997, "MTD(f), A Minimax Algorithm Faster than NegaScout", Available on the Internet at http://www.cs.vu.nl/~aske/mtdf.html

4. Richard Cant, Julian Churchill, David Al-Dabass, 2002, "A Hybrid Artificial Intelligence Approach With Application To Games", World Congress on Computational Intelligence (2002-WCCI), IEEE Int Joint Conf on Neural Networks (IJCNN02), 12-17 May 2002, Honolulu, Hawaii, pp1575-80, ISBN 0-7803-7278-6, and ISBN 0-7803-7281-6 (CD-Rom).

5. Sutton, R. S., 1988, Learning To Predict By The Method Of Temporal Differences, Machine Learning 3(1): 9-44.

6. Rutquist, P., 2000, Evolving An Evaluation Function To Play Go, Available on the Internet at http://www.eeaax.polytechnique.fr/papers/theses/per.ps.gz