

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).

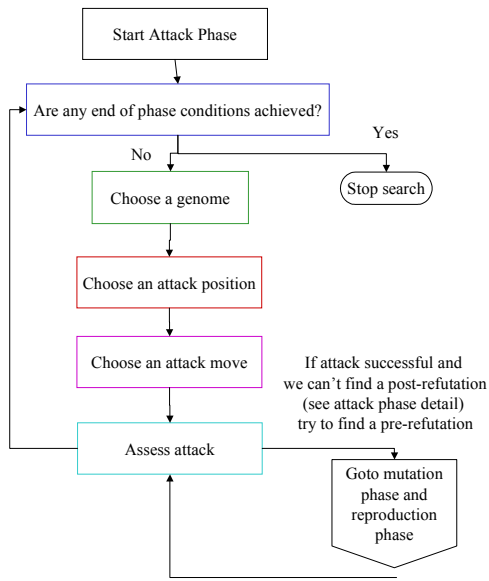


Figure 1 Attack Phase Overview

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.

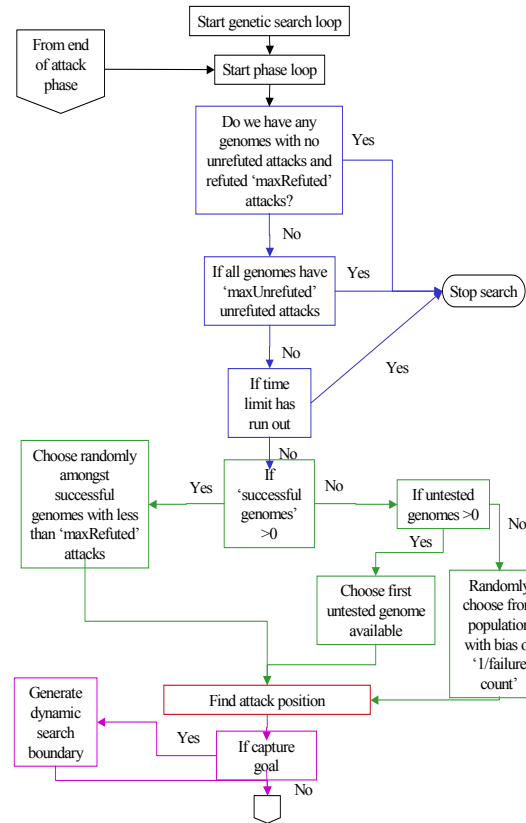


Figure 2 Attack Phase Detail Part 1

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.

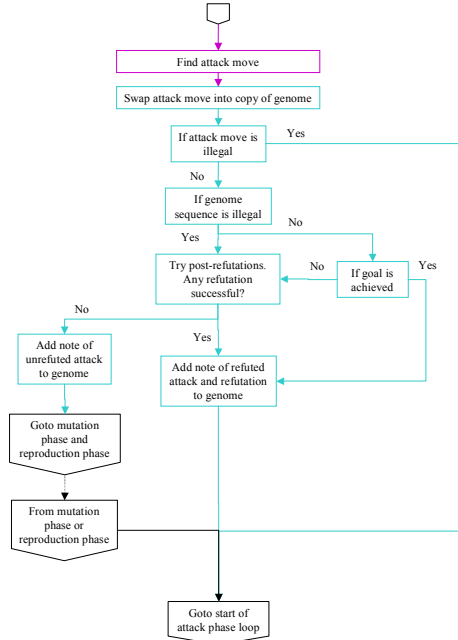


Figure 3 Attack Phase Detail Part 2

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

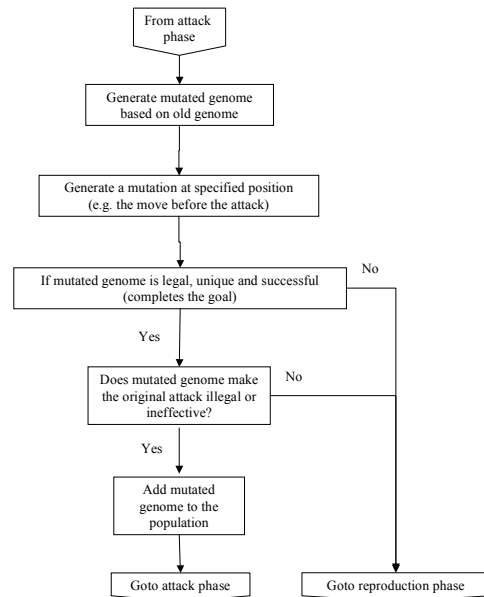


Figure 4 Mutation genetic methods.

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

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.

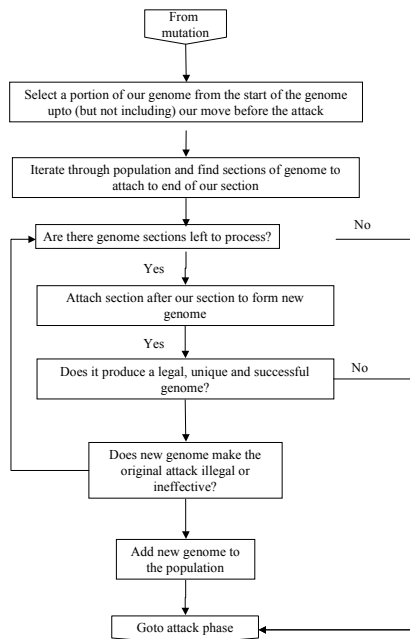


Figure 5 Sexual reproduction of lines of play those cases.

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

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	2 nd 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	2 nd 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 to 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/pe_r.ps.gz