Evolving an Evaluation Function to Play Go

Per Rutquist

Ecole Polytechnique

July 27, 2000

Contents

Abstract

This report describes an experiment to produce strategies to play the game of go by artificial evolution. An attempt is made to combine genetic algorithms with temporal difference learning by breaking up the representation into patterns that are evolved and weights that are learned. Results are compared with those obtained when weights are not learned, but found by random mutations. The patterns are allowed to include the results of tactical searches and other information computed by a conventional go playing program (Gnu Go), which provides some expert knowledge in the input data.

Chapter 1

Introduction

The simplicity of the rules and the complexity of the game are two of the reasons that have made Go so popular a target for implementing AI techniques. Also, and perhaps more importantly: While the world champions of most other board games are computers, no computer program today comes near matching human performance in the game of go. This indicates that this game is relevant to some of the most interesting (and the most difficult to understand) aspects of human intelligence.

1.1 Computer Go compared to Computer Chess

The only game that has received more attention from AI programmers than go, is chess. While the game of chess is often seen as the ultimate mind sport in western cultures, it is not as famous as go in the orient, and for AI research it is almost boring when compared to go.

When Deep Blue defeated the world chess champion Garry Kasparov, it did so by combining state of the art goal-oriented search techniques (implemented on a parallel supercomputer) with a modest amount of chess knowledge. Such an approach would not work for the game of go.

Brute force searching is impossible because of the high number of moves available in each position, and even if it were possible to search ahead for a large number of moves (as it is when playing on a very small board), there exists no easily calculated evaluation function.

In chess, if a piece can be captured easily, it is usually removed immediately, so just counting pieces provides a reasonable evaluation function (provided that the search is deep enough to end in a quiescent position.) In go "dead" stones are usually left on the board until the very end of the game because capturing them would be a waste of time, so just counting stones does not work as an evaluation function. Table 1.1 compares some of the aspects of chess and go. A more extensive comparison between the two games can be found in [1].

1.2 State of Computer Go

The best computer go programs to date are at the level of a good amateur. They are based hand made rules, and use limited searches to solve local subproblems.

	Chess	Go
Board size:	8 x 8	19 x 19
Moves per game:	~ 80	~ 300
Branching factor:	~ 36	~ 250
Human lookahead:	up to ~ 10 moves	up to ~ 60 moves
Evaluation function:	Simple	Hard
Computer strategy:	Brute force search	Several methods being investigated
Time to develop top programs:	Months	Years
Computer compares to human:	World champion	12-year old

Table 1.1: Some of the differences between chess and go, from an AI programmers perspective.

Despite the 5-10 person-years that have gone in to the development of each top program, they all have weaknesses. A strong player, once he has found the program's weaknesses, can give any computer program a very large handicap and still win.

There is, however, steady progress being made. At international competitions, computer programs compete against each other and against human opponents. The handicap given to the computer playing against a human has decreased from sixteen moves in 1991 to nine moves today.

(See [10] for a more complete account of the state of the art in computer go programming.)

1.2.1 Strategies for an automatic player

Computer go programs can either be based on rules of the type "In this type of position, make this move." or on an evaluation function in combination with a shallow (1-ply) search. Some programs use a hybrid approach; Using rules to find a few candidate moves and then using the evaluation function to select the move to be played.

1.2.2 Rule-based players

The most common approach to (non-machine-learning) go programming is to write modules that find moves to complete sub-goals of the game, such as attacking, defending, connecting, etc. One of these modules usually consists of a database of patters that suggests moves. The modules typically have some way of indicating the urgency of a move so that the main program can decide which of the suggested moves should be made. The top programs contain 50-100 modules that handle different kinds of subproblems, and they have databases of more than 1,000 patterns. Some of these modules are based on search techniques like in a chess program, but the difference is that a chess program always searches for moves to make, while in go, it is often sufficient to know that certain goal *can* be obtained. Actually making the move to reach that goal might not be a good idea if it is unnecessary[1] or if another goal is more urgent.

[1]For example: If a group of stones can be captured no matter what the opponent does, then making the move(s) to actually capture it is a waste of time. Typically such a group is

1.2.3 Evaluation function

Instead of having rules to determine the urgency of a move, the program could simply try making the move and look at the results directly. This requires an evaluation function, that can assign a score to a position.

In contrast to in most other games, where evaluation functions are usually fast and combined with a rather extensive search, evaluation functions in go are usually so complex that only one ply of search is possible.[2] Evaluation functions often use the same kind of modules as do the rule-based players.

The players that are created by machine learning [8] are usually based on evaluation functions.

1.2.4 Global vs. Local Evaluation of the Board

The evaluation function for a game like chess is global, in the sense that it gives one evaluation for the entire board, and does not say anything about the individual squares or intersections.

A common method when programming Go is to construct a local evaluation function that for each point on the board gives the probability of that point belonging to one side or the other at the end of the game. This method has an obvious advantage in machine learning; There will be a lot more data to learn from, because at the end of each game each point on the board provides one bit of learning data (as compared to only looking at the score). A disadvantage is that it is not obvious how to add up the local evaluations to a global evaluation.

An advantage of a global evaluation function, on the other hand, is that it can take into account moves that affect the final score without affecting ownership of territory locally. For example, sometimes it is wise to sacrifice a stone to make a group more difficult to capture. The group can still be captured, but as the opponent has to spend several moves capturing it, the player that made the sacrifice can use that time to gain territory on the outside of the opponent's stones.

1.2.5 Machine Learning

Humans seem to be able to evaluate go positions by "seeing" what stones are connected, what stones are alive or dead, what parts of the board will become territory of each player, and so on.

Formulating these intuitive concepts into a computer program has proven very difficult. Thus the interest in machine learning.

Several machine learning approaches have been tried for go. Most of them are based on the evaluation function approach. The reason for preferring this approach to rules that suggest moves is that it is easier to judge if an evaluation was correct (look at the end result) than to say weather a move was good (requires an independent expert.) Having as much and as reliable feedback as possible is important in any kind of reinforcement learning.

A very radical approach is to use genetic programming [6, 7] to create strategies that have very few a priori constraints on what kind of computation to

left on the board until the end of the game, when the players agree to remove it.

[2] Usually, not even all of the possible moves are tested, but some filter is used to find likely candidates for evaluation.

perform. However, the search space of such functions turns out to bee much too vast. [5]

Using a more limited type of architecture, such as neural networks [3], has other problems. For example, a neural network would need to be very complex to be able to solve a very simple type of go problem known as *ladders* (see appendix A.)

A possible approach is then to use conventional methods to solve that kind of problems, and use the solutions as input for machine learning. A successful application of this approach is M. Enzenberger's *Neuro Go* [4] that uses a Neural Network whose topology reflects the concept of strings, and that takes some "expert knowledge" (including the results of reading out simple ladders) as input.

Since machine learning often requires a large number of games to be played, a smaller size board than the standard 19 x 19 is often used. On a smaller board both the number of moves per game, and the branching factor are lower. Therefore gameplay is much faster, which decreases the demand for computational resources. A common board size used in machine learning is 9x9. All the experiments described in this report were conducted on a 9x9 board.

Chapter 2

Overview

2.1 The Goal of this Project

The goal of this project is to use machine learning to create an automated go player.

- The automated player should be able to play well.
- In order for the program to be able to profit from the areas of go programming where conventional techniques are already doing well, the player should also have access to some of the functionality of a conventional go program.
- The amount time required for learning should be reasonable.

To fulfill all of these goals, we need a representation of our player that is complex enough to allow it to play well and that includes the primitives of a conventional go program. Still, it must be such that it is possible to find a good player in limited time. This implies that the representation must not bee too complex, unless we can find a very efficient way to explore the search space.

2.2 A Go Player Based on an Evaluation Function

The architecture that was chosen for the player consists of a set of patterns, each with an accompanying weight. Such a collection will henceforth be referred to as a *strategy* for lack of a better word.

A strategy is then interpreted as a board evaluation function; A function that given the state of the board returns an estimate of the end result. The function is implemented as a weighted sum of the number of occurrences of different patterns on the board. Figure 2.1 illustrates this.

2.3 Gnu Go

As a basis for the evolved strategies described in this report, the free go program "Gnu Go" [9] was used. Gnu Go is a rather good computer go program, and

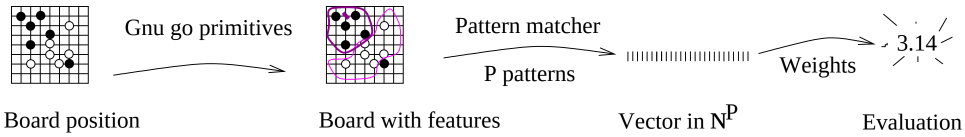


Figure 2.1: The general idea: Gnu Go's interpretation of the position is given to a pattern matcher, that in turn gives a vector of integers. By multiplying that with a set of weights, the evaluation is obtained.

its source code is freely available. It can be made to use different playing styles which makes it somewhat varied as an opponent. When it was used as a basis for the evaluation function it provided routines for tactical searching, connectivity and territory estimation, eyes and many other things. (The main part of Gnu Go's playing algorithm is a database consisting of over 1,500 hand-made patterns that suggest moves. Its creation represents a considerable effort by expert Go players.)

2.4 Patterns

The patterns are represented as lists of variable length. Each element of the list specifies what to look for at one point on the board.

$$Pattern = \begin{pmatrix} (x, y, type, constraint), \\ \cdots \end{pmatrix} \tag{2.1}$$

Where (x, y) is the relative position of the stone in the pattern. (The pattern matcher will try to match the pattern in all eight possible transformations.) $type \in \{white, black, empty, edge\}$ specifies what we are looking for. The *constraint* is used to look for special features. It can be something like $fourth\_order\_liberties > 3$ or $captured\_in\_ladder = \text{TRUE}$ using some property that is easily calculated. The special value *no_constraint* means that anything of the correct *type* is acceptable. The same stone can have several constraints attached to it.

In all 39 different boolean constraints were used for empty intersections and 47 for the stones. These constraints were taken from the internal functions used by the Gnu Go engine. Many of the constraints depend on the results of tactical goal-based searches (life and death of collections of strings). The search depth of these searches must be rather small since these searches have to be performed for each move that is tried.

2.5 Weights

The weight of a pattern is a measure of how important the strategy thinks it is to find that pattern on the board.

2.5.1 Finding Weights

Given a fix set of P patterns then finding weights is a prediction learning problem in $\mathbb{R}^P$. It can be stated as:

> Find the set of weights that minimizes the expected difference between the strategy's evaluation of a position and the expected end result of the game in which that position was found.

or:

$$weights = \arg\min\left(E(eval(position) - E(endresult|position))\right) \tag{2.2}$$

To solve this problem temporal difference (TD) learning seems like a good approach.

2.5.2 Temporal Difference Learning

The basic idea behind TD learning is that the evaluation of one position should be the same as the evaluation of the following position, and the evaluation of the end position should be the end result.

Temporal difference methods have been shown temporal to be superior to conventional prediction learning in a special case [12], and they should produce more accurate predictions in most supervised learning problems as they make more efficient use of experience.

Applying the simplest form of TD learning to our problem we obtain a linear system of equations.

Equations

Let p_{ij} be the number of occurrences of pattern j at move i and w_j be the weight of pattern j. W_i and L_i represent the number of prisoners won and lost between the position at move i and that at position $i+1$. *End_result* is the true evaluation of the end position, not counting any prisoners already removed from the board. N is the number of moves in the game and P is the number of patterns. (w_0 is a bias that is always present exactly once, and that is not associated with any pattern.)

$$\begin{array}{rcl}
w_1(p_{21}-p_{11})+\ldots+w_P(p_{2P}-p_{1P}) & = & L_1 - W_1 \\
w_1(p_{31}-p_{21})+\ldots+w_P(p_{3P}-p_{2P}) & = & L_2 - W_2 \\
\vdots & = & \vdots \\
w_1(p_{N1}-p_{(N-1)1})+\ldots+w_P(p_{NP}-p_{(N-1)P}) & = & L_{N-1} - W_{N-1} \\
w_0 + w_1 p_{N1} + \ldots + w_P p_{NP} & = & \text{End_result}
\end{array} \tag{2.3}$$

(This description is for only one game, in reality the database contains thousands of games, so the above list of equations is much longer.)

The first $N-1$ equations assure that the evaluation of a position changes as little as possible from one move to the next, while the last equation means that the evaluation of the end position should be as close as possible to the actual end result.

This method immediately finds the optimal weights for a given set of patterns on a given set of games. This may appear to be an advantage, but it might be bad for learning if the weights of patterns change too drastically as the result of changing a single pattern in the strategy.

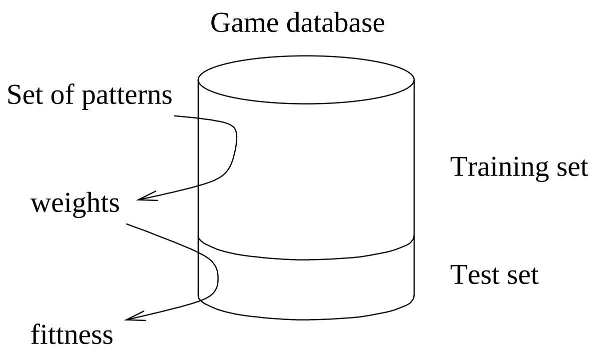


Figure 2.2: The fitness of a set of patterns is determined in two steps. First, a good set of weights is found based on one set of games, then prediction ability is tested on a second set of games.

Noting the distinction between the last equation of a game, that ties the evaluation to the end result, and the other equations, it seems natural to treat it differently than the others. Multiplying that equation by a factor γ[1] assures that the final predictions won't be to far off mark.

2.6 Fitness Function

To guide the selection process we need a function that gives a measure of how good a strategy is. The obvious way to measure this is of course to let the it play a large number of games against various opponents. This is very costly however, and we only have one opponent available. (Even though Gnu Go can be made to change its playing style slightly.) A much faster way is too look at how well the strategy predicts the outcome of games that are stored in a database. (On a small board, a game might consist of 25 moves with on average 40 moves available in each position, so playing a single game is about as costly as testing the strategy's ability to predict the score from 1,000 positions in the database.) As a means of measuring prediction error the TD equations from section 2.5.2 can be used.

Having too many patterns in a strategy is probably not good. A regularization term should be added to avoid overfitting. Our fitness function then becomes:

$$F(strategy) = \sum_{test\ \ set} ||error||_2 + \alpha P \tag{2.4}$$

(P is the number of patterns in the strategy. The *error* is the residual in equation 2.3 from the section on temporal difference learning.)

It is worth noting that this method effectively discriminates against patterns that mimic some other pattern. Since these patterns do nothing new, they will not decrease the error, and so they will not survive. This might be a good thing, but some redundancy can sometimes be good for evolution, and maybe removing to much of it is bad.

[1] $\gamma = 4$ worked well in the experiments

2.7 The Game Database

Using a database of games in to make a fitness function only works if it can be assumed that a strategy that predicts the end results of games in the database well also plays well itself.

2.7.1 Generating Games for the Database

A player that can correctly calculate the expected outcome of the game from any position is obviously optimal.[2] For this to be the case we have to assume that the player knows what the probability is for each of the opponents possible move in any position.

However, predicting the outcome of games in a database is not the same thing since those games might have been played by a different player against a different opponent. Hence it is important that the database contains games that are as similar as possible as those played by the strategy. A way to accomplish this is to use games played by the strategies against a computer[3] opponent.

However, it is equally important that the database contains as varied games as possible. If a certain kind of position is never found in the database, then strategies will never learn to recognize that feature, no matter how important it may be. A way of ensuring that new positions will be encountered is to let the player make random *exploratory* moves some times. Since such moves usually lead to a worse position than necessary we should not include the corresponding equation in equations 2.3.

2.7.2 Implementation

The positions in the database will be used for matching patterns that contain Gnu Go primitives. Those are expensive to compute, and therefore they must be stored in the database, along with the state of the board. A typical constraint in a pattern might be "string A has more than 3 liberties". Rather than storing the number of liberties etc. of each stone, as integers, the boolean result of each interesting test that can be done on it: $(x > 0,\ x > 1,\ \ldots,\ x > n)$ is stored. This has two important consequences.

- Primitives can be compacted into bit strings, for efficient storage.
- Pattern matching can be implemented efficiently using bit operators.

Storage

Games are stored in the database by run-length encoding the differences between successive positions. Using this method a 9x9 game takes up 3-4 kilobytes of computer memory.

[2] "Optimal" in the sense that it will, by construction, always make the move that optimizes the expected end result.

[3] "Computer", since it is not likely that a human will play thousands of games against a computer program. Go servers on the Internet might however make it possible to find human opponents for a smaller number of games.

2.7.3 The Opponent

The degree of determinism in the opponent a very important factor to consider.[11] Since the program will learn to evaluate positions based on the result of playing out the game against a particular opponent, it is important that this opponent is as good and as varied as possible. If an opponent has a weakness, then the algorithm is likely to learn to exploit that weakness rather than to play good go in general.

When Gnu Go is used both as an opponent and for providing features, the algorithm has an extra advantage since it has access to Gnu Go's life-and-death assessment. It might therefore (for example) learn that it is favorable to masquerade its living groups as dead, etc. This effect is somewhat diminished by the fact that the tactical searches used to provide indata for the evaluation function were not as deep as the searches Gnu Go makes when it serves as opponent.

2.8 Evolution Scheme

The part of the strategy that needs to be evolved is the pattern set. Once a set of pattern is given, the corresponding set of weights can quickly be found using the game database.

2.8.1 Algorithm

Initialization:

As the initial strategy was taken the one consisting of all possible single-element patterns.

Mutation:

A mutation on a strategy consists of adding, removing, and/or replacing a small number of patterns. (The details of this are discussed under *Implementation* below.) New patterns are created by a type of pattern mutation, discussed in the section below.

Crossover:

Crossover between two sets of patterns simply consists of taking the union of the two sets. It may appear that this would quickly result in very large sets, but since the two strategies can be expected to have most of their patterns in common, the strategy size stays reasonable.

2.8.2 New Patterns

When adding a new pattern, we either create a stone with a random constraint or copy and modify another pattern from the strategy.

A mutation on a pattern can be one of the following:

- Add a random element at a position adjacent to one that is already used in the pattern.

- Remove an element from the pattern.
- Change the constraint of a stone in a pattern.
- Inverse the pattern by replacing all black stones with white ones and vice versa.

After a new pattern is created, some checking is done that it is "legal", i. e. that it does not contain conflicting constraints, and that the edges are consistent with what can be found on a go board. It is also checked that it is not identical to any other pattern in the strategy.

Hash values

To avoid having several versions of a pattern present at the same time a hashing algorithm was used. A hash value is created based on properties of the pattern that are invariant to translations, transformations, and reordering of the stones in a list.[4] Invariance to translation is achieved by comparing all positions relative to the position of the top left element. Then the hash value is calculated based on what constraint is found where, making the value invariant to re-orderings of the list. Finally transformation invariance is achieved by calculating the hash values of all eight possible transformations, and using the lexically smallest one.

2.8.3 Implementation

For simplicity we summarize equations 2.3 into

$$\mathrm{Ax} = \mathrm{b} \tag{2.5}$$

Passing through the database is expensive, since the run-length encoded positions have to be decoded. Therefore rather than considering just one new pattern at a time, the mutation algorithm creates a large number of patterns (200), and runs them all through the pattern matcher at the same time.

With a database of 20,000 positions, and a set of 200 patterns, A would then be a 20,000 x 200 matrix.

The least squares solution to the system is given by:

$$(\mathrm{A}^\top \mathrm{A})\mathrm{x} = \mathrm{A}^\top \mathrm{b} \tag{2.6}$$

To find the weights to go with subsets of the pattern set, we only have to solve the above for sub-matrices of $\mathrm{A}^\top\mathrm{A}$ and $\mathrm{A}^\top\mathrm{b}$.

The matrix $\mathrm{A}^\top\mathrm{A}$ is 200 x 200 and can be calculated simultaneously with the pattern matching, so there is never any need to store all of A in computer memory.

The type of mutation that was actually done was:

- Add new patterns up to the maximum amount (200).

[4] The present implementation of hash values cannot, however, handle the fact that some of the constraints are identical. For example: if a pattern contains two adjacent stones of the same color, then testing a constraint on the worm that one of them belongs to is the same as testing the constraint on the worm the other one belongs to since the two stones belong to the same worm. Such patterns are sorted out when they are found to display identical behavior on the game database.

- Do pattern matching against database.
  Calculate $A^\top A$ and $A^\top b$.

- Test many (7,500) different subsets of patterns, using a simulated annealing scheme. Starting from the pattern set of the original strategy, patterns can be added or removed with a probability that depends on how the fitness value changes, and on a "temperature". Initially changes that are bad for fitness are allowed to occur with some low probability, but in the end only "good" changes are allowed.

Chapter 3

Parallelization

3.1 Motivation

Since evolution requires a very large number of games to be played, it would be too slow to run on a single computer. Therefore the computation was run in parallel on about 40 Linux workstations.

3.2 Master/Slave architecture

On a parallel architecture it is both costly and unnecessary to force generations to be synchronized. Instead a fixed size population is kept on the "master" computer, where individuals are created and destroyed one at a time.

The "slaves" are divided into two types. One type of slaves keeps playing games. The games are then sent to the slaves of the other type, who store them in their databases. Those slaves then use their databases for evolving patterns.

3.2.1 Master

The task of the master is just to handle the population, doing crossover, and sending orders to the slaves. All real work is done by the slaves.

3.2.2 Slaves, players

The players generate games for the database. Before each game they receive a new strategy from the master. At the end of a game, the game record is broadcast to all the evolving slaves.

3.2.3 Slaves, evolvers

The evolvers receive strategies, improve them as described in section 2.8.3, and send them back to the master.

Since there are constantly new games being produced, the databases of the evolvers will eventually become to large. Then some of the oldest games are removed, so that the database always contains as up to date games as possible. Continually updating the database should also help against over-fitting.

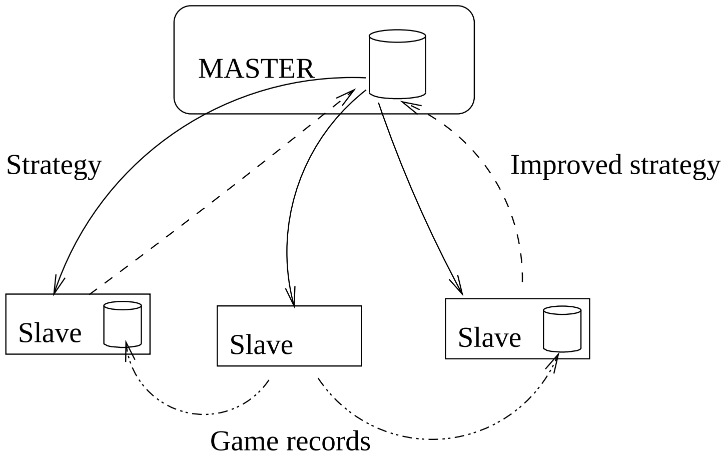


Figure 3.1: The master/slave architecture: The master keeps track of the population and handles crossover, while the slaves do small mutations (improvements) to individuals. Each evolver-slave keeps a copy of the entire game database, and updates it as it receives new games from players.

Chapter 4

First Experiments

4.1 Results

The initial random strategies obtain results of about -80 since they fail to make any territory whatsoever. Strategies that achieve an average result of -15 to -20 appear very quickly (in less than 2 hours). The strategies produced typically contained 100 - 200 patterns and the patterns contained on average 3 - 4 stones/constraints each.

While the learning algorithm did succeed in producing patterns that predicted the score well on both the test and the training sets, the patterns that were found turned out to be rather useless in actual play.

After the initial climb from -80 to -20 there is no more progress. Patterns are replaced by new ones, but the score obtained when strategies play against Gnu Go continues to be miserably low.

After an extensive search for outside sources of errors[1], the experiment finally had to be declared a failure.

4.2 Explanation of Results

It seems that the bad results are caused by irrelevant patterns that by chance were correlated between the sets, even though they were not significant in actual gameplay. A way to overcome this might be to use a larger database.[2] As described in the previous sections, the database was continuously updated by adding new games and deleting old ones. This lead to a lot of patterns disappearing as they were no longer correlated between the sets, but unfortunately new irrelevant patterns appeared to replace them, overshadowing all but the most obvious "good" patterns.

[1] Bugs!

[2] How much larger the database would have to be is not known. (In the experiments performed at most 30,000 positions were used, because of limitations in computer memory.)

# elements	# possible patterns
1	~ 130
2	$\sim 17,000$
3	$\sim 25,000,000$
4	$\sim 25,000,000,000$

Table 4.1: The number of possible patterns increases very quickly with the number of elements.

Chapter 5

Further Experiments

5.1 Redefining the Goal

Initially the strategies were made to play entire games, but as it became apparent that the strategies had trouble combining good opening and midgame with reasonable endgame play the algorithm was changed so that the evolved strategy only plays the first N[1] moves (opening and midgame) and Gnu Go finishes the game of by playing itself.

The endgame is typically the easiest part for computers to handle. The concept of territory is no longer as fuzzy as it was in the beginning, and the branching factor is significantly lower than in the earlier parts of the game. This makes it possible for the computer to find optimal or near optimal moves by conventional search techniques.

5.2 Changing the Implementation

The new method that was tried was to evolve both weights and patterns at random. Since there is no learning this method is very slow. (A possible improvement would be to use a bit of learning as a type of mutation.)

A "master" keeps track of all the strategies in the population. When a strategy needs to be evaluated, it is sent to a "slave" that evaluates it, and sends the result back to the master. Mutations and crossover are handled by the master, who creates a new strategy as soon as there is a slave available to evaluate it. Only strategies that are not awaiting the results of their evaluations are allowed to breed, so the population size must be greater than the number of slaves.

The slaves' task is simple. The strategies received are used to play Gnu Go a fixed number of times[2] calculating the average score.

When a strategy reaches the top of the population it is allowed to generate 5 offspring. Then it is sent away for reevaluation. The offspring is generated either by a mutation, or by crossover with another strategy in the population. Any strategy that has been in the population for over 50 turns is also sent away for reevaluation. The purpose of reevaluation is to make sure that only strategies

[1] Values of N that were tested were 5, 10 and 14 on a 9x9 board

[2] Usually 10 games were played for each evaluation.

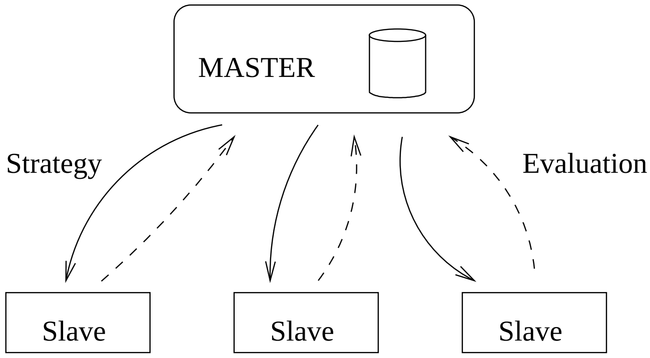


Figure 5.1: The new master/slave architecture is much simpler than the first one. The master keeps track of the population and handles mutation and crossover, while the slaves evaluate strategies by letting them play Gnu Go.

that give good results consistently remain in the population. Another way of achieving this would be to increase the number of games that the strategies play in the first evaluation. This would slow things down, however, since also the bad strategies would play many games. A side effect of the constant re-evaluations is that strategies that require much CPU-time will be absent from the breeding population on the master computer more often than fast strategies, and so they will have less opportunity to reproduce. This favors fast strategies, which might be a good thing if the goal is to find a fast strategy, and since fast strategies means less time per generation in evolution. It might be a disadvantage, however in the case where strategies that play well also are slow.

The number of strategies in the population was initially set to 100, a little more that twice the number of slaves. There is constantly one strategy absent for each slave, so at any given time only about half of the strategy were available for breeding.

Finding Weights by Random Mutations

To avoid any problems associated with the database-learning method it was decided that weights should not be learned, but instead found by evolution.[3]

It is of course possible to create good weights using only random mutations. The method should be expected to be extremely slow, however, since it adds a whole new dimension to the search space for the evolutionary algorithm. The question was if it could be done within a reasonable time frame.

Because of the way patterns are formed and the way they interact the impact of their weights varies enormously. When making mutations on weights, and when adding a brand new pattern to a strategy we have very little a priori information on what order of magnitude change is appropriate. Making a too small change would mean that the new strategy plays exactly the same game as the old one. Making a too large change on the hand could change every decision that the strategy makes, probably rendering it useless. To make sure that a mutation has a reasonably large impact on the strategy we can test the

[3]The obvious next step would of course be to reintroduce learning as a type of mutation. Unfortunately, there was not enough time to try that before the deadline of this report.

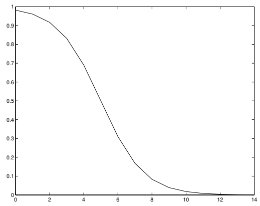


Figure 5.2: The sigmoid function that was used to provide a smooth transition from the opening patterns to the midgame patterns.

number of times the strategy would make a different move for different values of the weight change.

For this purpose the strategy is made to play a number of test games. For each move that it makes, the smallest increase or decrease in pattern weight that would have produced a different move can be calculated. These values are stored in a table, and all that is left is to decide how big a fraction α of the decisions should change. This α should of course be strictly greater than zero and much smaller than one.[4] This type of mutation is of course very costly, almost as costly as an evaluation, but every mutation that generates a useless individual is just as costly since the new individual has to be evaluated at least once. Therefore this type of mutation is justified if it decreases the ratio of doomed to successful mutations by half.

In the initial experiments, the weight of a pattern was constant over the course of a game. As it turned out that some patterns are very important in the opening, but not as important in the midgame, and vice versa, the weights were allowed to be a function of the number of moves made. The function chosen for this was a sigmoid function, that allows a pattern to have one constant value during the first few moves, and then smoothly transfer into another constant value for the midgame. (See figure 5.2) The use of two free parameters per pattern of course makes the learning more difficult, but on the other hand more powerful strategies can be evolved.

5.3 New Results

This method did result in strategies that defeated Gnu Go, although the way they do so is by finding the types of opening moves that Gnu Go does not respond well to. Against a human opponent the strategies are not as good as Gnu Go. Specifically they are easily beaten if the human makes a few unconventional moves that Gnu Go would never make. They do, however, appear to know a few tricks, and someone who is used to playing Gnu Go might be caught off guard.

The strategies produced using this method were typically much smaller with 10 - 100 patters, most of them containing only one stone/constraint. (It is

[4] The parameter α can either be chosen deterministically: "I need to present some results by next week, so α has to have reached zero by then, therefore I take $\alpha = 0.1 * (1 - \frac{t}{1\text{week}})$" or α can itself be a parameter is modified by mutation. In the latter case it would be possible to have a different α for each pattern. (Of course α should still be constrained to stay between zero and much-less-than-one))

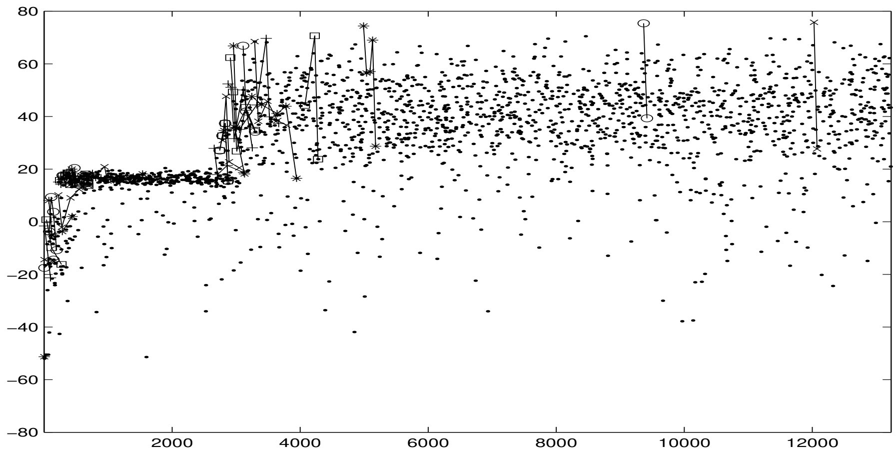


Figure 5.3: Playing only the first five moves (the rest of the game being played out by Gnu Go) strategies evolved by random mutations achieved an average result of +40 after about 3 hours of evolution. To give an indication of the error in evaluations, the re-evaluations of some individual strategies have been connected by lines. (For clarity reasons, only a random subset of the evaluations are plotted. The actual number of evaluations is on the horizontal axis.)

possible that more and larger patterns might have resulted if the computation had been allowed to run for a longer period of time.)

As a consequence of the shallow searches used by the evaluation function, the evolved strategies were much weaker than Gnu Go in tactical fighting. It was observed that the best evolved strategies actually never tried to win tactical fighting. Rather, they used a guerilla warfare tactic, sending lots of stones into enemy territory, and finally managed to steal enough territory elsewhere while Gnu Go captures the offending stones. I had hoped for the strategies to evolve patterns that would help them win tactical fighting as well. Sadly, this was not observed.

In the experiments, only Gnu Go was available as an opponent. Gnu Go has some nondeterminism built in, choosing at random when it finds several moves that it considers as almost equal, but as it turns out evolution still manages to find a way to win every game the same way, and by a huge margin. (See figure 5.4.) An effort was made to make the games more varied by changing Gnu Go's playing style, and by starting each game with a few random stones already placed on the goban, but this also adds noise to the evaluations of strategies, since those initial stones may give one side an unfair advantage.

5.4 Bugs?

Gnu Go in itself consists of much more than 30,000 lines of code. Adding to that the over 3,000 lines of lines of code written to handle the evolution of strategies and the parallelization (keeping in mind that Gnu Go originally was not written to be run in this way) there are bound to be some bugs in the code. In spite of great efforts taken to produce bug free code, several important bugs were

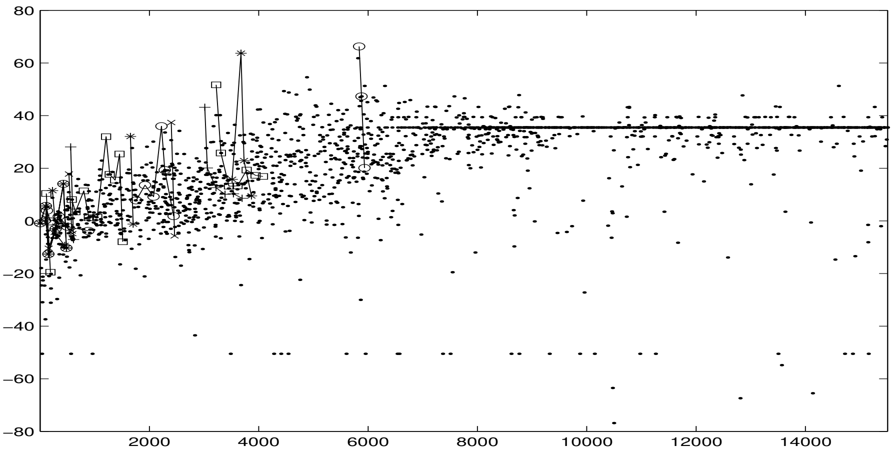


Figure 5.4: In this (typical) run, playing the first ten moves with no random start, strategies eventually learned how to win every game by 35.5

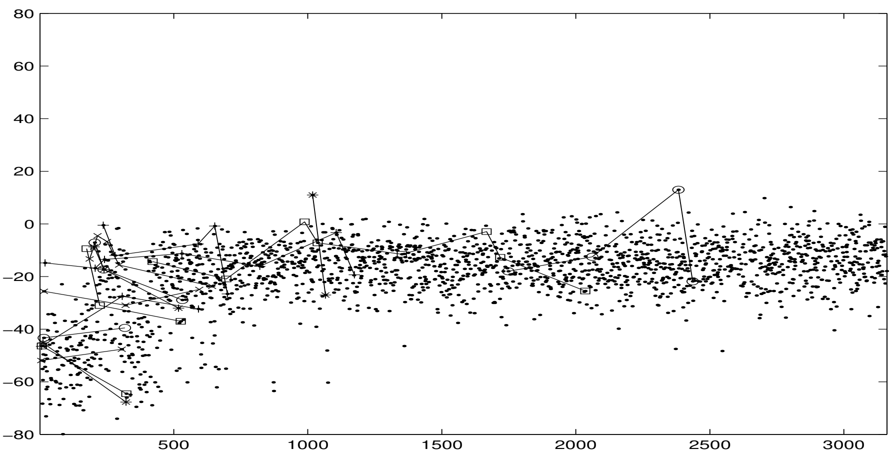


Figure 5.5: A more difficult test: Letting a strategy play the 14 first moves from random starting positions. Here, the evaluation function uses twice as many games as when the starting position is not random. The weights of patterns are a function of the number of stones on the board. Evolution has to find the two parameters of that function. As can be seen in the graph, this task is too difficult. (When this run was aborted, the top strategy only contained 9 patterns. See appendix B)

found in the code during the course of the experiments. It was very amazing to discover that often there had been good strategies produced in spite of those bug. In a way this shows the robustness of genetic algorithms.

Chapter 6

Discussion

6.1 Conclusions

- When the problems with overfitting to the database had been overcome, strategies had some grasp of go were found. This seems to indicate that the chosen representation is adequate.

- Neither one of the fitness evaluation methods that were tried is good. Using a database of games leads to problems with overfitting. Using Gnu Go as an opponent yields players that specialize in finding Gnu Go's weaknesses.

- While it is possible to such an evaluation function purely based on random mutations and crossover, the method is extremely slow even when run in parallel on several workstations. To evolve an evaluation function for 19x19 board using this method does not seem feasible.

6.2 Further work

An obvious next step is to combine the advantages of selecting strategies based on gameplay, and those of learning weights using a database of games by introducing a new type of mutation that consists of modifying the weight a bit in the direction suggested by the database. Also, using the database-learning approach to determine weights for newly created patterns might speed up evolution considerably.

In addition to this, there are several points that would be interesting to investigate further:

- The evolved strategies are specialized in defeating Gnu Go, and not in playing a good game of Go. Using self-play and hall-of-fame it might be possible to produce a strategy that plays well against any opponent.

- One way to speed up gameplay would be to evolve strategies that contain a suggestion function in addition to the evaluation function. The suggestion function would try to guess the usefulness of moves, so that less positions need to be evaluated. This would be a must if strategies to play on a

19x19 board are to be evolved. (The go playing program Golem [2] quite successfully uses a neural network this kind of suggestion function.)

- When weights are being determined by random mutations, we no longer have any good reason to stay with the evaluation function architecture. It would probably be much more efficient to evolve patterns that suggest moves directly, such as the patterns used by Gnu Go. Since the evolution-in-parallel scheme seems to be effective for this kind of learning, it would be most interesting to try to evolve that kind of strategies. They would play much faster.

Appendix A

Main Concepts of the Game of Go

Go is an ancient board game. It is played by two players each taking turn placing *stones* of opposite color on a board. The goal of the game is to surround as much *territory* as possible. At the same time the players try to capture each others stones.

The board consists of a square grid like in chess or checkers, but stones are placed on the intersections rather than on the squares. Normally the board has 19 by 19 intersections, but for beginners and for fast games smaller boards are often used.

The rules of the games are very simple: A stone is said to belong to a *string* if it is of the same color as the string and it is adjacent to at least one other stone that belongs to that string. The empty intersections that are adjacent to a string are called the *liberties* of that string. (See figure A.1.) After a move is made, strings of the opposite color are said to be *captured* and removed from the board if they have no liberties. Then, if there are any strings of the same color as the stone that was played that have no liberties, those strings are also removed. At the end of the game the number of empty intersections (territory) that each player has surrounded plus number of stones he has captured becomes his score.

An important concept in go is *eyes*. An eye is a place where the opponent can not place a stone unless he can capture the string to which the eye belongs. A string which has two eyes is unconditionally *alive* (can not be captured), because it is impossible for the opponent to place a stone in either eye. (See figure A.2.)

Another important concept is that of *ladders*. Ladders are sequences of moves to capture or defend a string that has few liberties. Such a sequence can stretch all the way across the board and as a consequence very distant stones can be vital to the life and death of a string. (See figure A.3.)

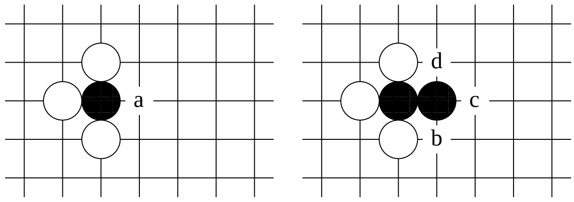


Figure A.1: The black stone on the left has only one liberty ('a'). If white is to play, he can capture the black stone. If black is to play he can play at 'a' and create a string of two stones with three liberties ('b', 'c' and 'd').

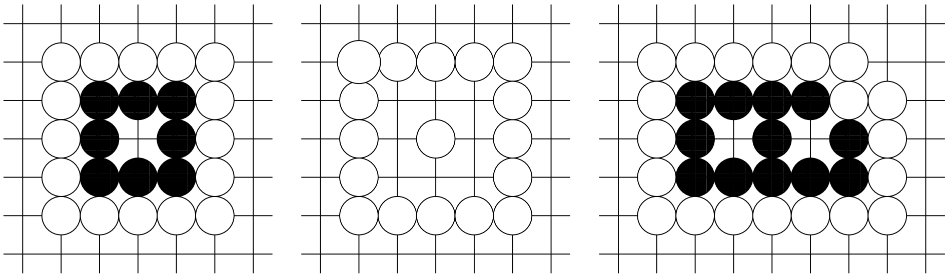


Figure A.2: Eyes: The black string on the left has one eye. White can kill it by placing a stone in the eye. The black stones on the right have two eyes, and can not be killed. If white placed a stone in either eye it would immediately be removed, since it would not have any liberty.

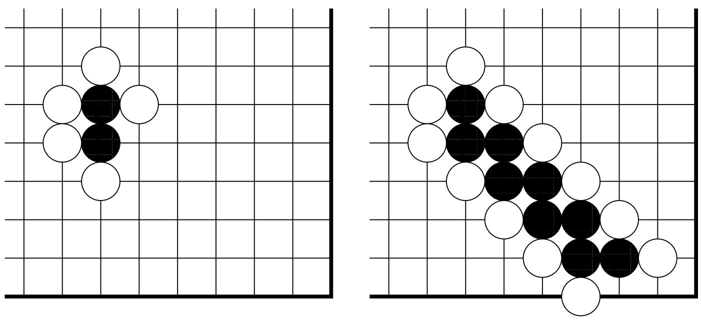


Figure A.3: Example of a ladder: Here the black stones are "dead". They can not escape since the ladder runs into the edge of the board. Another group of stones somewhere along the path of the ladder could change the situation drastically however.

Appendix B

(Small) Example of a Strategy

This strategy was evolved through random mutations and is still very small after 3,000 evaluations. It has only nine patterns, eight of which contain just one stone/constraint. It should be pointed out that most strategies that were evolved contained more and more complex patterns. This one was chosen for an example, so that the meaningful patterns would not be hiding among too many less relevant ones.

The two weights associated with each pattern refer to the opening and midgame respectively. High (low) weights mean that it is bad (good) for black (who just made a move) if this pattern is found.

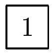
The constraints are expressed in terms of Gnu Go function calls. Their precise definitions can be found in [9].

$\boxed{1}$

Pattern 0: weights = -0.002310, -0.004512
Empty '1' moyo_color(i,j) = 2 (BLACK)
"Moyo" is a measure of influence that Gnu Go uses. This pattern says that it is good to have influence over many empty intersections.

$\boxed{1}$

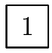
Pattern 1: weights = 0.059255, 0.069654
Empty '1' ¬(terri_color(i,j) = 2 (BLACK))
"Terri(tory)" is a stronger measure of influence. Saying that not having territory is bad is equivalent to saying that having it is good, since the number of empty intersections would be the same regardless of the move made (unless "pass" moves are considered.)



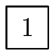
Pattern 2: weights = -0.015893, -0.037706
Empty '1' distance_to_white(i,j) > 2
Black can decrease the number of empty intersection that are close to white stones if he places his stones between white and the empty. This pattern says that such moves are good.

Pattern 3: weights = -0.004198, -0.007089
Black 'z' worm(i,j).liberties4 > 2
Liberties4 is the number of liberties that are four intersections away. Having many fourth order liberties means that the string is probably not trapped, which is good.

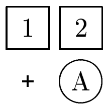
Pattern 4: weights = -0.000319, -0.000004
Black 'z' ¬(dragon(i,j).size > 2)
In the very first moves it is better to have many small strings than one large, but this pattern has very little influence (small weights).



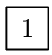
Pattern 5: weights = 0.008726, 0.046579
Empty '1' whitelibs > 3
Empty intersections where white would have many liberties if he played are bad.



Pattern 6: weights = 0.000096, 0.003521
Empty '1' strategic_distance_to_black(i,j) > 5
Empty intersections far away from black are bad.



Pattern 7: weights = 0.011808, 0.000193
Empty '1' ¬(terri_color(i,j) = 2 (BLACK))
White 'A' ¬(dragon(i,j).size > 1)
Empty '2' ¬(worm(i,j).color = 5 (BLACK_BORDER))
The only multi-element pattern in this strategy. It seems to suggest for black to play near single white stones. (The constraint on the empty '2' not to have black bordercolor is redundant since it is adjacent to a white stone.)



Pattern 8: weights = 0.658172, 0.000000
Empty '1' whitelibs > 0 (x)
A place where white would have zero liberties is a potential eye, and a secure connection.

Bibliography

[1] Burmeister, Jay and Wiles, Janet *The Challenge of Go as a Domain for AI Research: A Comparison Between Go and Chess* 3rd Australian and New Zealand Conference on Intelligent Information Systems, Perth, 1995

[2] Herbert Enderton: *The Golem Go Program* Technical Report CMU-CS-92-101, Carnegie Mellon, 1991.

[3] J. Hertz, A. Krogh, and R. G. Palmer. *Introduction to the Theory of Neural Computation.* Addison Wesley, 1991.

[4] Markus Enzenberger: *The Integration of A Priori Knowledge into a Go Playing Neural Network*, http://home.t-online.de/home/markus.enzenberger/neurogo.html 1996.

[5] Jeffrey Greenberg: *Breeding Software to Play the Game of Go* http://www.cris.com/ Jgberg/Research/Go/breed.html

[6] J. R. Koza. *Genetic Programming: On the Programming of Computers by means of Natural Evolution.* MIT Press, Massachusetts, 1992.

[7] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Massachussetts, 1994.

[8] T.M. Mitchell. *Machine Learning.* MIT Press, 1995.

[9] Man Li, Daniel Bump, David Denholm, Gunnar Farneback, Nils Lohner, Jerome Dumonteil, Tommy Thorn, Nicklas Ekstrand, Inge Wallin, Thomas Traber, Douglas Ridgway, Teun Burgers, Tangay Urvoy, Thien-Thi Nguyen: *Gnu Go* (source code and documentation). http://www.gnu.org/software/gnugo/

[10] Mller, Martin *Computer Go* 2000. Draft. Will become the Go chapter of a book about the state of the art in game programming, edited by J. Schaeffer and J. van den Herik. http://www.brl.ntt.co.jp/people/mueller/ps/Go2000.ps.gz

[11] Norman Richards, David Moriarty, and Risto Miikkulainen. *Evolving Neural Networks to Play Go*, Applied Intelligence, 8:85-96. 1998.

[12] Richard S. Sutton: *Learning to Predict by the Methods of Temporal Differences*, Machine Learning 3: 9-44, 1988.

There is a very extensive online computer go bibliography available on Markus Enzenberger's homepage:
http://home.t-online.de/home/markus.enzenberger/compgo_biblio.html
As of July 2000 it lists 126 articles, all of which can be found online.