Machine Learning, Game Play, and Go

David Stoutamire

Abstract

The game of go is an ideal problem domain for exploring machine learning: it is easy to define and there are many human experts, yet existing programs have failed to emulate their level of play to date. Existing literature on go playing programs and applications of machine learning to games are surveyed. An error function based on a database of master games is defined which is used to formulate the learning of go as an optimization problem. A classification technique called *pattern preference* is presented which is able to automatically derive patterns representative of good moves; a hashing technique allows pattern preference to run efficiently on conventional hardware with graceful degradation as memory size decreases.

Contents

1 Machine Learning **6**
- 1.1 What is machine learning? . . . 6
- 1.2 Learning as optimization . . . 7
- 1.3 The Bias of Generalization . . . 10
- 1.4 Two graphical examples . . . 11

2 Computer Gamesmanship and Go **14**
- 2.1 Computers playing games . . . 14
  - 2.1.1 Simple state games . . . 14
  - 2.1.2 Computer chess . . . 17
- 2.2 The game of go . . . 18
- 2.3 Go programs . . . 21
  - 2.3.1 David Fotland . . . 22
  - 2.3.2 Ken Chen . . . 29
  - 2.3.3 Bruce Wilcox . . . 30
  - 2.3.4 Kiyoshi Shiryanagi . . . 30
  - 2.3.5 Elwyn Berlekamp . . . 30
  - 2.3.6 Dragon II . . . 31
  - 2.3.7 Swiss Explorer . . . 31
  - 2.3.8 Star of Poland . . . 32
  - 2.3.9 Goliath . . . 32
  - 2.3.10 Observations . . . 33

3 Machines Learning Games **34**
- 3.1 Samuel's checkers player . . . 34
  - 3.1.1 Learning by rote . . . 34
  - 3.1.2 Learning linear combinations . . . 34
  - 3.1.3 Learning by signature tables . . . 35
- 3.2 Tesauro's backgammon player . . . 37
  - 3.2.1 Backgammon . . . 37
  - 3.2.2 Neural Nets . . . 37
  - 3.2.3 Features . . . 38
- 3.3 Generalities . . . 38
  - 3.3.1 Signature tables vs. Neural Nets . . . 38

3.3.2 Importance of representation 39

4 A Go Metric **40**
4.1 Defining an error function 40
4.1.1 Expert game data 40
4.1.2 Simple rank statistic 41
4.1.3 Normalized rank statistic 42
4.2 Plotting 43
4.3 Evaluation of performance 45

5 Pattern Preference **46**
5.1 Optimization 46
5.2 Simple methods of categorization 48
5.3 Hashing 55
5.4 Pattern cache 58
5.5 Improvements 59

6 Example Game **61**
6.1 Evaluation on a master game 61
6.2 Discussion 71

7 Conclusion **75**
7.1 Summary 75
7.2 Future work 76

A Resources **78**

B Methods **80**

C Game record **81**
C.1 Moves from section 6.1 81

D The Code **85**

List of Figures

1.1 Simple two dimensional error function 8
1.2 Complex two dimensional error function 8
1.3 Effect of the bias of generalization on learning 12
1.4 Simple function to extrapolate 13

2.1 Part of the game graph for tic-tac-toe. 15
2.2 Examples used in the text 19
2.3 Important data structures in Cosmos 23
2.4 Group classification used by Cosmos 25
2.5 Move suggestion in Cosmos 26
2.6 Sample move tree from pattern 28

3.1 Samuel's signature table hierarchy 36

4.1 A sample NRM plot 43
4.2 Random NRM plot 44
4.3 Example study plot 44

5.1 Windows used for pattern extraction 50
5.2 NRM plots for 3x3 and radius 1 diamond windows 51
5.3 Comparison of NRM plots for diamond window 51
5.4 Study graph for plots in previous figure 52
5.5 Comparison of NRM plots for square window 52
5.6 Study plot of previous figure 53
5.7 Comparison of NRM plots for square window 53
5.8 Study plot for graph-based window 54
5.9 Comparison of hash and map stategies 56
5.10 Effect of collisions 57
5.11 Study plots with liberty encoding 60
5.12 NRM plot for best classification 60

6.1 Move 2, master game 62
6.2 Move 5, master game 63
6.3 Move 6, master game 63
6.4 Move 9, master game 64
6.5 Move 11, master game 65

6.6 Move 15, master game 65
6.7 Move 20, master game 66
6.8 Move 21, master game 67
6.9 Move 22, master game 67
6.10 Move 24, master game 68
6.11 Move 28, master game 68
6.12 Move 43, master game 69
6.13 Move 47, master game 70
6.14 Move 79, master game 70
6.15 Move 92, master game 71
6.16 Move 138, master game 72
6.17 Move 145, master game 72
6.18 Move 230, master game 73

D.1 Classes used in `iku` 86

Chapter 1

Machine Learning

1.1 What is machine learning?

ML is an only slightly less nebulous idea than *Artificial Intelligence* (AI). However, Simon[35] gives an adequate working definition of ML, which I shall adopt:

> ...any change in a system that allows it to perform better the second time on repetition of the same task or on another task drawn from the same population.

The point is that in a system which *learns*, there is (or can be) *adaptation of behavior over time*. This may be the response of a biological organism to its natural environment (the usual meaning), or a system wallowing in an artificial environment designed to coerce the learning of some desired behavior (the ML meaning). In this broad sense, the long term adaptation of genetic material in a population of cacti to a changing climate represents learning; so does the short term adaptation of a studying undergrad to the new, unfamiliar symbols found in her calculus book.

Much of AI research has been unconcerned with learning, because it is perfectly possible to get programs to do things which seem intelligent without long-term adaptation. For example, most systems which are labeled 'expert systems' are frameworks for manipulating a static database of rules, such as "Cars have four wheels". As one may imagine, it takes an enormous number of such rules to represent simple everyday concepts; having four wheels really also assumes that "Wheels are round", "Wheels are put on the underside of a car", "A car 'having' a wheel means that the wheel is attached to it", "If a car 'has' a wheel, that wheel may be considered part of the car"... etc., *ad nauseum*.

No matter how well such systems can perform in theory, an immediate practical problem is that they require a human to decide the appropriate rules to include in the database for the problem at hand. The difficulty of generating and maintaining such rule databases led to 'expert' systems: because the size of a rule database sufficient to deal with solving general problems is prohibitive, it is necessary to focus on smaller, less complex, encapsulatable problems. In this way depth can be achieved at the expense of breadth of knowledge.

In many fields of study, the time is approaching (if, indeed, that time has not long past) when the human digestion and regurgitation needed to crystalize a problem domain into a form useful to an artificial system of some sort will be impractical. It doesn't take long to

think of many such fields. The amount of information that has been gathered by a doctor or lawyer in the course of their career is daunting; indeed, this is why those professions are so highly rewarded by society, to return on the investment of education. However, long ago the knowledge needed for comprehensive doctoring exceeded what was possible for individuals to learn. For this reason a wide variety of medical specialists can be found in any phone book.

Similarly, no scientist is expected to have a deep grasp of areas outside of a few primary fields of study. The complexity is simply too great. One useful role of AI could be to help us understand those areas of research which presently require half a lifetime to understand. Some system needs to be found to facilitate the understanding of subjects which appear to be taxing the apparent limits of the present system of human learning.

Far afield from the methods of science, game playing happens to be studied in this paper because it is concrete and easy to formalize. In addition, people are familiar with game play—who has not played tic-tic-toe?—so the immediate goal (to win!) is not obtuse and doesn't need explanation. The everyday sort of problem, such as walking a bipedal robot across a room with obstacles such as furniture in real time, is the really tough kind; still beyond our present ability to hand-code. While it is universally believed that such artificial walking will become possible, it is also true that a new-born foal can accomplish this task after a learning period of a few minutes. The foal is not born with all the skills needed to walk smoothly; it quickly derives the dynamics of balance and locomotion by experimentation. Similarly, to be useful, an AI system has to be able to learn its own rules; having a dutiful human interpret data into new rules is putting the cart before the horse. Clearly we need systems with *self-organizational* abilities before we can claim to have any palpable success in AI.

1.2 Learning as optimization

It's all very well to speak of adaptation over time, but how can this be quantified? Some kinds of problems suggest a clear notion of what learning is. Optical character recognition systems have some sort of error rate of recognition (such as the ratio of incorrect characters to the total number of characters) which one wishes to have as low as possible; similarly, a chess program has an objective international rating, based on games played against other (most likely human) players. In any event, some numeric measure of 'skill' can be obtained for the problem. This measure is known as an 'objective' function because it encodes the objective, or goal, of the learning process. When increasing skill numerically reduces the value of the objective function, as in a character recognition system, the function is refered to as an *error function* for the problem.

An error function associates a number with each possible *state* of an ML system. A computer program, for example, can frequently be represented as a string of bits. Each possible combination of ones and zeros is an individual state of the program. So if the ML system generates programs to solve some task, the current program would be considered its state.

Consider the task of fitting a straight line to some set of points. A standard error function which is used to do this is the sum of the squares of the differences at each point

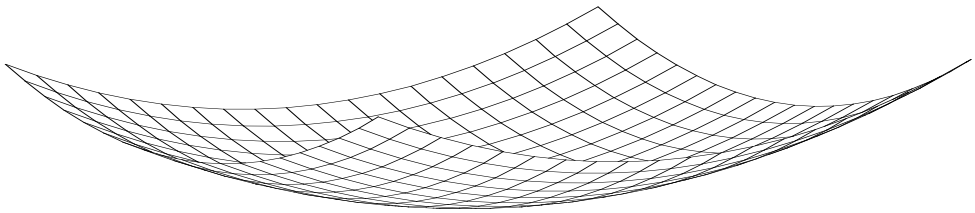

Figure 1.1: A simple two-dimensional error function, representing a problem such as simple linear regression.

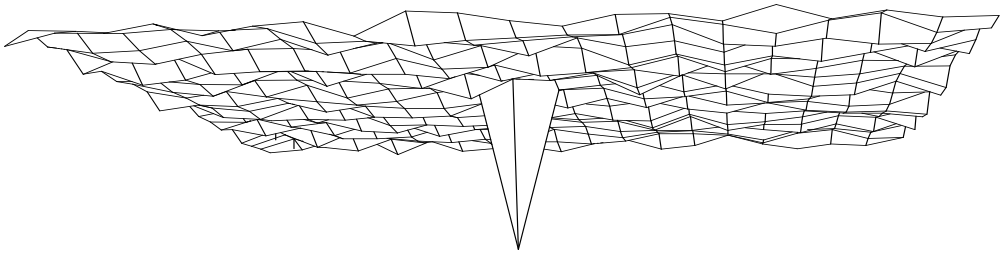

Figure 1.2: An ugly two-dimensional error function, with many local optima.

i's abscissa with the straight line. (Squaring is done to make any deviation from the line a positive value, hence always contributing to the error.) The state of this system is the position of the line, typically encoded as the values a and b in the equation $y = ax + b$. The error function is thus:

$$E(a, b) = \sum_i (ax_i + b - y_i)^2$$

Because there are two variables, the state of the system can be depicted as a point (a, b) in a two-dimensional plane and is said to have a two dimensional *state space*. An optimization problem with such a two-dimensional state has a convenient graphical representation—see figure 1.1. Unfortunately most problems involve vastly higher dimensions and have no such convenient visualization.

The line-fitting problem above evidently has a single lowest point at the bottom of the 'valley' of the graph. This is an *optimum*. Line-fitting is a particularly simple problem, because it has at most one optimum, and the graph at other points slopes more-or-less towards that optimum, so it can be learned by the simple procedure of picking any point and sliding down the side of the graph until one can't go down further. Much more complex error spaces, where this is not sufficient, are common. An example is the error plot graphed in figure 1.2. In this error function, there is only one true minimum, but there are lots of *local* minima, points where moving in any direction locally increases the error. Minima which are truly the least among all points are called *global minima*.

Optimization problems have been studied for many years. While no general solution exists for quickly finding global optima of a function, many well understood methods exist for finding local optima and have been used with success on a wide variety of problems.

In order to cast a problem as an optimization, it is necessary to define the encoding of the state space, and an appropriate error function to be minimized[1] with respect to the inputs. For example, suppose we wish to train a program to play checkers (or go, tic-tac-toe or any other similar game). Here are three possible error functions (of many):

- We could estimate a given program's play by having it play N games against some human opponent, and rank it 0.0-1.0 by dividing the number of games it won by N.
- We could collect a database of games played by human masters of the game with records of every move made. Then we could have the error function be the fraction of moves made by the experts which were also selected by the program.
- We could ask a human player to observe the program play and rate it 1–10.

Each of the above methods have strengths and weaknesses. The first and third functions only give an estimate of the supposed real error function (the fraction of games it would win when played a very large number of times), so the optimization strategy must be *noise-tolerant*. In addition, each evaluation is very costly in human time (which is likely to be a great deal more expensive than the time of the computer). The second function is not so noisy (in the sense that it won't give different errors for the same program on different attempts) but is noisy in the sense that the recorded games are not perfect and include some poor moves[2]. It also has the cost of gathering and typing in the data, and the risk that, even if the program manages to become optimal and correctly predict all the moves, it learns in some twisted way that won't generalize to new, unseen games. For example, it is easy to construct a program that remembers all the example moves *verbatim* and guesses randomly for any unseen positions, but one certainly wouldn't accuse such a program of really playing the game, or of accomplishing any real AI.

Note that we have dissociated any particular learning method from the definition of the problem as an optimization. We are not yet concerned with *how* to optimize, although a given error function may dictate when a particular optimization technique may be likely to succeed.

The meaning of the input (the *coding* of the problem, or the way that a position in the state space is translated into an error) is important because a very small, seemingly trivial, change in the coding can greatly affect the optimization process. Thus the coding is very important to the eventual success of an ML technique. Most ML techniques are particular to a given type of coding, that is, they have been found to be successful at solving problems when posed in a particular input coding style.

[1] Or an negative-of-error function to be maximized. Henceforth in this paper optimization will be assumed to mean minimization.

[2] Even masters make mistakes. After all, someone probably *lost*.

1.3 The Bias of Generalization

Some readers will object to the foal example (section 1.1), on the basis that our foal was not starting quite *tabula rasa*. Millions of years of evolution were required to generate the architypical muscular, skeletal and nervous systems which developed over months of gestation that allow those first trial steps to be made. This is very true, and an important example of the importance of the *bias of generalization* to learning. A foal is marvelously preconfigured to allow learning to walk in a short time.

The foal learns to walk by interaction with its world. This occurs by appropriate sensitization of neural circuitry by a not fully understood learning mechanism. The foal's brain delivers commands to the world via its musculature. The result of this is a change in the perceived world, completing the cause-and-effect cycle through sensory impressions of pressure, sight, sound. As a result of this circuit to the world and back, changes occur in the foal: it learns. The same idea is used whether we are training a system to detect explosive materials in airline baggage, negotiating the terrain of some hairy multivariate function to be optimized, or teaching a system to play the game of go. There is some desired, or (to make peace with behaviorists) rewarded behavior; the system interacts with its environment, by outputting probabilities, new trial vectors, or a new generation of go programs; and the system reacts to changes in the environment in an attempt to influence the system's future behavior.

Generalization means the way that learning data affects future action. Bias of generalization means that different learners will learn the same data in different ways; in the future they will respond differently to the environment, because they have internalized that data in different ways.

Consider the way a young child and an adult would perceive this report. To the child, it is a bunch of papers, with what is recognized as writing and some peculiar pictures, but what the report *means* probably isn't much understood. To an adult, the fact that the text is on paper rather than displayed on a computer screen is probably held to be less important than the interpretation of the words. The information available to the child and the adult is the same, but what is learned is different ("I have some papers, one side of which is blank, so I can draw on them."[3] vs. "I have this report, which doesn't look interesting enough to actually read, but I skimmed it and it had nice pictures.").

Each era of thought in history brought a new interpretation of nature; the world looked at the same heavens through different lenses before and after the Copernican revolution. Science itself is a process of refining our model of reality to better describe, and hence predict (generalize) in greater accord with observations. The collective bias of generalization of the scientific community changes from decade to decade[19]. Representing the world as a combination of fire, water, earth and air is convenient because it is close to everyday human experience, but it is not appropriate because it is not useful. This was the problem with Lamarkism, the Ptolemaic system, and creationism; each does provide a description of the universe, but at the expense of additional patchwork to explain away observations. Each is a valid generalization bias, but the scientific community favors simple theories over complex (Occam's razor). The apparent appropriateness of a bias of generalization must

[3] From the author's memory.

be taken with a grain of salt to avoid pitfalls, because once committed to a given bias, that bias sets a fundamental limit both on how fast learning will occur and ultimately how well a problem can be learned.

> We must remember that *thought is abstraction.* In Einstein's metaphor, the relationship between a physical fact and our mental reception of that fact is not like the relationship between beef and beef-broth, a simple matter of extraction and condensation; rather, as Einstein goes on, it is like the relationship between our overcoat and the ticket given us when we check our overcoat. In other words, human perception involves *coding* even more than crude *sensing*. The mesh of language, or of mathematics, or of a school of art, or of any system of human abstracting, gives to our mental constructs the structure, not of the original fact, but of the color system into which it is coded, just as a map-maker colors a nation purple not because it *is* purple but because his code demands it. But every code excludes certain things, blurs other things, and overemphasizes still other things. Nijinski's celebrated leap through the window at the climax of *Le Spectre d'une Rose* is best coded in the ballet notation system used by choreographers; verbal language falters badly in attempting to convey it; painting or sculpture could capture totally the magic of one instant, but one instant only, of it; the physicist's equation, $Force = Mass \times Acceleration$, highlights one aspect of it missed by all these other codes, but loses everything else about it. Every perception is influenced, formed, and structured by the perceptual coding habits—mental game habits—of the perceiver. [34]

So learning is not an objective process. How learning occurs, and how the learner explains its world, is not a function of only the problem at hand, but of the learner as well.

For purposes of ML, it should be noted that here 'learner' refers not only to representation, but also the *process* of change to that representation. In the everyday world the representation (brain) controls the presentation of information; when the undergrad studying calculus finds her mind wandering she takes a break before continuing. In this way the learner moderates the learning. ML systems are not yet so capable: usually data to be assimilated is presented in a predetermined manner, and if the data is presented in a different order, alternate understanding may occur.

1.4 Two graphical examples

Figure 1.3 shows a simple example of the effect of the bias of generalization on a simple problem domain: inputs are two numbers x and y. The system is presented with *exemplars*: those points indicated by light and dark circles, which approximate two continuous curves. Presentation occurs until the system is able to correctly respond *dark* or *light* according to the color of each and every exemplar. Without yet delving into the details of each learning mechanism, we can see remarkable differences in the way each might learn to generalize about its world from specific experiences.

Naturally, the different algorithms are in the most agreement close to the exemplars, and tend to develop differing theories about the lightness or darkness of the unknown points

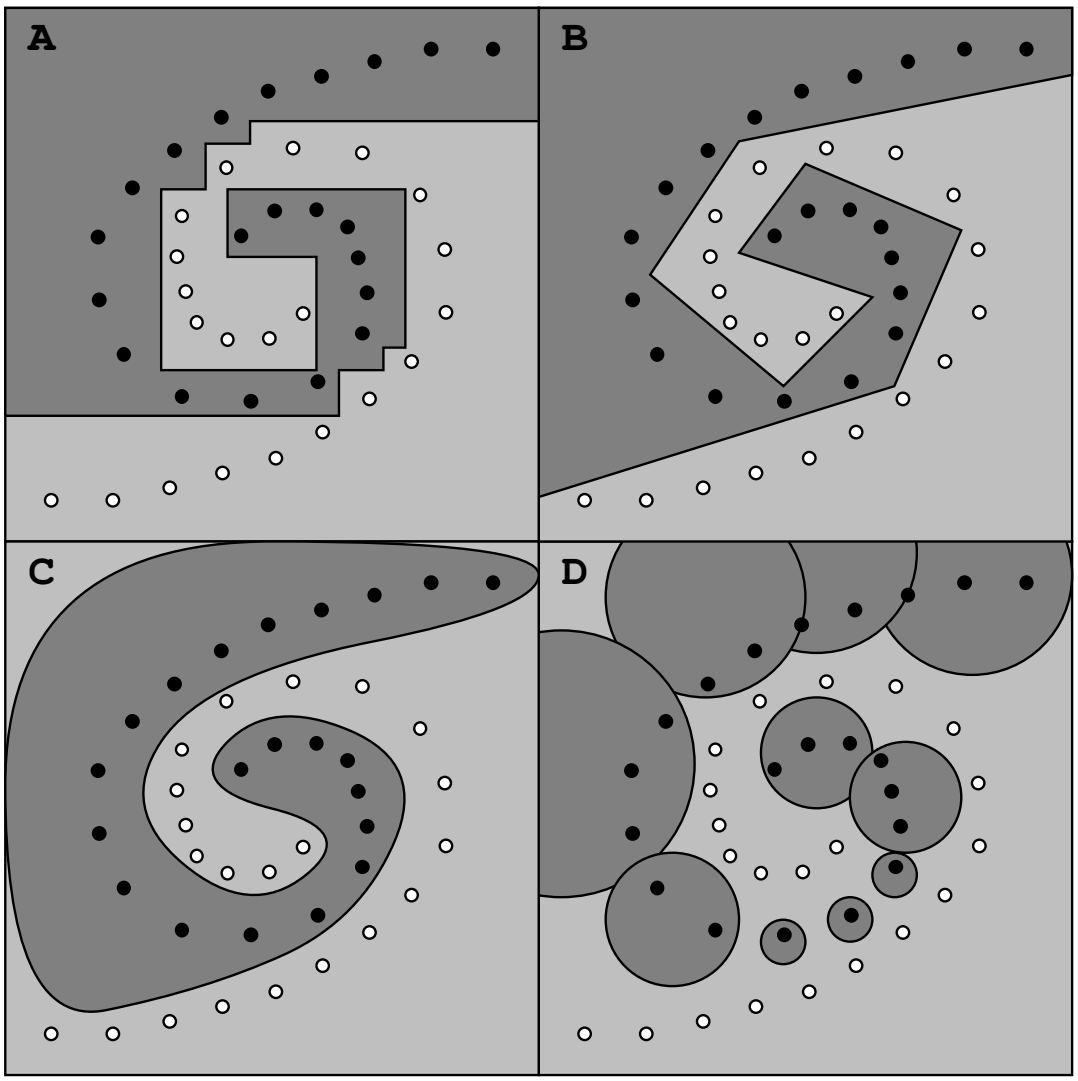

Figure 1.3: Four examples of the effect of the bias of generalization on learning. Each square represents all the possible values of the inputs; ○ and ● symbols are exemplars; the actual behavior learned for each input after training in indicated by the shade of gray. **A** and **B** represent decision tree systems such as `ID3` and *Athena*. `ID3` is only able to use a single input variable to distinguish light and dark, while *Athena* is able to make use of a linear combination of the inputs. **C** represents the smoother nonlinear models obtainable using artificial neural nets, and **D** represents the learning that might occur with a nearest-neighbor algorithm. It should be noted that these are rather optimistic generalizations for the methods mentioned.

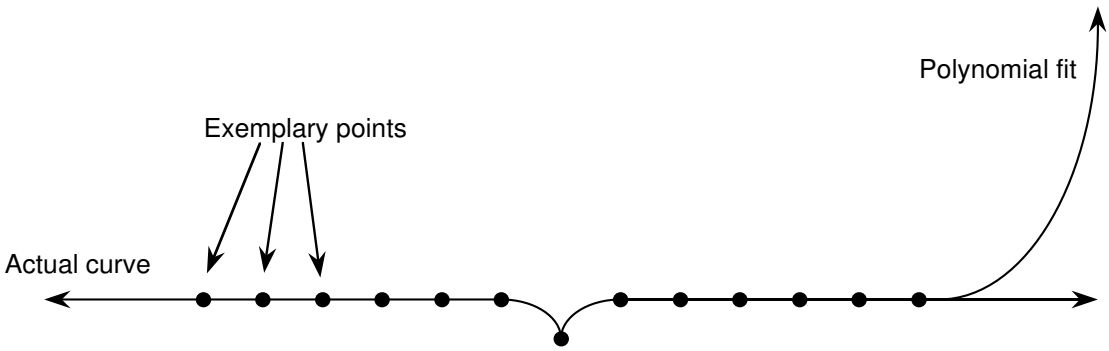

Figure 1.4: A high-order polynomial will perfectly fit the points yet produce disasterous values outside the domain of the abscissas of the exemplars.

the farther away one goes from the exemplars. But serious differences of opinion exist. **A** and **B** catagorize the upper-left corner as catagorically dark, while **C** and **D** develop the opposite view. **D** perhaps departs the most in not even expecting a single connected dark area.

Note that no notion of the "actual" light- and dark-ness of the untested areas has been given. Each of the four methods is correct for *some* problem. Each managed to correctly "understand" the exemplars. The appropriateness of a method for a problem depends on how well its bias of generalization matches that of the problem.

One last example is given in figure 1.4. Polynomials are often used to approximate functions, but they have the unfortunate effect of "screaming off to infinity" if one tries to use them to extrapolate data points. For this set of exemplars, the most common generalization technique (polynomial fitting) fails spectacularly. In this case, appropriate generalization could have been achieved if a technique that used only a subset of the exemplars had been used, such as a spline fitting. However, this would have required *a priori* knowledge of the function to be approximated, and this is knowledge that cannot in seriousness be granted to a learning algorithm; one may have intuitions about what sort of functions one expects to encounter, and this can be used to guide the choice of an appropriate learning technique, but there are no guarantees. The true function *could* have been similar to the polynomial we generated, in which case the spline approach would have been the failure.

It should be clear that the bias of generalization has an overwhelming effect on the interpretation of evidence. The immediate question is: which bias is correct? The answer is, as the reader knows, relative to the problem. However, it is safe to say that, for real-world problems, we are often very much interested in emulating the bias of generalization of real-world people. The techniques examined in the following chapters will be in part judged by this criterion.

Chapter 2

Computer Gamesmanship and Go

2.1 Computers playing games

2.1.1 Simple state games

Many games can be thought of as having a set of states (for chess, the positions of the pieces on the board) and rules for moving between those states (the legal moves). For a given state, there is a set of new states to which a player can bring the game by moving. An example for the first few moves of tic-tac-toe is shown in figure 2.1.

The rules of a game define a directed graph, where each node is a state and each arc is a move. A particular game represents a traversal from some starting state (in chess, all pieces lined up to begin) to some ending state (a stalemate or a king in checkmate). Each ending state has the concept of outcome (such as a win, draw or loss) associated with it.

By tranforming the rules of a game into such a graph, all relevant aspects of the game can be encapsulated - if two games have isomorphic graphs, they can be played by the same strategy. A great deal of work has gone into discovering relationships between the graphs of games; in particular, many games are in some sense reducible to a simple game called Nim[3]. A program that can search graphs can thus play any game that is reducible to a graph. For this reason many textbooks emphasize the searching aspect of game play, because it is an invariant among most popular games.

In chess as in many games, players alternate moves. If a win is 1 and a loss is 0, each move can be considered an optimization problem: from which new state do I have the highest probability of winning?[1]

Looking at each move as an optimization problem leads to a recursive view of the game: I desire the move from which I have the highest probability of winning. Because my opponent will also be seeking to maximize his probability of winning, I must select the move which leaves him with the least potential probability of winning even after he has

[1]For games (like chess) in which draws are frequent, having the "highest probability of winning" is not sufficiently descriptive; drawing against a strong opponent may be desirable. Similarly, for games with outcomes other than simple win/loss (such as go, in which the game is ultimately decided by a score) winning by a large margin may or may not be seen as more desirable than simply winning at all. For this report, I assume one wants to maximize the probability of winning, and that the possibility of a draw is ignorable, which is true for go.

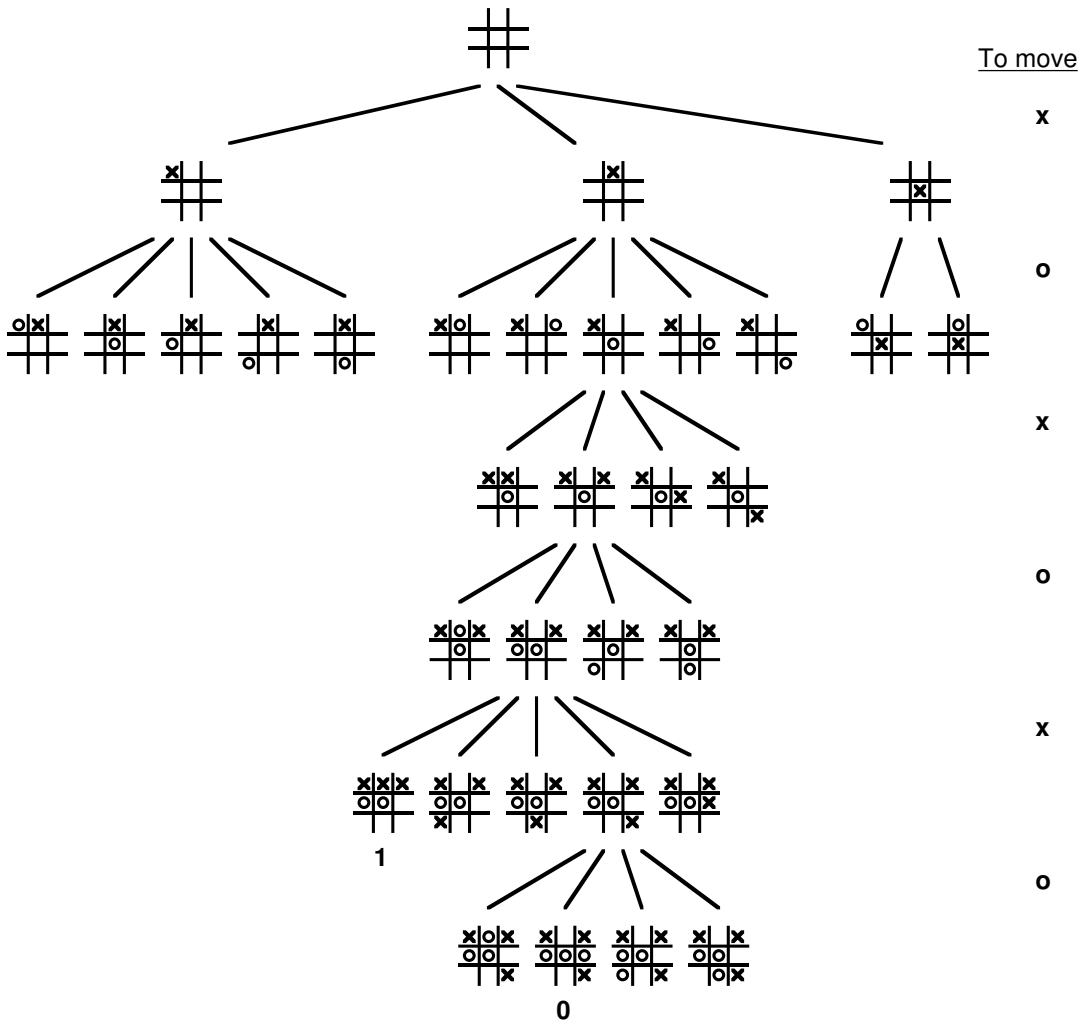

Figure 2.1: The game graph for tic-tac-toe. The entire graph is not shown, only the first two moves and two possible games (traversals from the beginning to an ending state). Positions which represent a win for X have a terminal value of 1; similarly, positions which are a loss for X have a terminal value of 0.

made the following move. Similarly, he will make his move with the knowledge that I will be maximizing my selection, etc. For a given state s with legal moves $l(s)$, this measure of worth for a move can be given by

$$
\begin{aligned}
W(s) = \max\, & m_1 \in l(s) : \\
& \min\, m_2 \in l(s_{m_1}) : \\
& \quad \max\, m_3 \in l(s_{m_1 m_2}) : \\
& \quad\quad \ddots \\
& \quad\quad\quad \min\, m_n \in l(s_{m_1 m_2 \cdots m_{n-1}}) : \\
& \quad\quad\quad\quad win(s_{m_1 m_2 \cdots m_n})
\end{aligned}
$$

where s_{abc} is the state obtained by applying moves a, b, and c to s in that order, and $W(s)$ is the probability of winning given state s, which is 1 or 0 for an ending state such as a checkmate. As given, n must be chosen large enough so that all s in win evaluations are known; that is, each is an ending state[2]. In other words, you try to *max*imize the results of your opponent *min*imizing the results of your *max*imizing the results of... whoever moves.

For ending states restricted to 1 and 0, it can be seen that the game graph can be reduced to a simple binary circuit with `AND` nodes for mins and `OR` nodes for maxes. Any other node will have a value of 0 (if one can't force a win) or 1 (if a win can be forced).

This is all simple enough but has a few conceptual problems (in addition to being absolutely infeasible). The above definition gives the worth of a move in terms of a complete search of all encounterable states, the price of which (excepting trivial games such as tic-tac-toe) would be astronomical. In practice, $W(s)$ values are estimated by some function $E(s)$ which it is hoped has some correlation with actual win values. This function E can replace W in the above formula. n can then be chosen acording to the time allowed for the program to make a move.

In the best of all possible worlds, $E(s) = W(s)$. With a perfect estimation function, no search is necessary to choose the best move. In general, the closer $E(s) \approx W(s)$, the better (for a given n) a program will play. In a worst case, $E(s)$ is very different from $W(s)$ except for end states, so that n must be virtually infinite to compensate for the poor estimations. Thus the quality of the function E is important, because it allows n to be reduced to allow feasible computation.

Evaluating W takes time exponential with n. To be illustrative, make the useful simplification that the number of moves from a given state is a constant b, called the *branchiness*. In chess, for example, there are approximately 35 moves available at a typical position. If E is evaluated for each state arising from each move on the board ($n = 1$), there will be b states to evaluate in order to choose the one with the highest E. If $n = 2$, each of the states evaluated for $n = 1$ will need to be evaluated b times for each move the opponent can apply, so b^2 evaluations are needed. In general, the time needed to evaluate for a given n is b^n.

Exponential complexity is a problem because it is impossible to avoid simply by hurling faster hardware at the program. For any constant increase in speed c, n can only be increased by $log_b\ c$, which is generally small. Nevertheless, it *is* an increase; game programs are unique in that their playing ability increases yearly without further programming effort, as the speed of the hardware on which programs are run improves. Nevertheless, this represents no long term solution: for a large b in a game like go, the increase is negligible. While there are no simple solutions to improving quality of play, two methods that do work are to try to improve E (that is, its correlation with W), and to try to reduce b by not always considering every available move.

The technique of not using every move is called *pruning*. The trick is to not prune away moves which are necessary for a correct evaluation, while removing those that are irrelevant. Even a small reduction of b can lead to a large speed increase; if b becomes $b = b/c$, for example, this will give a speed increase of $b^n/b^n = (b/b)^n = c^n$. This speed

[2]Arbitrarily defining $M(s)$, where s is an ending state, to be $\emptyset$ and $s_\emptyset = s$. The formula shown assumes n is even; if n is odd, the last term would be a max rather than a min.

can be used to search deeper (increase n) or improve the accuracy of E.

No discussion of computer games would be complete without a mention of *alpha-beta* pruning. When all conditions are right, the technique allows a $b = \sqrt{b}$, which is substantial. In practice quite this efficiency of pruning is difficult to attain, and as a consequence often much effort is put into getting the most out of alpha-beta pruning.

The essence of alpha-beta pruning is the observation that the existence of an opponent's move which is really bad at some point in the game means that one doesn't have to look deeper into the other possibilities at that point—what's bad is bad, and it doesn't matter what else the opponent *might* do because she isn't stupid and won't make another move which would be worse (for her).

The exact relationship between quality of E, pruning, and effective play has resulted in much literature[24]. In particular, there are some disturbing results indicating that for some games a poor E may not always be able to be compensated for by simply increasing n. It is clear that improving E will always improve game play. Improving E will often mean a decrease in n because a better E may be slower. However, it is important to realize that while an increase in n requires exponential time, for a given game, corresponding refinement of E may require only a small constant increase, which makes it more worthwhile. The reason that refinement of E is not done more often is that it requires human time and effort to do so, as well as expertise in the game, while n can always be increased by faster hardware and clever programming without resorting to deeper understanding of the problem. Proper pruning is very similar to refinement of E, in that it requires knowledge of the game to do well.

2.1.2 Computer chess

Application of computers to the game of chess has been, in a word, successful. There is much literature on the game and it's computerization; a good account of a recent chess playing machine can be found in [14]. Why has the computerization of chess been so successful?

Go plays a social role in the Orient similar to the role of chess in the west. Both games have a long history, large groups of devotees, and a rating scale based on competition between individuals. With a large base of experts, both games have a large traditional body of knowledge which programmers can draw from. Because chess is a western game and computers first emerged as a western enterprise, one would expect a head start in computer chess play; however, the best computer chess systems are now near the level of play of the best human chess players, whereas the best go programs are near the level of moderate beginners to the game. The reasons for this can be traced to fundamental differences in the games.

Chess is a game with a comparatively small branchiness. This allows complex tactical searches to occur; tactics dominates strategy as a factor determining chess play. A program able to search out one ply (a move) deeper in chess than its opponent has a solid and substantial advantage. Tree search of chess neatly fits in with the state-based model of game play presented in the last section, so it is easy to code.

Chess also has a simple definition of winning and losing, if not drawing. A given winning or losing position can be identified with a small (within a constant) amount of time.

With the addition of basic heuristics (such as pawn structure and control of the center), a surprisingly good chess player can be constructed using simple alpha-beta searching.

Neither of these factors (small branchiness and simple end state determination) hold true for go. Go is a very branchy game, and identification of what constitutes a winning position can be very difficult. The success of computer chess appears to have very little to contribute to computer go aside from encouragement.

2.2 The game of go

It is not the intent of this report to teach the game of go in detail; excellent books exist[3]. This section can be avoided by those already familiar with the game. A brief summary follows.

Go is a two player *perfect information* game, which means that both players have complete knowledge of the game (in contrast to most card games, where opponents' hands are unknown). It is strictly a game of skill; there is no element of chance in who wins and loses. Players alternate placing stones of their color on unoccupied positions of the 19×19 board. A *group* of stones consists of all stones of the same color which are adjacent in one of the compass point directions. A group may be captured by completely surrounding it; that is, all immediately adjacent empty positions (called liberties) must be filled with stones of the opposing color. When a group is captured its pieces are removed from the board. Figure 2.2 shows a go board; if white were to place a stone at `A`, the three adjacent black stones would have no remaining liberties and would be removed from the board.

There are multiple scoring conventions. The differences are mainly a matter of style and do not affect what defines good play. In this report I use the so-called "Chinese" method for simplicity: After both players decline to play further, the game can be scored by determining ownership of territory. An area of the board is considered to be *live* for a player if the opponent will never be able to play a stone there that will be uncapturable. Whether this is the case must be mutually agreed on by both players, who generally have sufficient understanding of the situation to agree on the ultimate capturability of given groups. If disagreements occur, the game may be continued until it becomes clear to the challenger that no points are to be gained from further play.

`B` in figure 2.2 is an example of a living group. Black cannot play in either liberties of the white group without being surrounded himself, so the white group can never be taken, even if white completely ignores the group. `C` and `D` together also have life, although this may not be as clear. `D` would have to be completely surrounded for black to play in the central position and capture the group; `C` has a similar property. Because of the two stones between the groups, `C` and `D` cannot be forcefully separated (try it), so as long as white responds in the correct manner to black's plays, `C` and `D` will remain connectable, each with a hole that cannot be filled without being completely surrounded, which can't happen as long as another such hole exists. Having two such connected holes always guarantees life, and is called *having two eyes*. For this reason `C` and `D` together are alive.

Two special cases deserve mention. *Ko* is a rule which prevents recurrence of past

[3] See appendix A.

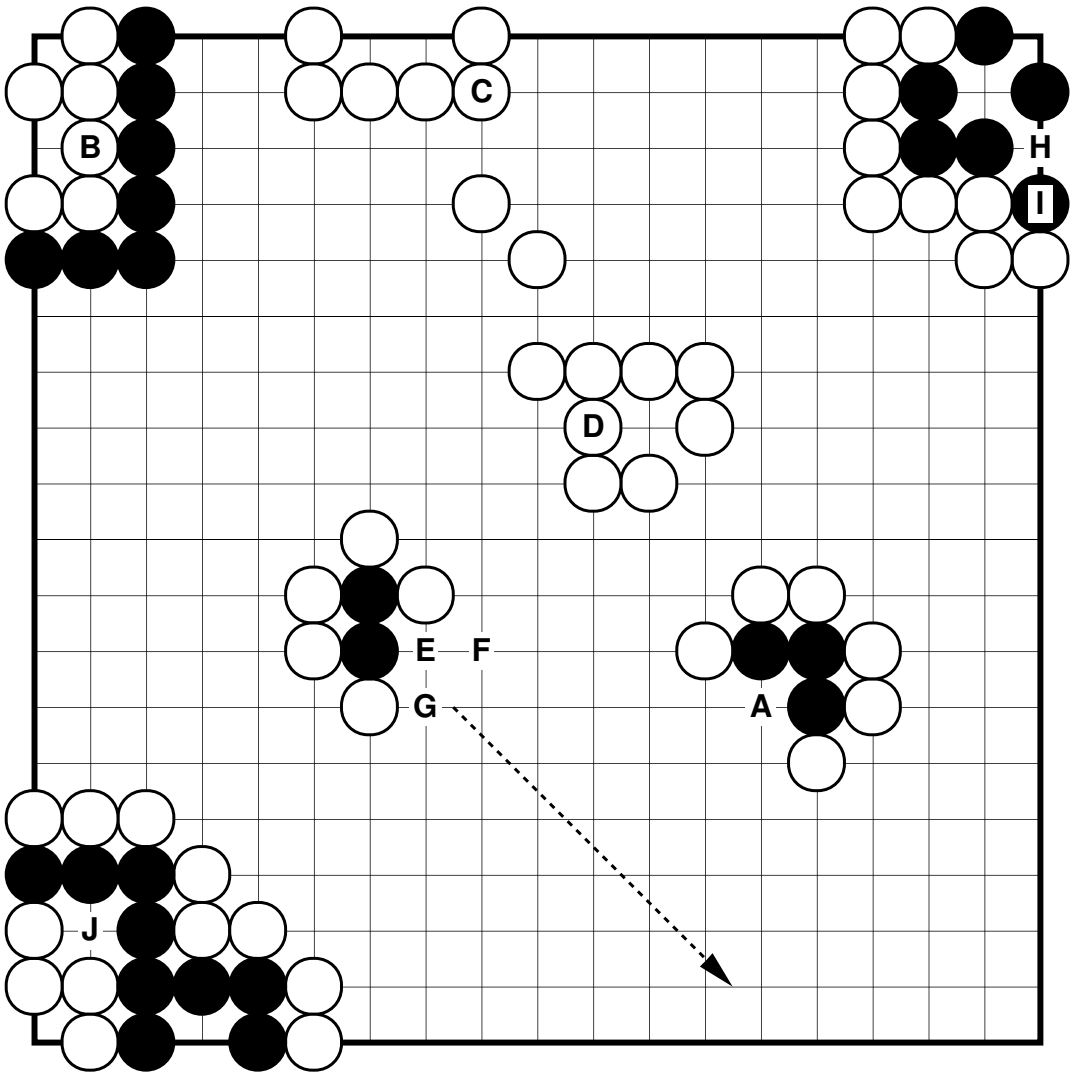

Figure 2.2: A go board with examples mentioned in the text.

board positions. I in figure 2.2 can be immediately captured if white plays at H. However, without the ko rule, there would be nothing to prevent black from immediately capturing the offending white stone by playing at I. The ko rule prohibits this sort of repetition.

The black group in the upper right corner has a big problem—it can't make two eyes, unless it can fill at H. Suppose it is white's turn. If she captures at H, black can make a play somewhere else (called a *tenuki*) because of the ko rule. She can then capture the group of three stones to the left of H, killing black's corner group. However, to do this white is effectively allowing black to make two consecutive plays elsewhere on the board—often enough to jeopardize one of white's groups. After black tenukis, white will have to decide if capturing the corner is worth the loss of a possibly larger group somewhere else. The ko rule often adds a combative element to the game, as it does here.

Seki is a condition where an area of the board is considered alive to both players, because players cannot be forced to move and thus allow capture of their own group. Having to place a stone on the board may be a disadvantage; Go, unlike checkers and chess, does not force a player to move. J is an example of seki; neither player can play at J without being immediately capturable. In this case the territory held by the black and white groups is considered alive to both.

Go is similar to chess in that a piece can have great influence on another piece far away. In chess, this is because of the nature of movement of the pieces; a rook can bound across the board in a single turn. In go, stones are immobile, but the tactical effects can sometimes be felt far away. The black group by E in figure 2.2 is dead; if black plays at E, white can play at F; the situation is now the same but one line closer to the edge of the board. If black chooses to continue running (by playing at G), he will eventually hit the edge of the board and be captured. The existence of a single black stone near the arrow would be sufficient to keep the group alive, dramatically changing the situation; white would no longer be wise to chase black, because black would eventually be free and white would have many stones on either side of black's well-connected black group which could be threatened and would be a liability.

Another example of non-locality is the living group composed of C and D. These two blocks of stones, each of which needs the other to make two eyes to live, could be much further apart. If one happened to fall or the two became disconnected, the effects could be felt across the board.

Stones have relationships to other stones. *Influence* is the effect that a stone may have at a distance on other stones; generally, having a stone of the same color nearby strengthens another stone. Highly influencing an area increases the probability that that area will remain in your control and become points at the end of the game. *Connectivity* refers to how likely two stones are to become connected. Often the connectivity of two groups determines whether both will live: if each has an eye, neither can live on their own but both can live once connected.

The unusual scoring method—consensus by the players—provides the first hurtle for computerizing Go. There is no simple test to determine who wins and loses a terminal board position, as there is in checkers (no pieces for a player) and chess (king is unconditionally capturable). Attempts have been made to modify the rules to allow a simple criterion for winning[46] but this is redefining the problem to suit the solver.

There is a similarity to the opening book in chess, called *joseki*[15]. However, because of the large size of the board, joseki take place around corners rather than the entire board. Joseki are less important to go than the opening book is to chess. Because of their unusual relationship to the rest of the board (each corner, for example, only has two adjacent positions instead of three at the sides, and four everywhere else) the corners are very important.

The value of a move is greatest near the beginning of the game and, with good play, generally declines throughout the game. Eventually a point is reached where further moves can only hurt each player; no worthwhile moves remain. Play thus continues until neither player wishes to move further. Because the nature of moves changes as the game proceeds, a game may often be conceptually divided into various stages. In the beginning game, the players roughly divide the entire board. In the middle they fight over who gets what, and

in the endgame they fight over how big each piece is. Different strategy is required for each stage. Although such stages can be identified, they are not as evident as in games like chess, and often part of the board will remain relatively untouched for most of the game, while other areas are being contested. It is perhaps more correct to refer to stages that areas of the board go through than to stages of the entire board.

Often one player will have the initiative for a number of consecutive moves, called having *sente*. Having sente allows one player to make strong threats which the opponent must respond to; in this way the player with sente can direct the action in a way which suits him, while the opponent must meekly follow him around the board.

Players are rated according to *kyu*. A difference of one kyu indicates that a handicap of an additional stone placed on the board before a game begins puts the two players at approximately equal strength. Beginning players are in the neighborhood of thirty kyu; lower kyu ratings correspond to greater skill. Players of higher caliber then 1 kyu are rated by *dan*, although this is a greater strength than computer go programs are likely to attain soon.

The theory of go was greatly advanced by governmental support of professional go players in Japan. Today in Japan there are hundreds of professional go players. Some tournaments in Japan have prizes equivalent to over $100,000. Go is very popular in China (known as *wei-chi*) and Korea (*paduk*). There are several professional players living and teaching in the United States. However, the players from Japan, Korea, China and Taiwan dominate the highest ratings of play.

2.3 Go programs

Computer go has not been around as long as computer chess. The first program to play a complete game was written in 1968 by Albert Zobrist[47]. The International Computer Go Congress, first held in 1986, has increased interest in computer go. This contest offers a prize for the top computer program and up to $1,000,000 in prizes for beating a professional Go player. The prize awarded is about $7,000 as well as the privilege to challenge a human expert. The handicap keeps going down and the prize up until the final challenge of a 7 game series for about $1.2 Million. All the handicap games are played against a competent youth challenger. The final series is against a professional player. No program has yet won against the youth player.

Properly rating a computer program is difficult, because programs can frequently be beaten by a player who has played the program before and understands its weaknesses, and yet play well against someone who has never played it before. Objective ratings are derived from tournament results[22], and typically players play each other only once. In addition, the commercial pressure for a program to have a good rating encourages rating optimism.

In this section the published work on a number of existing go programs will be examined. In part because of the competitive nature of computer go, specifics of individual programs are not always easy to obtain; in fact, because most of the work appeared to be competitively (and commercially) motivated, the few publications which exist often give only a smattering of abstract concepts, stopping short of permitting reproducibility of results. While this literature is good for stimulating thought in a potential go programmer, virtually no serious

attempt has been made to deal with the go problem as scientific inquiry, disallowing trivial variants of the game[12, 41, 42, 46] and papers on knowledge engineering methodology to which go play is tangential[7].

David Fotland's programs will be examined in greatest depth because he has been willing to make his program (as well as specific information about internals) available; in addition, many of his techniques are found in other programs, so they may be generally representative.

2.3.1 David Fotland

David Fotland currently calls his program `Many Faces of Go`, which evolved from the previous program `Cosmos`. This description of the original `Cosmos` program is adapted from a file Fotland originally distributed with the game (see appendix A) and personal communication.

Cosmos

`Cosmos` has a set of rules which suggest moves to try and probable evaluations of the moves. There are 250 major move suggestors, a joseki library, and a pattern matching library. Suggested moves are sorted and a subset selected for futher consideration. Each of these moves are applied to the board and the resulting position is evaluated and scored by evaluating connectivity, eye shape, and life and death situations. Territory is evaluated based on eye space, liberties, and influence. The move which leads to the best score is played.

Data structures Sorted lists of integers are a frequently occuring data structure, used for maintaining lists of liberties, neighboring groups, and eyepoints. Some structures, such as lists of liberties, are maintained incrementally, while others, such as the amount of influence on each square, are completely recalculated on each move. Most data structures are attached to strings (adjacent blocks of stones) or groups (logically connected strings) rather than to positions on the board. A table of important data structures is given in figure 2.3.

Evaluating a position To determine the status of each group, the following procedure is followed. Classifications of groups are shown in figure 2.4.

- Determine list of strings and the liberties associated with each string. This is done incrementally.
- For each string, see if it is tactically capturable if it moves first. If so, mark it `DEAD`.
- For each string, see if it is tactically capturable if it moves second. If so, mark it `THREATENED`.
- Determine which strings are connectable, by checking tactically if they are cuttable or not. (Knowing which strings are `DEAD` or `THREATENED` is a big help here.) Collect all of the strings that are unbreakably connected into groups.

Per point	Color (black, white, or empty) String number Number of adjacent black, white, empty points List of adjacent empty points Influence
Per string	Number of liberties List of liberties List of adjacent enemy strings Group number List of connections Aliveness
Per group	List of component strings Aliveness Number of liberties List of liberties List of eyes Number of eyes List of potential eyes
Per connection	Two string numbers Status of connection (is it cuttable?) Type of connection (hane, knight moves, etc) List of points in connection
Per eye	List of points in eye How many eyes if I move first How many eyes if enemy moves first How many eyes if enemy gets two moves in a row List of vital points Type of eye (one_point, dead_group, line, etc.)

Figure 2.3: Important data structures

- Analyze all the eyes on the board and assign them to groups. Cosmos knows some dead shapes. It checks the diagonals of small eyes to see if they are false. Figure out the value and potential of each eye. (For example, 3 empty positions in a row has a value of one eye and potential of making two eyes).
- Determine what territory is completely controlled by each group which was not already counted as eyes. This is that army's `EYESPACE`.
- For each group, if it has eyes plus sufficient `EYESPACE` to construct two eyes, mark it `ALIVE`.
- Radiate influence from the `ALIVE` groups (and negative influence from `DEAD` ones).

- For each group that is not `ALIVE` or `DEAD`, figure out how strong it is. Take into account potential connections, potential extensions along the edge, and potential eyes. If it has two independent moves that could make two eyes, mark it `MIAI-ALIVE`. If it has only one move that can make it alive, mark it `UNSETTLED`.
- The remaining groups are `WEAK`. Look for two adjacent `WEAK` groups to find semeais and sekis. See if the radiated influence from a friendly live group hits a weak group; if so it is not surrounded. Separate the `WEAK` groups that can run or fight from the ones that are hopeless. Generally a `WEAK` group that can't run and has `ALIVE` neighbors and few liberties is hopeless.
- Each point with a stone on it, adjacent to a stone, or between a stone and the edge is scored according to how alive the stone is. Other points are scored based on radiated influence.
- Unsettled groups are scored differently depending on who is to move.

The evaluation function is based almost entirely on life and death considerations.

The influence function Once the strength of groups is known, influence is radiated to determine territory. The influence function falls off linearly with distance, and does not go through stones or connections. Dead stones radiate negative influence.

The tactician The tactician does a single-minded search to try to capture a string. It is passed the maximum liberty count, maximum depth, and maximum size of the search. When seeing if strings are `DEAD` or `THREATENED`, the maximum liberty count is 4, so if a string gets 5 liberties it is assumed to escape. The maximum depth is about 100; this allows a ladder to always be read out. The size of a search is determined by the playing level of the program. Forcing moves don't count, so a ladder can be read out even when at low levels.

Move selection There are over 250 reasons for making a move; some are shown in figure 2.5. Each reason comes with a bonus value, a guess value, a minimum aliveness, and an indication of which groups are being attacked or defended, if any. The moves are sorted by the guess value and some number of them are tried (controlled by the playing level). After a move is tried it is checked to see if the rule applied. For example, if the rule is "Make an eye to save an unsettled group" and after the move is made the group is still unsettled, then the rule did not work, so it must be rejected. Similarly, if the move is an attacking move, the attacked group must end up weaker, and if the move is a defending move, the defended group must end up stronger. If the rule applied, we check to see if the move is sente, and add a bonus if it is. The position is evaluated and the rule bonus and sente bonus are added.

There is also a joseki library which can suggest joseki moves. Joseki moves are marked as normal, urgent, bad (to be ignored unless the opponent makes it), followup, and so forth.

There is a pattern matcher with 60 patterns in it. An example pattern is:

Dead	
25	Tactically captured, unconditionally
24	Presumed dead; surrounded without eyes (smothered small group)
23	Temp used for weak groups which are undecided yet
22	No eyespace or potential
21	Probably dead; some eye space or potential
20	In semeai, loses
Probably will die	
19	No running ability, some eye potential
18	Can't run, lots of eye potential, only one eye has aji to live; could be used as ko threat
17	In a semeai. Behind - aji for later
16	Poor running ability - can't live in one move
unsettled	
15	In a semeai, unsettled
14	Lives if wins ko
13	Surrounded, can live or die in one move
13	Would be alive, but tactically threatened
12	In a semeai, ahead or temporary seki
11	Unsettled—can live in one move or limp away
Alive (or settled for now)	
10	Can run away easily, no eyes
9	Can run away easily, one eye: needs two moves to make second eye
8	Can live in one move or run easily
7	Alive because wins semeai
6	Alive in seki
5	Miai for barely space for two eyes (dangerous)
4	Barely territory for two eyes (dangerous)
Very alive	
3	Miai for lots of eye space - 3 or more ways to make second eye
2	Absolutely unconditionally alive; two eyes or lots of territory

Figure 2.4: Group classification

Fuskei	Playing in empty corner Shimari and kakari Joseki moves
Big moves on edge	Towards enemy Invasions Between friendly stones
Playing in the center	Reducing moves and junction lines
Playing safe when ahead	Respond when adds to dead group
Squirming around when behind	Make a bogus invasion Try to make a hopeless group live
Pattern matching	Patterns for cutting and connecting Patterns to avoid getting surrounded
Saving a weak group	Making eyes Running Fighting semeais
Killing a weak group	Surrounding Taking eyes
Cutting and connecting	
Contact fights	Blocking Extending for liberties Hane
Ko threats	Make a big atari
Filling dame	

Figure 2.5: Reasons to suggest moves in Cosmos

```
{ /* knights move when pushing from behind */
        { BL, BL, BL, BE, BE },
        { WH, WH, EM, EM, EM },
        { EM, EM, EM, EM, EM },
        { EM, EM, EM, EM, EM } },
{ /* critical points */
        { 0,0,0,4,0 },
        { 0,1,3,0,0 },
        { 0,0,0,2,0 },
        { 0,0,0,0,0 } },
```

BL is black, WH is white, BE is black or empty, and EM is empty. Each pattern has code associated with it that gets called when a match is found. In this case the code suggests point 2 for white (but only if white has 5 liberties) and point 3 for black (unless there is a black stone on point 4). There is code for figuring out how to save or kill unsettled or threatened groups, for making walls, ko threats, and extensions, invading along the edge, and deciding which move to make in an empty corner.

The move suggestion code is easily extensible by adding more patterns or firing rules based on other criteria. The program has text associated with each rule so it can "explain" its reasons for making moves; this makes it easy to debug.

Recent improvements

Fotland has recently released a new version of his program, now called "The Many Faces of Go":

> I made two major changes to the playing algorithm. First, I replaced the pattern matcher used to suggest moves to try. The old one had 60 patterns and suggested a single move. The new one has 210 patterns and suggests a whole move tree. Second, I added limited full board lookahead. Instead of just placing a single move on the board and evaluating, MFGO plays out a sequence, evaluates at the endpoints and minimax's the scores. The sequences are suggested by the pattern library or Joseki library, or are invented based on simple contact fight rules. The program has some understanding of the value of sente now as well.

The pattern format has been considerably enhanced. All patterns are 8x8, which allows compilation to a bit array representing which positions must be white, black and/or empty; comparisons using these compiled patterns are fast because they can be done bit-parallel as words. Separate catagories exist for patterns that must be on a side or in a corner.

Each pattern may also have a number of attributes which must be true for the pattern to apply. These may refer to the number of liberties groups have, the "aliveness" of groups (found during pre-processing before patterns are applied), and connectivity.

A move tree associated with each pattern gives likely moves and their responses; an estimate of the worth (in points) of each play is provided as well. An example pattern:

```
eebb defend edge territory
eeeb
eeeB
eeee
eeWe
eeee
EN43<A20,35<A20,
23W6:34B4;45W4.33B4:24B4.
```

which is interpreted as follows: `e` means must be empty, `B` must be black (relative to the color of the current player), `b` may be black or empty, and `W` must be white. All positions not shown don't matter. The line beginning `EN...` says this patterns must be on the edge, and `N` implies this is a normal move (there are other categories, such as cutting, surrounding, endgame, etc.). `43<A20` means that the stone at position (4,3) (the `B`) must have an aliveness value less than 20; similarly, the white stone must have an aliveness less than 20. The move tree given is shown in figure 2.6 for a particular matching pattern. The worth of a move is defined as the number of points to be lost if one plays elsewhere.

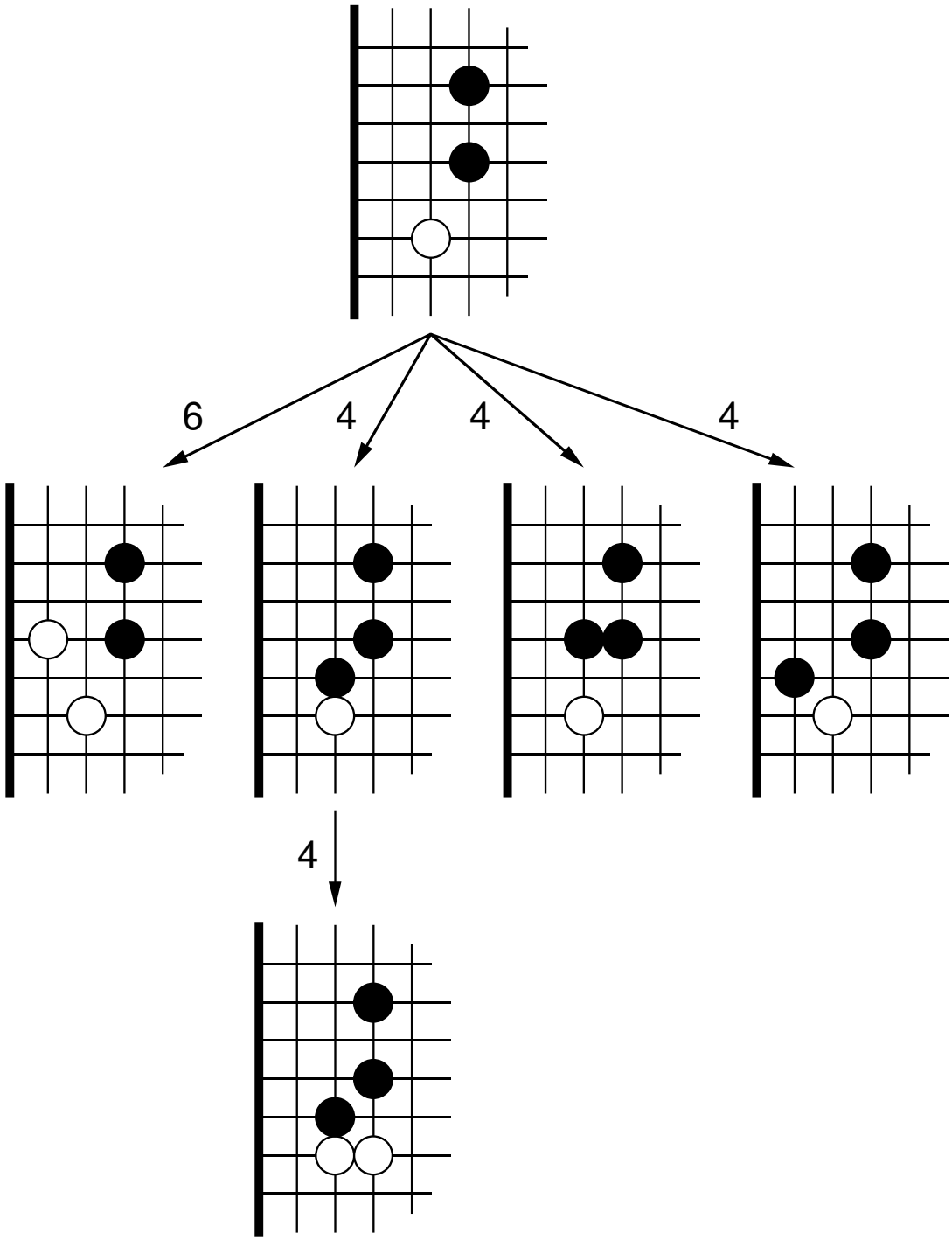

Figure 2.6: Move tree derived from pattern given in the text. The initial state is one of many to which the pattern could match.

Observations

While the pattern representation is powerful, it relies on a human to guess what set of patterns is appropriate to get the machinery of the program to play go. This requires expertise and a lot of time:

> The strongest programs (Goliath, Star of Poland, Go Intellect, Nemesis) are all written by 5 or 6 dan go players, so I'm taking lessons to try to get there myself (I'm 3 dan now, was 15 kyu when I started). Also the lessons are tax deductable.

There is no fixed methodology for deriving patterns; trial and error plays a role:

> One of the things I do now is have it select its move at each position from a professional game and compare it to the one the pro picked. If it didn't even consider the move the pro picked I manually add a new pattern to the pattern database since I can tell what's significant.

2.3.2 Ken Chen

Ken Chen of the University of North Carolina has published several papers explaining hueristics used to estimate influence and connectivity [6, 7, 8]. Chen's influence function is simple: for each position p and stone s on the board, the influence felt at p from stone s is $2^{6-d(p,s)}$, where d(p,s) is the shortest empty path from p to s. In addition, positions near the edge of the board are given special treatment; influence is more effective near the edges.

Chen goes on to define two groups as being a-connected if there exists a path between the groups for which the influence never drops below (or goes above, for white) a. For an appropriate threshhold, a-connectedness implies the groups will ultimately be connected.

It is my experience that this heuristic may be appropriate locally, but breaks down as the distance between groups increases; consider groups A, B, and C. If A and B are a-connected, and B and C are also a-connected, this implies A and C are a-connected. Certainly situations can be constructed where either A and B or B and C but not both can be connected, hence a-connectedness has some weakness as an estimator.

Chen points out a few other heuristics which can be used to determine connectivity. For example, if two groups have two or more common liberties, they can often be considered to be connected.

Chen's present program, *Go Intellect*, has about twenty specialized move generators. These, in addition to pattern matchers, suggest moves. These are combined using "certain linear combinations"[8, p. 12]. Ultimately a full board alpha-beta search is applied to select the best move. This contrasts with fast local search, associated with tactical exchanges:

> How hard is it to develop a useful Go tactics calculator? We expected to develop an interactive program capable of analyzing and solving clear-cut tactical tasks identified by an expert amateur user. This tactics calculator has proven to be much harder than anticipated, and so far is useful for only a very limited range of tactical problems: Mostly, complicated ladders and loose

ladders, i.e. forcing sequences in capturing a block that has no more than 2 or 3 liberties at any time. With hindsight, of course, we can see why this problem is hard–the interface between human strategy and machine tactics requires difficult formalization of intuitive, fuzzy concepts. To substantiate this point, we can do no better than quote the operator of Deep Thought (DT), today's strongest chess computer, as it assisted grandmaster Kevin Spraggett in an elimination match for the world championship [7]: 'I was in the fortunate position of being able to watch and assist the possibly first-time cooperation between a grandmaster and a computer.... Other problems were related to "interfacing" DT with Spraggett or his seconds. It is indeed hard to translate a question like "can black get his knight to c5?" or "can white get an attack?" or "what are black's counter chances" into a "program" for DT, since it just doesn't have these concepts. Translating back DT's evaluations and principal variations (PV) is easier, but this information is incomplete most of the time. Indeed, the alpha-beta algorithm generally gives accurate values on the PV only (other lines are cut off), whereas a human wants to know why alternatives fail.' Given the weak play of today's Go programs, a useful Go tactics calculator is far off.[7, p. 155]

2.3.3 Bruce Wilcox

In the seventies Bruce Wilcox, with William Reitman, constructed a large LISP program that played go[25, 26, 27, 43, 28]. Later this was re-written in C as a port to PCs. An interesting account of the lessons learned in writing these programs is given in [44]. Wilcox gives many examples of how going for a more complex representation to avoid recomputation (and thus enhance speed) created so much pain to the programmer that doing so ceased to be useful. This paper also gives a concise summary of representation techniques and move suggestors used by the programs.

2.3.4 Kiyoshi Shiryanagi

Shirayanagi[37, 38] has a program called `YUGO` which uses LISP expressions to symbolically manipulate go knowledge. He claims that a symbolic representation using specialized terms, notions, and pattern knowledge as primitives will be superior to any purely arithmetic (meaning linear combination of attributes) approach. The LISP expression database has to be built by hand through observation of the program's play.

2.3.5 Elwyn Berlekamp

A theory of games demonstrated in [3] can be applied to the endgame of go[45, 12]. It has the drawbacks (in its present form) of only being applicable to the late endgame of go (in fact, the "very late endgame", in which who gets the very last move is being contested) and requiring a slight modification of the rules to be applicable. Presently there is no generalization of the theory to earlier phases of the game.

2.3.6 Dragon II

The program *Dragon II* (as described in [9]) is a good example of hard coded tactics:

> Dragon uses depth-first techniques to enumerate all variations of attack and defence. If the opponent's stones have less than 5 liberties, Dragon II will attempt to attack. During the attack, it will check whether its stones surronding the enemy are in any danger. If so, it will see whether it can kill the enemy's stones to become safe. Otherwise, it will consider a long term attack...
>
> As far as variations in defence, Dragon gives priority to considering counter attack. If taking stones can eliminate the danger, the program will attempt to do so.
>
> Next, it will consider attack (thus both sides are attacking). When Dragon determines that the opponent has strings with a lower liberty count, it will try to take the opponent's liberties; otherwise it will run away...
>
> Attack and defence cause the formation of a recursive "and-or" search tree. When no defence technique provides any way to survive, then the program concludes the defense fails. On the other hand, if there exists any plausible attacking procedure, it will be deemed as a successful attack. This search process is the one that takes the longest time for Dragon II. In competition, we restrict the search to go as far as 12 levels deep, and cover at most 200 points.

The authors also have special routines for a few josekis, extension near the edge of the board, sufficient eye space, a special shape detection (such as eyes). They concede that there are a number of improvements that can be made. Dragon is, however, a good example of what a tightly coded program can do—they say 6000 lines of C code, with only 70k object code plus a 12k shapes database. This is extremely small in comparison to other programs in this section.

2.3.7 Swiss Explorer

Swiss Explorer is descended from Go Explorer, co-authored with Ken Chen. Kierulf and Nievergelt give a refreshingly honest account of games which Swiss Explorer played in "Swiss explorer blunders its way into winning the first Computer Go Olympiad"[16]:

> For the sake of measuring future progress... it may be worth recording the state of play of the "best" computer programs of 1989. In a word, it's dismal. In most games, both opponents repeatedly blundered to the tune of 50 to 100 points. The winner's secret? Avoid suffering the last calamity.
>
> Many skillful sequences were played in between blunders. We show some of these... But today's Go programs lack consistency, and their uneven level of play makes it difficult to attribute a rank. You might say 10 kyu during their "rational" moments, but bloody beginners as you observe some of the atrocious moves we choose to record for posterity.

Swiss Explorer has no full-board look-ahead and is entirely dependent on static shape analysis.

2.3.8 Star of Poland

Janusz Kraszek, the author of the program Star of Poland, has an unusual notion for programming Go. In his view, *homeostasis*, balance or equilibrium, is the important thing, and a program should be written to restore equilibrium in the game rather than overtly trying to win:

> The game of Go is probably the only one where balance and equilibrium are serious strtegic and high level concepts, not easy for beginners to understand... A sudden increase in any system is dangerous. It might do the job, but it might also destroy the system...
>
> ...Remember that the strongest Go players, the top professionals, do not play with thoughts of winning in mind. On the contrary, trying to win is one of the first things the young professionals and inseis (student professionals) have to forget about. The quicker, the better.
>
> This causes amateurs a lot of trouble in their games, and often is the main reason they lose. Why? Because... they involve too much energy into the game and positions, until it destroys them.
>
> Otake Hideo, 9 dan professional, represents the so-called intuitionist way of playing, in contrast to an analytical approach. He claims that, when he concentrates on the game, the right point for his next move seems to be burning. We can say that the whole personality of a professional Go player, his psychological and physical power–engaged in the process of the game—are proposing the next move in this way. If he disturbs the equilibrium be any slip or over extension, his inner strength would not be able to work so effectively.[18]

Kraszek goes on to describe the implementation of this in terms of a set of priorities, from the number of liberties of groups to whether the program is winning or losing, which it tries to maintain by responding to the opponent's move. From the description given it would appear that optimization would an equivalent perspective on his program: each of the points for which he tries to attain homeostasis only gets adjusted in one direction—whether one is winning or losing wouldn't have to be compensated for, if, for example, the program found itself winning by too wide a margin(!). To an implementor, talking about homeostasis is presently pointless. The concept of balance is a very important one, but may be next to useless at the trivial level of skill at which existing programs play.

It can also be argued that the reason people should not be obsessed with winning during play is psychological and not the sort of thing to which present programs are prone.

Kraszek describes the life and death analyzer of his program in [17].

2.3.9 Goliath

Mark Boon describes a pattern matcher for the program Goliath in [5]. He emphasizes speed of matching an existing pattern base, but gives no information on how this is to be used or how the database is to be created.

2.3.10 Observations

Each of the authors mentioned above has choosen a particular computer model of the game and pursued its implementation. Common features are:

- Use of an influence function, which propagates influence out from stones, decreasing with distance. This is used as a heuristic to estimate which regions of the board are likely to end up as one player or another's territory, as well as a rough measure of connectivity of groups.
- Use of a full-board alpha-beta search to discover sente-gote relationships.
- Use of local tactical search for illuminating life-and-death struggles, such as ladder reading and eye forming.
- Pattern and rule databases, for move generation and to estimate connectivity. The databases are always constructed by hand; no automated techniques are explored or methodologies provided.

It is this last point which the following chapters will expand on. If the strength of a go playing program is determined by its pattern and rule database, then after an initial framework is built, most effort will go into improving and maintaining this database. Shiryanagi's approach of providing a simple, consistent framework for the introduction of new rules into a program is a step towards recognizing this need. However, in none of the papers mentioned above is a method for objectively rating the effect a new rule on the play of a program given, and no effort has been made to automatically learn new rules or patterns in the absence of a human expert.

Chapter 3

Machines Learning Games

3.1 Samuel's checkers player

Arthur Samuel conducted a now famous set of experiments with learning applied to the game of checkers. He tried several methods, some of which are presented here.

3.1.1 Learning by rote

The relatively small search space of checkers allowed Samuel to employ the simple technique of storing each board position encountered along with the results of computed lookahead evaluations for that position. These could later be looked up and used in place of static evaluations of terminal nodes of the search tree, increasing the effective search depth.

This technique would be easy to implement today, but when Samuel was doing it (1960's) core memory was small and expensive, so much of his work involved discovering efficient means of using magnetic tape for storing the boards. After learning about 53,000 board positions, the program certainly was playing better, but was definitely not an expert[32].

The value of this technique was to increase lookahead power; by itself, the increase in lookahead was not sufficient to achieve master level checkers play (with the computational power available at the time). This indicated a need for improved static evaluations, rather than simple brute-force lookahead.

In general, learning by memorizing board positions can only work when a game is sufficiently trivial such that most occuring positions can be stored (like checkers), or if some board positions arise with great frequency, such as at the beginning of the game in chess.

3.1.2 Learning linear combinations

To improve the terminal node evaluations, Samuel investigated static evaluation functions which were linear combinations of features. Each feature was a board attribute guessed to be relevant to good checkers play; for example, total piece mobility, back row control, or control of the center of the board.

One method used to derive the proper weights for each feature amounts to a gradient-following technique: the weight vector is frozen, and a version of the program using a copy of the old weight vector plays it. This copy is updated according to differences between its evaluations of moves and an evaluation resulting from a deeper search. When the copy wins most of the games against the previous version, the improved vector is frozen and the process begins again.

There were a number of ad hoc procedures that Samuel followed to speed up the learning task and avoid convergence at local optima. Samuel had hand-coded 38 features, of which only 16 were evaluated at any time; which 16 depended on recent contributions of the feature. When a feature had a weight remain near zero, it was rotated out and replaced with one of the 22 reserve features. This was not a particularly satisfying arrangement; if two features have a high correlation, then both may be included in the linear combination although one, with a higher associated weight, would do as well. To avoid convergence at local optima, Samuel would set the largest weight to zero after apparent convergence, which would force the search into a new region.

Samuel found that stabilization would occur after about 30 such self-played games, and the result was a program that had learned basic skills but could not be considered an expert. This should not be too surprising; Samuel's features were all relatively simple, and a linear combination of weights is easy to compute but can't capture any nonlinearities of the domain.

3.1.3 Learning by signature tables

A signature table is a table with an element for every possible combination of its (discrete) inputs. Because such a table completely maps out the input space, it can potentially discover nonlinear relationships between input features. This is a substantial improvement over linear combinations, which only work well if each feature contributes independently of the others, which is often not the case. The practical disadvantage to signature tables is their large size, which grows exponentially with the number of input features.

For example, in Samuel's signature table work he pared down the input features to 24 of the better performers from previous experiments, and discretized their ranges; for example, a feature only returned a value in $\{1, 0, -1\}$ indicating whether the feature was good, neutral, or bad for a player. Even with this simplification, a signature table combining each of these inputs would have 3^{24} or nearly 3×10^{11} entries. Since this is so much larger than Samuel's 250,000 recorded board situations of master play, one could not expect to be able to fill such a table meaningfully in reasonable time.

Samuel took several steps to make the table managable. One was to split it up into a hierarchy of three levels of signature tables, as in figure 3.1. He manually divided the inputs into six subgroups, each with three inputs with the range $\{1, 0, -1\}$ and one with the range $\{2, 1, 0, -1, -2\}$. Symmetry in the input data was used to reduce the necessary table size of each of these first level tables to 68 entries. The outputs of these tables were then presented to second level signatures, whose outputs in turn were sent to the final third level table, with 225 entries.

In general, training signature table hierarchies is difficult; the same improved nonlinear representational power that makes them useful guarentees that gradients will be hard to

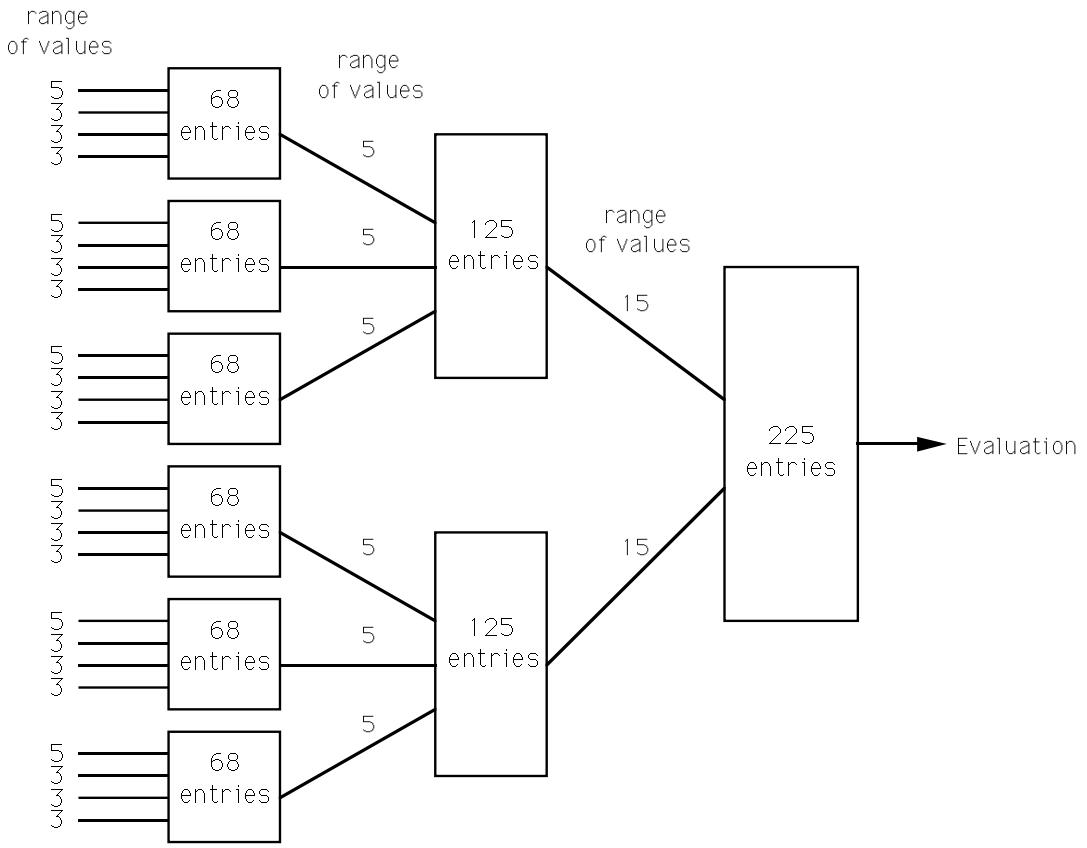

Figure 3.1: Samuel's signature table hierarchy. An arbitrary combination of feature inputs are merged in three levels to ultimately produce a single evaluation which is a function of each input.

determine. Nevertheless, there are simple techniques which can be used to find local optima. Samuel did the following: for each element of each signature table, a separate count was kept for agreements and disagreements with the masters' games. The output of a signature table was computed to be a correlation coefficient which corresponded to what extent that element was being selected for book moves. By restricting the output of each signature table to have a correlation with the output of the third-level signature tabel, Samuel greatly constrained the problem. Another technique used was interpolation of values in elements which were not frequently accessed from nearby elements as a form of generalization, employed in the early stages of training when the tables were not yet reliable.

A final optimization was to use seven different signature table hierarchies for different stages of the game, since endgame play in checkers is substantially different than early play.

The signature table approach performed much better than the earlier approaches. After training on 173,989 book moves, the program was able to correctly predict the master players' moves 38 percent of the time in a set of 895 moves not used in the training, *without employing lookahead*[33].

3.2 Tesauro's backgammon player

3.2.1 Backgammon

Since Samuel's pioneering efforts, there has been relatively little exploration of learning in games. However, there has been recent success in applying ML to backgammon. Backgammon is a different sort of game than checkers:

> The choice of a particular game of study should depend on the particular types of skills required for successful play. We make a distinction between two fundamentally different kinds of skills. First, there is the ability to "look ahead," i.e., to work out the future consequences of a current state, either by exhaustive tree search, or by more symbolic, deductive reasoning. Secondly, there is "judgmental" ability to accurately estimate the value of a current board state based on the patterns or features present, without explicitly calculating the future outcome. The game of backgammon is unusual amongst games because the judgmental aspects predominate, in comparison with other games like chess that often require lookahead to great depth. The principal reason for this is the probabilistic element in backgammon: each move depends on a roll of the dice. There are 21 distinct dice rolls and arouns 20 possible legal moves for each roll. Tree search algorithms would thus be inappropriate, since the average branching factor at each ply of the search would be about 400. As a consequence most backgammon programs and most humans do not search more than one or two ply, but rather rely on pattern recognition and judgmental skills. [40, p. 358]

Backgammon is similar to go in that branchiness forbids exhaustive searches; go, however, is strictly a game of skill: the branches are all potential moves which a player may choose, rather than a set of possibilities which the role of the dice will determine. Like backgammon, judgemental aspects predominate until contact between opposing groups begins, at which point local tactical search is important.

3.2.2 Neural Nets

Tesauro's backgammon player plays by assigning values to moves. The greatest value among the legal moves is the one chosen to play. A *neural net* was used to learn these evaluations from a database containing moves paired with expert evaluations of their relative worths. Neural nets, in this context, refer to directed graphs of functional elements, in which each edge has a "weight", or value, associated with it. Each node computes some nonlinear (usually sigmoidal, or s-shaped) function of the sum of the products for each edge of the output at the tail of the edge and the corresponding weight. Such networks have desirable properties, such as an elegant means for computing a gradient (known as *backpropagation*) of the output with respect to the edge weights, and the theoretical ability to encode arbitrarily complex functions.

In this case, what the network learns is a correspondence between the moves in the database and their evaluations by experts. Specifically, the gradient followed sought to

minimize the sum of the squares of the differences between the evaluations of the network and the expert. This was assumed to correlate with better backgammon play; with some caveats, it did. Tesauro included specific examples in the database to correct for shortcomings observed during play.

3.2.3 Features

Using a network for learning poses a number of difficulties, including what the best network topology is and what the input encoding should look like. Tesauro added a number of backgammon-specific feature codings to the other information representing the board to increase the information immediately available to the net. He found that this additional information was very important to successful learning by the net:

> Our current coding scheme uses the following eight features: (1) pip count, (2) degree of contact, (3) number of points occupied in the opponent's inner board, (4) number of points occupied in the opponent's inner board, (5) total number of men in the opponent's inner board, (6) the presence of a "prime" formation, (7) blot exposure (the probability that a blot can be hit), and (8) strength of blockade (the probability that a man trapped behind an enemy blockade can escape). Readers familiar with the game will recognize that these are all conceptually simple features, which provide an elementary basis for describing board positions in a way that humans find relevant. However, the information provided by the features is still far removed from the actual computation of which move to make. Thus we are not "cheating" by explicitly giving the network information very close to the final answer in the input encoding. (We might add that we do not know how to "cheat" even if we wanted to.) The first six features are trivial to calculate. The last two features, while conceptually simple, require a somewhat intricate sequential calculation. The blot exposure in particular... would be a very high order Boolean predicate... and we suspect that the network might have trouble learning such computations. [40, p. 368]

3.3 Generalities

3.3.1 Signature tables vs. Neural Nets

There is an important similarity between signature tables and neural nets. A simple signature table reserves a distinct value for each possible input. A single layer neural net (that is, one with a single set of edges connecting the input and outputs) computes essentially a simple function of a linear combination of the inputs, which is determined by the weights on the edges. If preprocessing of the input is done to transform all inputs into a zero vector with a single one (a classification), then this performs the same function as a signature table; it associates a single value with each possible classification.

A signature table thus can be trained with the same gradient techniques applicable to neural nets. There is an important difference, however; when the input is transformed into a classification, then only one edge can affect the linear combination of inputs to the output

unit, and the gradient for those other edges is guaranteed to be zero. This means that no other edges need to be considered for changing. Because of this, the computation required for a given training instance does not scale with the size of the table as it does for the net. For this reason, signature tables may be more appropriate representations for sequential machines, *if* a satisfying classification can be found.

Samuel used multiple level signature tables as well. A multiple level table presents a problem because a gradient cannot be computed by backpropagation; there exists a discontinuity between successive levels, when the information is being, in effect, re-classified. This makes the error with respect to a given table entry non-differentiable. Nevertheless, it is possible to construct systems for which an approximate gradient exists, as Samuel did, and still be able to achieve learning.

3.3.2 Importance of representation

Changing the input into a classification, by making it a vector of zeros with a single one, is a change in representation. The representation of the data presented to a net can have a substantial effect on the net's ability to learn; by changing the represention, one is implicitly changing the generalization bias of the system. Tesauro's net, for example, required the addition of specialized backgammon features to the raw data in order to learn to play backgammon well.

There is a danger in using any learning technique of considering it a kind of magic black box. Not understanding the learning mechanism implies that the generalization bias is also not likely to be well understood; this leaves open the possibility that the kind of generalization of the system is completely inappropriate to the problem.

Automatic definition of what may be appropriate features for a given domain remains very much an unsolved problem. In a layered neural net, one way of looking at the function of intermediate "hidden" layers is that they are changing the representation from layer to layer. In this case, the representations are still constrained to be linear functions of previous features. When there is a good notion of what may be good features to add, they can be added explicitly, as both Samuel and Tesauro did. In particular, if a feature is believed to be essential to good play, then adding it as an input saves the training technique from having to learn to perform this computation.

Chapter 4

A Go Metric

4.1 Defining an error function

Good go play involves rapid selection of a small number of good moves which are further polished to determine which in particular is the best. The branchiness of go necessitates an ability to prune away most inferior moves quickly so that lookahead can proceed. In this section I define a metric that can be used to evaluate heuristics which are meant to quickly strip away the large portion of moves which will not need to be considered for further analysis.

4.1.1 Expert game data

257 games between go players of exceptional ability were used to define an error function for playing go. Of these, 136 were games from *Go World on Disk*, machine readable game transcriptions appearing in the publication *Go World*[1].

Of these, the majority were from tournament games played between professionals or amateurs of high caliber. Many of these games' records contained commentaries as published by *Go World*, meant to be tutorial. For training purposes these commentaries were ignored. The other games were taking from `USENET` postings; these were collections of games by Go Seigen and Shusaku, both legendary go players.

Raw game data consists of lists of moves. Often there is no information given on final score or outcome of the game: some game records stop when one player resigns, some when the recorder loses interest, and some when both players elect to pass. Because there is (at the time of writing) no universally accepted format for game records, the information about the end of the game is often only found in comments meant for human eyes only, when it is given at all. I assigned no significance to the end of a game record. As a result, proper passing and resignation are not properly learned.

This is a liability, because exactly when passing occurs is worthwhile information and important to proper play. The onset of passing indicates the dip of worthwhileness of moves

[1] Information on this and other mentioned publications as well as FTP information can be found in appendix A.

below zero points; a good go player will never make moves which are worse than simply passing. It could be corrected by modifying the game records.

4.1.2 Simple rank statistic

A simple measure of error, for a given program, might be how many moves in the database were incorrectly predicted. For example, if out of a 30,000 move database, a program decides on the move the human played for 3000 moves, then the error might be $27/30$, or 0.9. Increased predictive ability then translates to a lower error.

This metric is easy to compute; Tesauro and Samuel both used it as a metric. After testing with some simple heuristics, it was found that for go, the failure rate is so large that very few moves actually succeed. For example, the aforementioned program `Gnugo` predicts correctly only four percent or so moves. Even professional players often will disagree about exactly which move is the best. However, they will almost always agree on the relative worth of moves; although one may feel a certain move is slightly better than another, there is usually complete agreement about the few moves which are likely candidates for further consideration.

Similarly, commentaries on go games often suggest a few good alternative lines of play. The notion of the 'best' move is not so cut-and-dried as in other games with less branchiness.

To take advantage of this, error functions were examined which took into account not only what was considered the *best* move on the board but the relative worth of different moves on the board. There are a number of ways to do this. For example, given that the program assigns values to each legal move (representing the point value of playing at a spot relative to simply passing, the greatest of which would be what should be selected during play), a possible error function for a heuristic h could be:

$$E(h) = \sum_{s \in S} \sum_{m \in l(s)} h(m, s) - h(M(s), s)$$

where S is the set of board situations found in the game database, $M(s)$ is the actual move played at situation s, $l(s)$ is the set of legal moves at situation s, and $h(m, s)$ is the evaluation of move m by heuristic h with the board at state s.[2] The intent is to assign some error whenever a move which was unplayed receives a more favorable evaluation then the 'best' move (which the human played), and give credit whenever a move which was not selected is evaluated as worse.

This error function has a number of liabilities. It meets the basic criteria of achieving the minimal score for a heuristic which perfectly predicts each of the moves in the database. However, it also has a trivial perfect solution for a heuristic which rates every move the same. In addition, the sum of the term h(M(s)) is a constant and can't be helping. How

[2]Generally I am considering moves as only positions, which requires knowledge of the board on which they are played to form any kind of evaluation.

about changing from subtraction to division?

$$E(h) = \sum_{s \in S} \sum_{m \in l(s)} \frac{h(m,s)}{h(M(s),s)}$$

There is another sort of liability. To have an arbitrarily low error function, all a heuristic needs to do is to rate a single unselected move as very, very bad. In other words, because there is no restriction on the contribution that a single move evaluation can make to the total error function, a heuristic can 'make up' for anything by giving an arbitrarily bad rating to any unselected move.

This can be corrected by adding some nonlinearity:

$$E(h) = \sum_{s \in S} \sum_{m \in l(s)} e^{\frac{h(m,s)}{h(M(s),s)}}$$

which prevents compensation. Many other functions besides e^x could be used; for instance, another legitimate error function is:

$$E(h) = \sum_{s \in S} \sum_{m \in l(s)} \begin{cases} 1 & \text{if } h(m,s) \geq h(M(s),s) \\ 0 & otherwise \end{cases}$$

which amounts to counting the rank of the selected move among the set of legal moves after evaluation. This is very robust, because is is *nonparametric*: it doesn't care about whether a value is two times or five times or a hundred times the value of another, only which one is of greater magnitude. This is in some sense very appropriate, because move selection is done in a nonparametric way; the move with the greatest evaluation is the one to play, regardless of the relative values of the other moves.

4.1.3 Normalized rank statistic

A problem still remains: some board positions have more legal moves than others. If the ranks are added up willy-nilly, greater error may be produced by those situations occuring at the beginning of the game than at the end, because there are more moves to play; there can easily be three times as many potential moves at the beginning of the game as at the end. To deal with this, two changes are made to produce the final error function:

$$E(h) = \sum_{s \in S} \frac{\sum_{m \in l(s)} \begin{cases} 1 & \text{if } h(m,s) > h(M(s),s) \\ .5 & \text{if } h(m,s) = h(M(s),s) \\ 0 & \text{otherwise} \end{cases}}{|l(s)|}$$

One change is the addition of the $=$ condition, which allows evenly distributing error for positions of symmetry such as often occur at the beginning of the game. .5 was chosen so that moves rated the same as the correct move can share equal blame with it, together contributing the same error as a single incorrect evaluation. One cannot simply change the

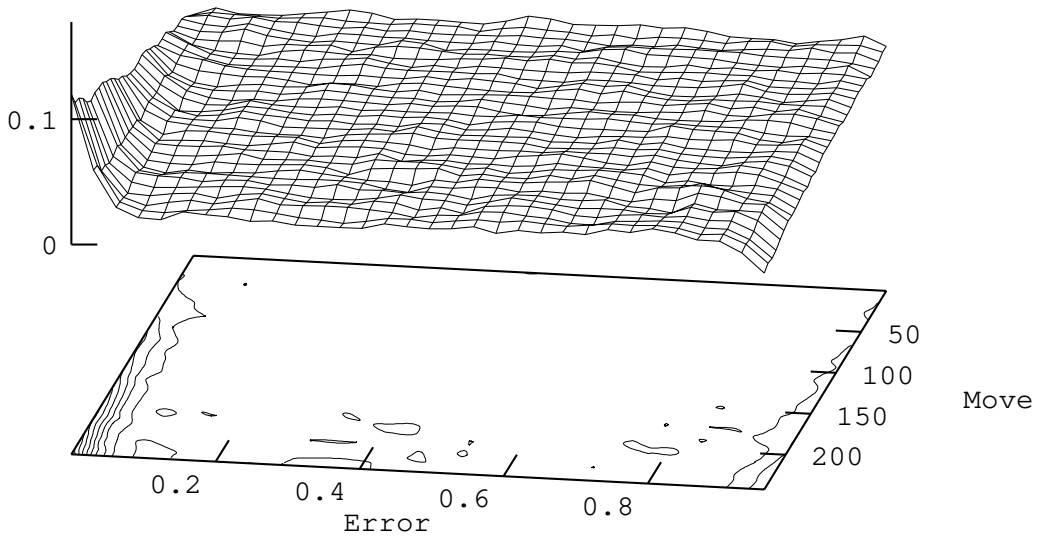

Figure 4.1: An NRM (see text) plot for a simple "greedy" opponent. All NRM plots in this report are shown with corresponding contour map.

$\geq$ to a $>$ because the trivial solution (with all evaluations the same) would become feasible. The other change is the normalization of error to the number of legal moves which exist on the board at the time.

Note that an exactly zero error is not possible; $h(m, s)$ must equal $h(M(s), s)$ for at least the case when $m = M(s)$. However, this slight deviation is expected to be very small ($.5/|l(s)|$ for each board s) and therefore negligible except at high skill levels.

4.2 Plotting

There are many plots in this report. Some heuristics display different behavior at different stages of the game, or subtly change the way learning occurs as time progresses. To illustrate the effectiveness (or lack thereof) of various heuristics, it is useful to plot the error against other variables, such as time or the move number.

An NRM (Normalized Rank vs. Move) plot shows, for a given heuristic, how that heuristic behaves as the game progresses. For each move number, a distribution is shown of how the error fell from 0-1 for each move evaluation. This can be illuminating, because the nature of go changes a *lot* from the beginning of the game to the end.

Figure 4.1 shows an NRM plot for an opponent who captures whenever possible, and when no capture is possible, rates all moves identically. Compare with figure 4.2, which is the response of an opponent who randomly evaluates each move. The greedy player can correctly identify a small percentage of moves (the captures) which are usually found towards the end of the game; this results in a progressive curling of the error distribution

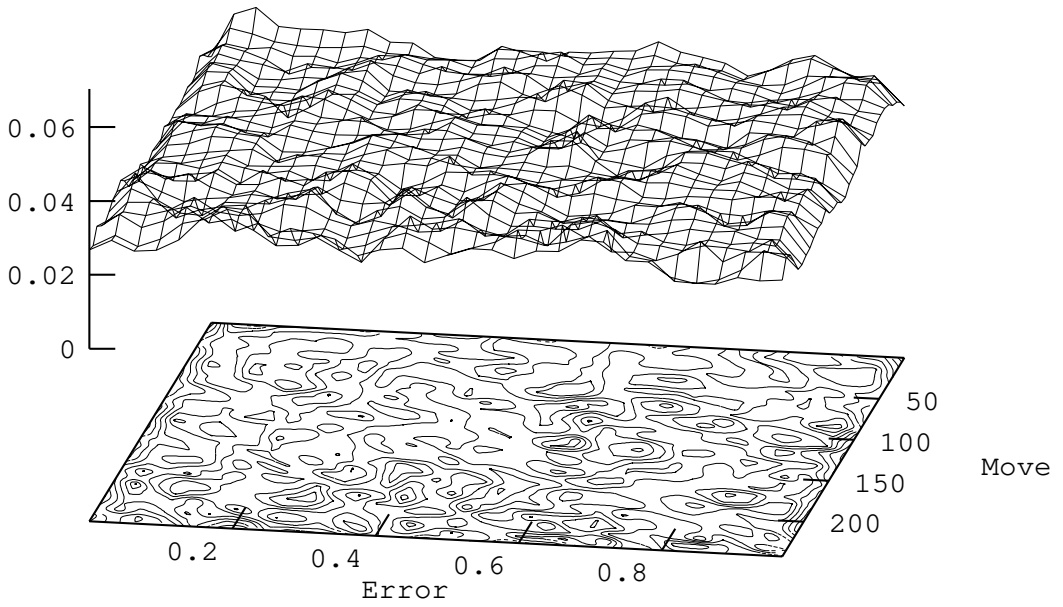

Figure 4.2: An NRM plot for a random opponent.

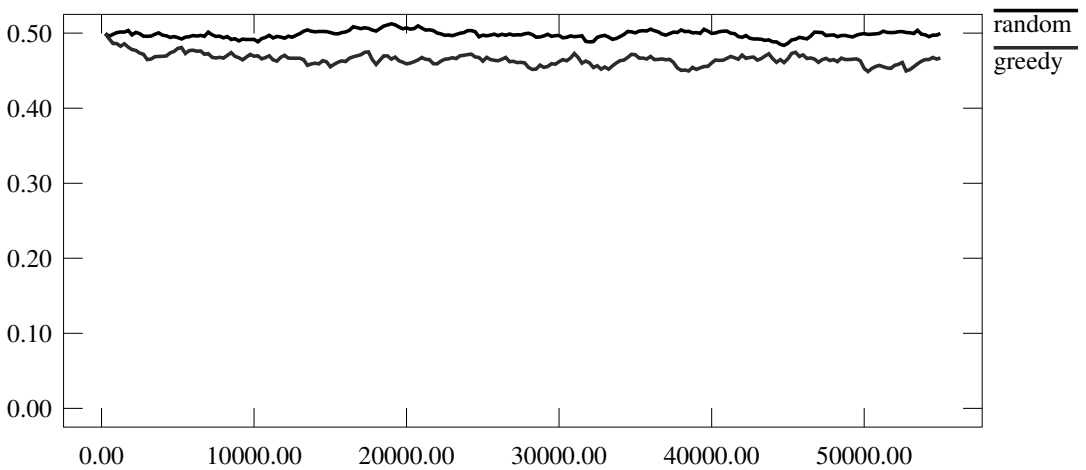

Figure 4.3: Study plot for the greedy and random opponents.

towards zero, when moves are precisely predicted. Similarly, there is a depression on the other end, towards one; this is caused by the removal of the really poor suicide moves from consideration. However, almost no discrimination is shown at the beginning of the game when life and death considerations are not important. A heuristic which rated every move the same has no discriminating ability, so its NRM plot is completely flat. Similarly, the random opponent has an evenly dispersed error as the game progresses; it is equally bad all the time, only with local bumps.

A *study* plot shows how the performance of a learning technique reduces the error over

time. The x axis is the iteration number (how many moves were previously studied) and the y axis is the the error, averaged over time (see appendix B for details on the generation of study plots). Figure 4.3 is a study plot of the data in NRM plots 4.1 and 4.2. This particular plot is not very interesting because no learning occurs with time, however, it is provided as reference for comparison with later study plots.

4.3 Evaluation of performance

There are a number of ways of assessing a program's performance (from [40]):

- Performance of the error function on the training data. This gives very little indication of the program's ability to generalize; hence it is not very useful. A program can always simply memorize the input data without performing any useful generalization which can be applied to new situations.

- Performance on a set of test data which was not used to train. This is not necessarily related to absolute ability of the program during a real game; some unforseen quirk of the error function might jeopardize this. Both Tesauro and Samuel made use of this as a metric of system strength; I did as well.

  If a study plot covers only one pass through the database, then each datum is seen only once. The error reported is for the initial response to the data, not that after learning, so the learning curve is representative of how the heuristic performs on new data.

- Performance in game play against a human expert. This will provide the most accurate assessment of the strength as a player, and allows detailed qualitative description of the play. This method is extremely time-consuming, and the understanding that arises is primarily qualitative.

- Performance in actual game play against a conventional computer program. This allows more games to be played out than against human players, but is plagued with other problems. If one wishes to observe the performance over the course of a learning run, this is very time-consuming. For go in particular, there is no easily programmable determination of the score at the end of the game; the few public go programs that are adaptable for machine play all have different protocols or run on different hardware; and the generally low skill of existing go programs also means that an improved performance could be due to exploitation of weaknesses of specific programs that are not generalizable to the general population of go players.

Chapter 5

Pattern Preference

5.1 Optimization

Go players classify positions on the board to describe relevant information about that position. For example, the empty position adjacent to a group with a single liberty (like figure 2.2, A) has a completely different effect on the structure of the board and the potential interaction between groups than an empty position comprising an eye of a group with only two eyes (like figure 2.2, B). In the first case, both players have an interest in playing on the position; one player would capture the group, and the other would possibly increase the number of liberties and make it uncapturable. The second case neither player would wish to play—the owner of the group with the eye would gain nothing by filling in an eye, and for the opposing player such a move would be suicide.

To utilize the error function defined in section 4.1, a function needs to be created which returns a value for each position on the board, such that the relative values of the positions indicates a ranking preference. Compare, for example, Fotland's use of patterns (section 2.3.1) to generate estimates of the worth of particular moves; numeric estimates of the worth of particular moves are given, so that all but a few of the best ranking moves can be immediately pruned and not considered for further analysis.

What I have dubbed the *pattern preference* approach is actually the definition of two functions; one identifies a class for any position on a board, and another returns a value for each such possible classification. More formally: a classification function $C(m \in \text{M(b)}, b \in \text{state}) \in \text{class}$ and a value function $V(c \in \text{class}) \in \text{real}$ so that $V(C(m, b))$ is an estimate of the worth of playing move m on board b. This is essentially a signature table approach, but with an arbitrarily complex procedure for finding which element of the table is appropriate.

It is worth pointing out that the choice of a nonparametric error function implies that worth is not an estimate of points not gained if the move is not played (a tenuki or pass), which is the usual meaning of worth. Only the relative ranking of $V(C(m, b))$ counts and not any absolute measure, aside from the worth of a pass, which is fixed at zero. It would be possible to enforce an ulterior mapping such as average point worth of moves by fixing the value of the classifications of moves with known point values.

The reason for dividing the problem into C and V is because, given some *a priori* classification C, an optimal V can be determined by analytic and/or numeric techniques.

For example, if V is looked at as an associative map from the class set to reals, the error function defined in section 4.1 is

$$E(h) = \sum_{s \in S} \frac{\sum_{m \in l(s)} \begin{cases} 1 & \text{if } V(C(m,s)) > V(C(M(s),s)) \\ .5 & \text{if } V(C(m,s)) = V(C(M(s),s)) \\ 0 & \text{otherwise} \end{cases}}{|l(s)|}$$

and almost has a gradient with respect to a class c ($dE(h)/dV(c)$) given by

$$\sum_{s \in S} \sum_{m \in l(s)} \frac{\left\{ \begin{array}{ll} 1 & \text{if } V(C(m,s)) > V(C(M(s),s)) \\ .5 & \text{if } V(C(m,s)) = V(C(M(s),s)) \\ 0 & \text{otherwise} \end{array} \right\} \left\{ \begin{array}{ll} -1 & \text{if } c = C(M(s),s) \\ 1 & \text{if } c = C(m,s) \\ 0 & \text{if } C(m,s) = C(M(s),s) \\ 0 & \text{otherwise} \end{array} \right\}}{|l(s)|}$$

Why *almost* a gradient? Because the error function is nonparametric, it will not vary for arbitrarily small changes in $V(c)$. The "gradient" given above it not really the gradient of the defined error function at all, but of a similar "smooth" error function which is differentiable. Following the above as if it were the actual gradient by subtracting the gradient from the current position in state space repeatedly (a simple technique often used with complex functions, such as those represented by neural nets, for which a gradient is derivable) has worked quite well in practice.

The similarity between signature tables and neural nets can be taken another step: for simple C there is a simple construction operation that can transform any trained (C,V) pair into a two layer neural net which will have the same performance. The first layer performs the classification operation of C by defining a node for each class, where each edge corresponds to the necessary inputs for that class. A sufficiently step-like activation function enforces the restriction that a single node be active at a time. The second layer, actually just a single node, implements the function V.

The existence of such a construction raises the possibility that the pattern preference technique could be used to quickly initialize a net with reasonable edge values, guarenteed to have a known minimum performance level, which could be further trained by backpropagation to improve the discrimination of the first "classifying" layer. Unfortunately such a construction could be taken only in one way, from the fast pattern preference technique to the computationally expensive neural net.

Other optimization techniques besides gradient following could be used as well. For instance, an early technique I used was to consider each class as a node of a directed graph. Each time a class is found superior to another (in the sense that the class of the move actually played is *superior to* the class of another move which also happens to be playable), then an edge is added to represent this relationship. A partial ordering of this graph then produces a possible ranking which attempts to satisfy as many of the edges as possible. A big problem is that the constructed graph is invariably degenerate, with many cycles and disconnected subgraphs. Another problem is that the partial order of a graph is not easy to compute incrementally; in general, adding another edge and recomputing the partial order

may take too much time. One might also question whether storing the entire superiority graph is a reasonable kind a learning—isn't it very similar to storing every exemplar? Edges represent an "on-off" condition: either a node is superior to another or it is not; in general, this is not always true for general classifier functions, and some kind of weighted edges would be more appropriate. For all these reasons, taking the partial order of the superiority graph is not an acceptable general solution.

Consider a numeric description of the constraints of a partial order on the superiority graph. It can be seen that the partial ordering problem of finding a solution to a system of linear inequalities of the form

$$\begin{array}{rcl} V(c_1) & > & V(c_2) \\ V(c_3) & > & V(c_4) \\ V(c_5) & > & V(c_6) \\ & \vdots & \end{array}$$

where each inequality represents a class found superior to another, as in the directed graph. This can be transformed to

$$\begin{array}{rcl} V(c_1) - V(c_2) & > & 0 \\ V(c_3) - V(c_4) & > & 0 \\ V(c_5) - V(c_6) & > & 0 \\ & \vdots & \end{array}$$

and brought to the matrix form

$$\mathbf{M} \begin{bmatrix} V(c_1) \\ V(c_2) \\ V(c_3) \\ \vdots \\ V(c_n) \end{bmatrix} > \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

where $\mathbf{M}$ is a matrix of coefficients, derived from the superiority observations. Cycles in the partial order problem are thus equivalent to overconstraining such a linear system. One approach to dealing with this is to reformulate as a minimization rather than a constraint satisfaction problem; an example would be to let the error be the sum of the squares of the elements of the right side matrix.

The error functions defined in section 4.1 are almost as simple; the unnormalized rank error function corresponds to minimizing the sum of the signums of the elements rather than the squares; the normalized rank error further would divide each of these by the number of legal moves on the board at the time. Each of these has a derivable gradient, or something closely approximating it; the presence of such a gradient allows the simple technique of gradient-following to be applied.

5.2 Simple methods of categorization

An obvious method of classification is to select a set of positions relative to the position of interest, and have every possible combination of stones that can be observed in those

positions represent a class. For instance, if the four positions immediately adjacent to the position of interest were chosen, each of these positions might be defined to have four possible states: black stone, white stone, empty, and being off the board.[1] In this case, there would be $4^4 = 64$ classes, minus those which are the equivalent due to symmetry.

How should such windows of positions be chosen? Positions closer to the position of interest are more likely to have an effect on which classification will be appropriate than positions further away. If a physical interpretation is made of what constitutes "closeness", the square (all positions within a constant w horizontally or vertically) and diamond (all positions within w by the Manhattan distance [the sum of horizontal offset vertical offset]) shaped windows of figure 5.1 naturally arise.

If two positions under a window can be shown to be equivalent by rotation or reflection, then they exhibit symmetry. All the fixed windows used in this report correctly classify symmetric patterns together. Figure 5.2 shows an NRM plot for a simple 3×3 window centered over the move in question as well as for a radius 1 diamond shaped window, really just the four adjacent positions to the considered move. In each, a clear effect of the dependence on locality for classification is seen at the beginning of the game. A moment's reflection explains this phenomenon. At first, the entire board is empty and featureless; when a player places a stone, it now has a feature by which the nearby moves can be categorized. A very small window, such as the radius 1 windows, exaggerates this problem—the inner 17×17 square of stones away from the edges *has* to be identically categorized, with the expected negative results.

Figure 5.3 shows how the error associated with the beginning of the game diminishes as the radius of the window increases. Figure 5.4 show the study plots of these windows. Radius one is too small to be of any worth, and radius two does much better. However, radius three already suffers from explosion, and radius four only makes it worse. From the NRM plots, however, we can see that this is not true for all kinds of moves.

Figures 5.5 and 5.6 are the same thing for square windows rather than diamond shaped. The square window approaches having twice the number of observable positions; because of this the window is more prone to exponential explosion. The 9×9 window is shown continuing on a second pass through the database; after a single training presentation, nearly perfect memorization has occurred. However, this is at the expense of generality; the performance on new data (the high error before the sudden drop) is much worse than that with smaller windows. I found that repeated training on a set of data did not significantly improve the performance on new untrained data.

Instead of classifications based on fixed windows, it is also possible to have classifications which are made on a dynamic set of positions, that is, a change in a particular position need not change the classification. For example, the graph based windows are constructed by first constructing a graph of the board with adjacent stones of a single color lumped into a single node; edges on the graph then represent adjacency in a more abstract sense. This window type was tried because it is closer to the human perception of the games; connected groups of stones always live and die as a unit, so shouldn't grouping them as units in the window be more appropriate for discovering tactical relationships between groups?

[1] A ko position (where it is empty but still illegal to play) could also be considered a state; however, this is rare and has not used for classification here.

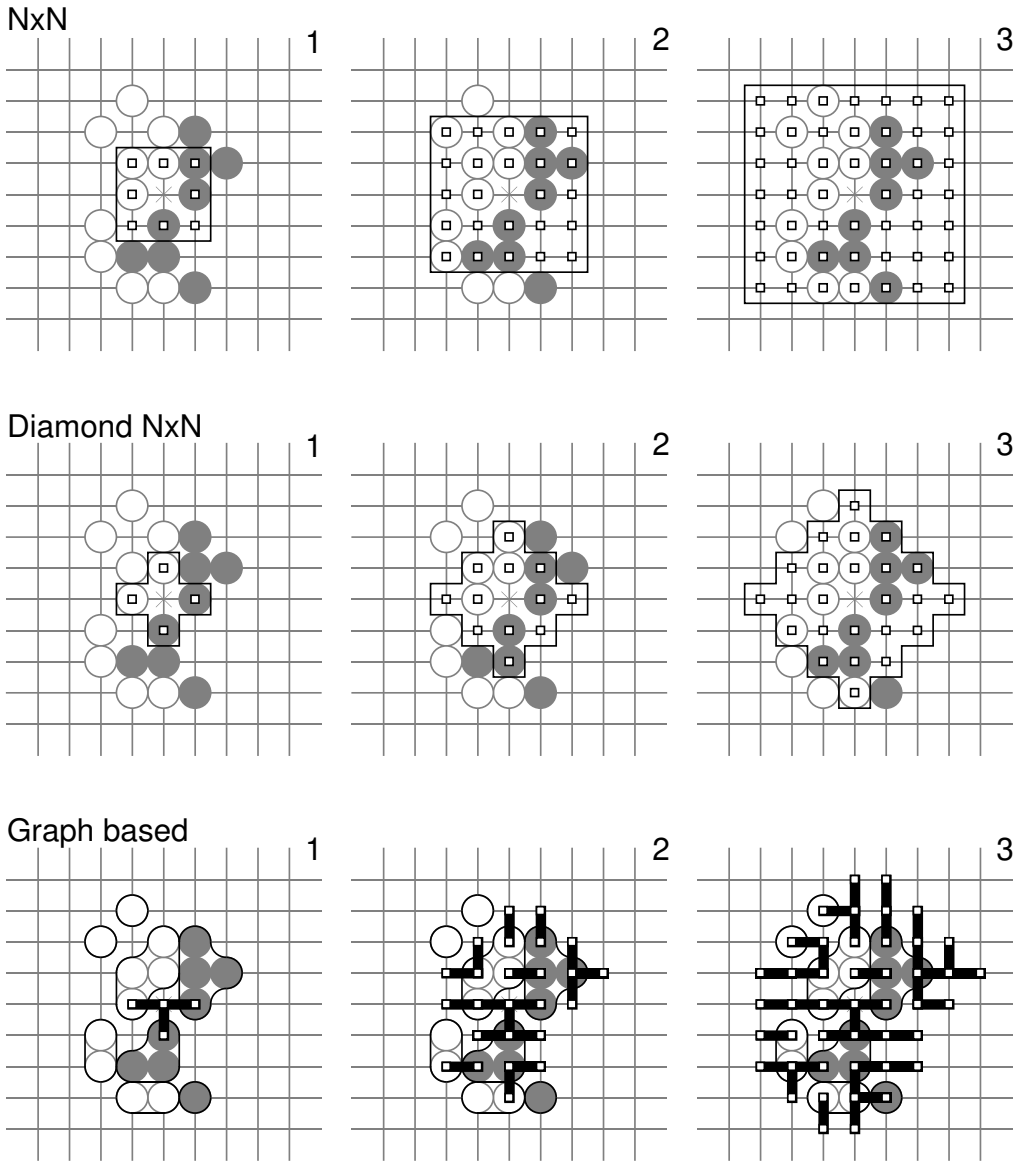

Figure 5.1: Windows used for pattern extraction.

The radius of such a graph-based window is the number of edges that can be traversed and still remain in the subgraph which is extracted. Figure 5.8 shows the study plots obtained for graph-based extraction of small radius.

It is suspected that the poor performance of the graph-based window is that it is very susceptible to explosive complexity, because a large group can have a very large and variable number of adjacent nodes, and the graph-based window as I have proposed it is unable to simply categorize large, strong groups together; perhaps some method of dynamically controlling the extracted subgraph on a better basis than radius would improve the extraction.

The reader familiar with the classifier literature will wonder why I have taken the

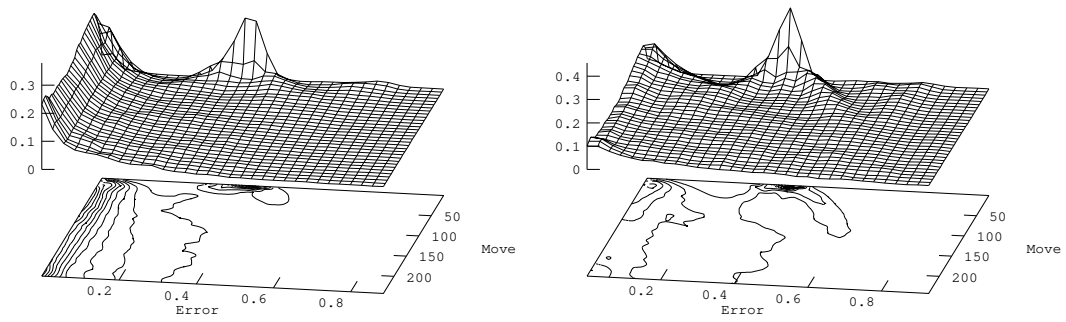

Figure 5.2: NRM plots for 3x3 and radius 1 diamond windows: there is something very wrong at the beginning of the game. The effect is even more pronounced with the smaller window.

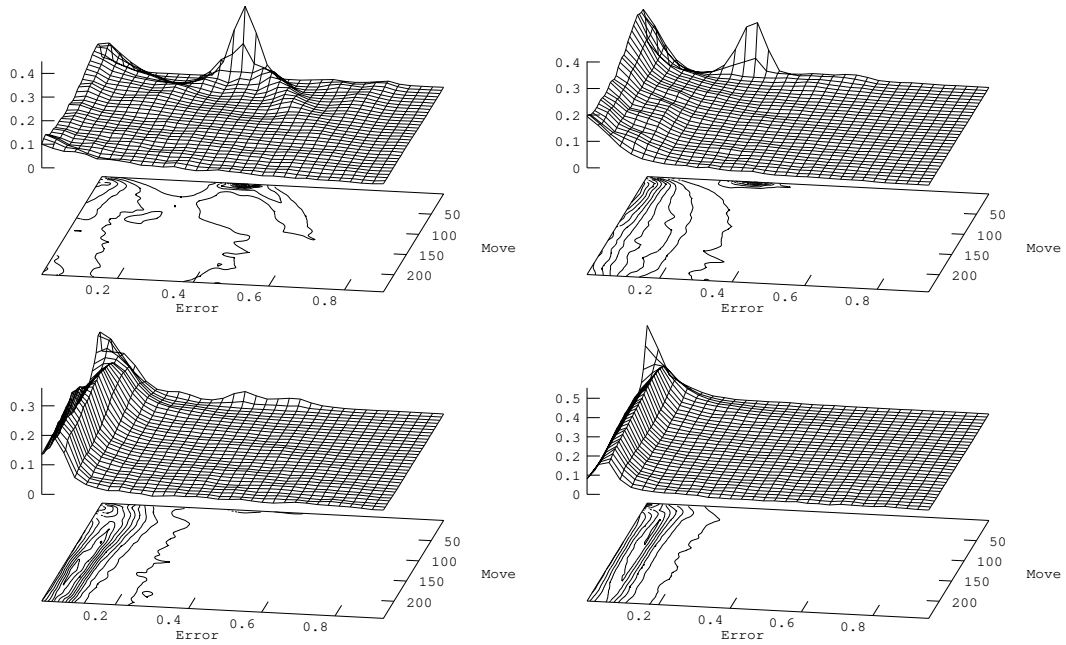

Figure 5.3: Comparison of NRM plots for diamond windows of radius 1, 2, 3 and 4 (left-to-right, top-to-bottom). The lack of discrimination near the beginning of the game nearly disappears as the window reaches to the edges for most joseki. Generated with the hashing technique of section 5.3, table size 2000003.

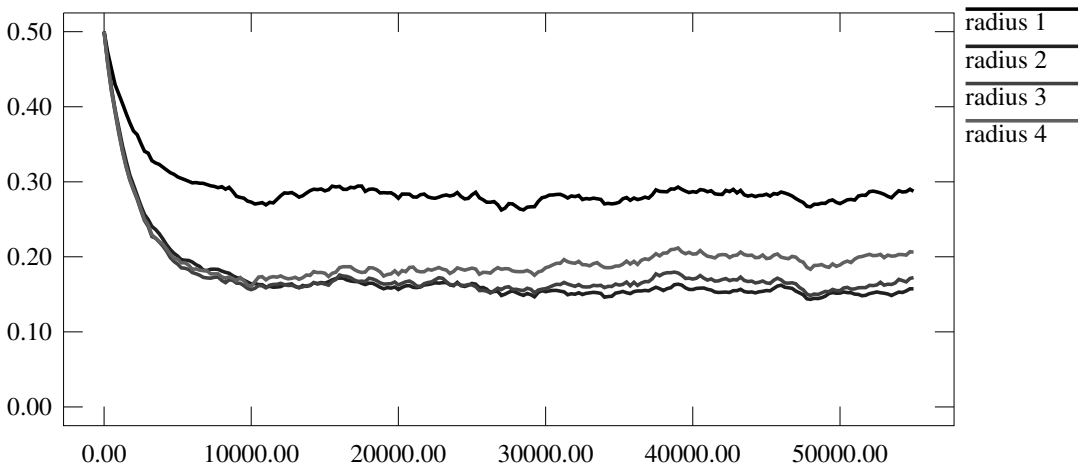

Figure 5.4: Study graph for plots in previous figure.

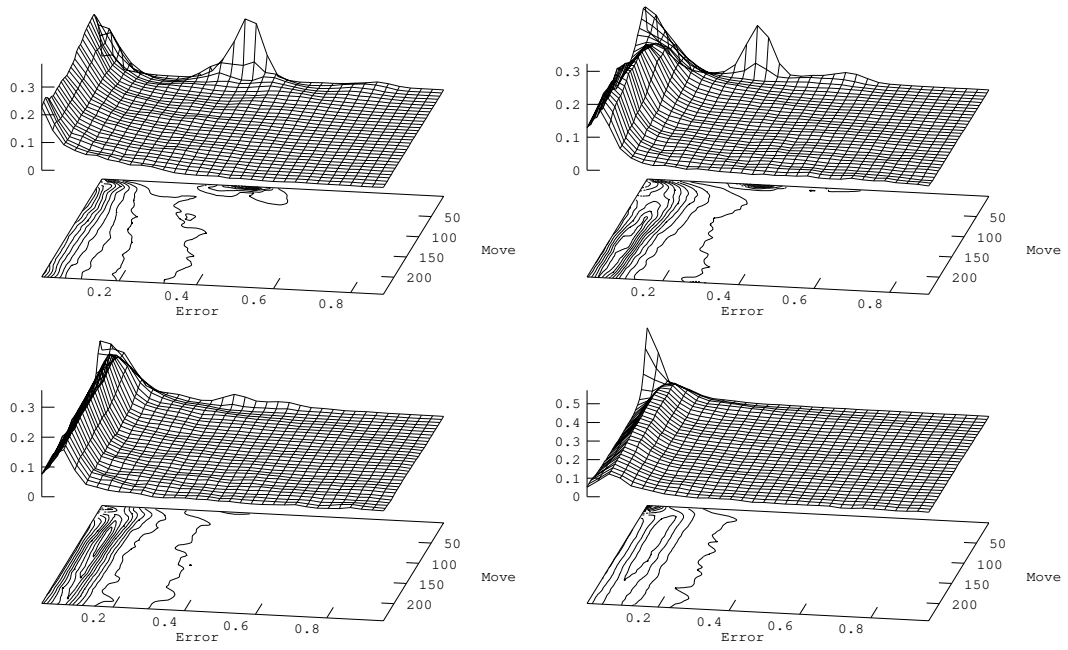

Figure 5.5: Comparison of NRM plots for square windows of size 3×3, 5×5, 7×7 and 9×9. Again, the lack of discrimination near the beginning of the game nearly disappears as the window reaches to the edges for most joseki. Generated with the hashing technique of section 5.3, table size 2000003.

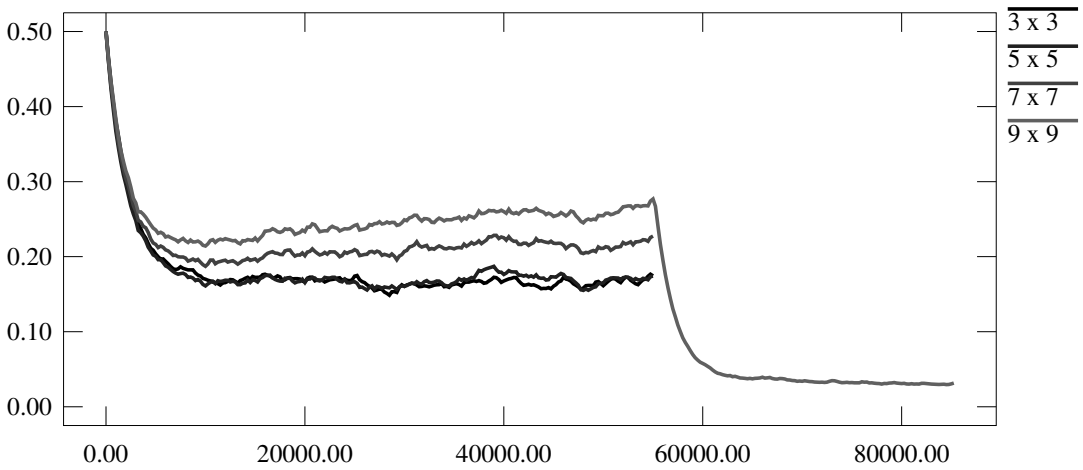

Figure 5.6: Study plots of previous figure. The 9×9 window is shown on a second pass through the database.

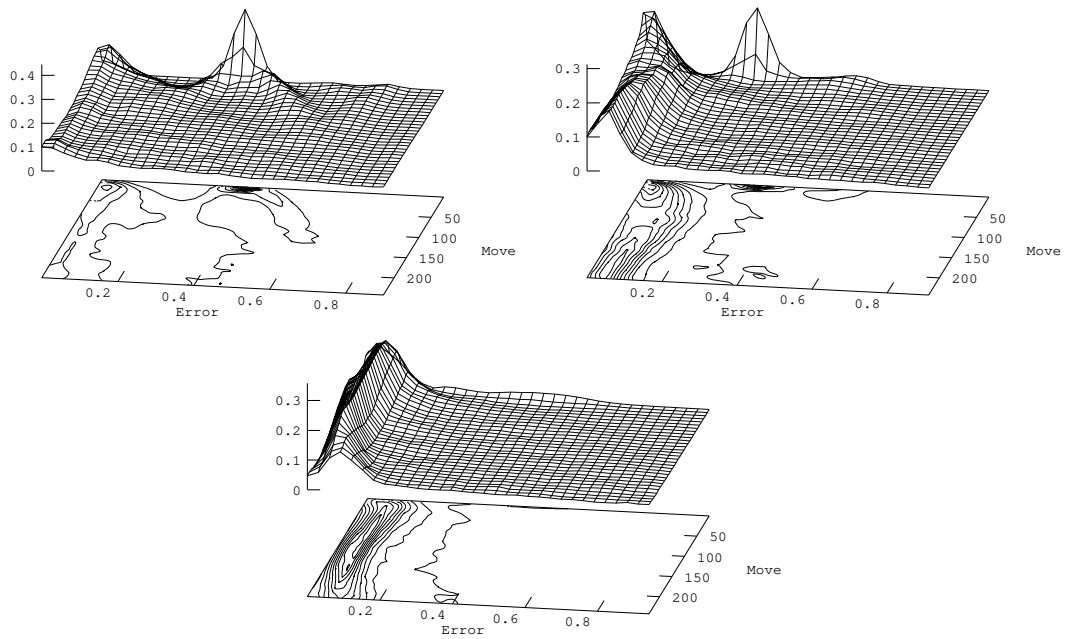

Figure 5.7: Comparison of NRM plots for graph based windows of radius 1, 2 and 3.

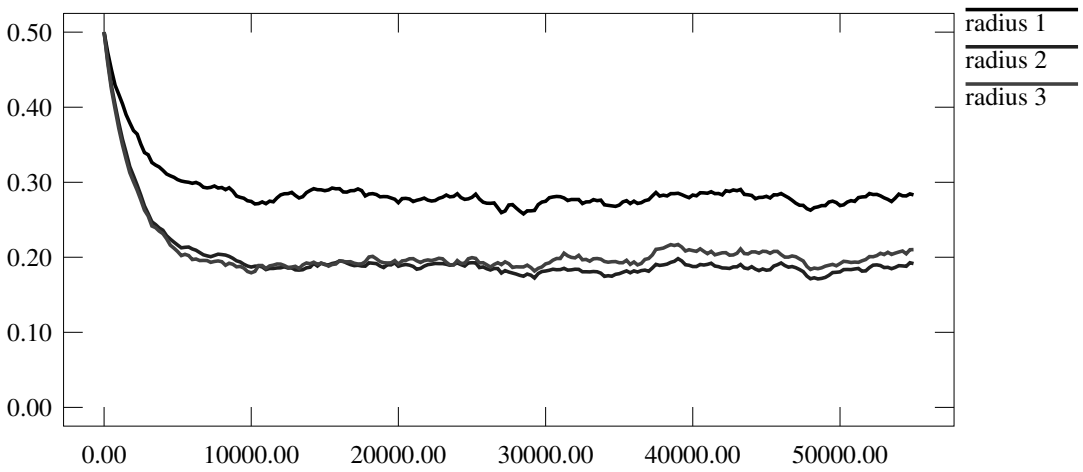

Figure 5.8: Study plot for graph-based windows of radius 1, 2, and 3. Generated with the hashing technique of section 5.3.

approach of fixing the classification function and varying values associated with each class; traditional classification techniques modify the classification function C on the fly (usually the desired classifications are known in advance.) Having C change negates the derivation of the gradient, which assumes fixed variables in order to optimize V. A fixed C is conceptually simpler, hence easier to program. Since we have some notion about what is important in classification, such as positional locality, we can hope that we can manually generate decent window extraction functions.

Chapter 1 focused on "generalization bias" as an important way of looking at learning problems. The choice of a particular classification function C chooses a particular bias; the optimization of V merely allows the learning to take place. When a fixed window of positions which are close to the position of interest is chosen, we are assuming that the most important generalizations will be made in positionally local ways; this is a significant bias. Certainly, it is possible to construct situations for which this is not true; see figure 2.2 at $\hat{\text{E}}$, where the evaluation of play is highly dependent on a situation far away on the board. The assumption of locality is a very dangerous one.

We are interested in a generalization bias similar to good go players. In a sense, by varying parameters such as window size and window shape, running the optimization program and plotting the results, we are trying to do a meta-optimization: which methods has the most appropriate bias of generalization, that is, which one most effectively emulates the human players in the game database?

Each of these meta-experiments requires too much effort (often, days of human and computer time) to automate in the same way the learning of appropriate V for a given C can be automated, at least on current, non-parallel hardware.

5.3 Hashing

There is a fundamental problem with maintaining a map of each pattern observed in a window: the number of observed patterns generally grows as an exponential function of the number of positions in the window. This is a very bad thing. It limits the size of the move database, and slows learning down: as the needed memory increases, the lack of local access slows access time (for example, paging starts to set in). I had to carefully tune things to the size of physical memory in order to avoid near zero CPU use as the machine thrashed itself into oblivion.

How can this be overcome? One way is to improve on the generalization so that the number of classes is smaller; for example, by reducing the window size. However, reducing the window size is exactly what we do not want to do, because it will overgeneralize and lump together any examples which happen to be similar near the considered move.

It is always possible to spend the time and develop a more sophisticated classification function by intuition and hard work. This is akin to how existing programs have been constructed, where the programmer is also the expert of the domain, except in this case the correct numerical estimates can also be generated automatically. I did *not* elect to pursue this method; I am not a go expert, nor have I had the decades of experience that others have had working on representational issues in go.

Another way to avoid this is to use a technique that doesn't require as much space for storing patterns and their values, possibly at the expense of precision. One way is to use a hashing strategy similar to that found in existential dictionaries (which can quickly determine set membership without storing the actual set members, but which are occasionally wrong), where precision is sacrificed for speed and small size[23].

Figure 5.9 illustrates the hashing technique. Instead of creating a new value to be adjusted each time a new pattern is discovered, a fixed number of values is allocated once. The number of values is choosen as large as possible; ideally, just large enough to reside in physical memory without paging. Each class is hashed to one of these values in a reproducible way. Consider what happens when the hash table is considerably larger than the number of classes. The probability of collisions between two classes is remote, so the overall effect of using hashing instead of a simple map between classes and values will be invisible.

As the number of classes increases, there will be more and more collisions. Each collision means that a value has to serve another class, so many values will be used for multiple classes. If these classes are both desirable, than the collision will be favorable; the combination of the two classes into one is a useful generalization. If they are disparate, then the collision will be harmful, and the value will have to seek an intermediate value. If two classes are served by the same value, but one of them occurs much less frequently than the other, it will have propotionately less effect on the gradient, and the value will not be swayed so far.

If the hash table size is too small, then collisions between disparate classes will become frequent, and we can expect to see a degradation of ability to discriminate. Figure 5.10 shows how this occurs. The hash table size is progressively increased; the rate of collisions drops, and learning is more effective.

Note that this approach is relatively robust. It doesn't require storing the access patterns

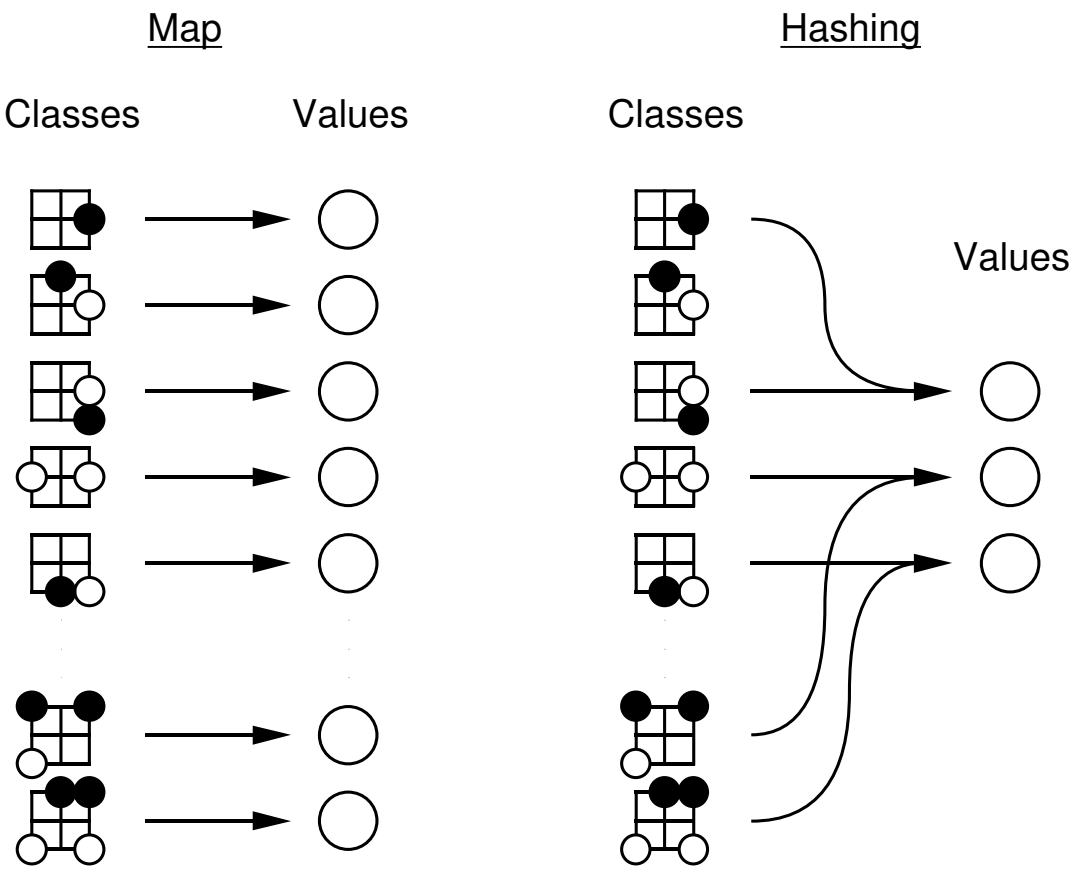

Figure 5.9: Comparison of hashing and map strategies. With the hashing technique, there may be collisions between classes.

like a map does, so better use may potentially be made of memory. As the the number of classes increases, there is a graceful degradation of performance.

Unfortunately, the size of available memory on workstations is not enough to prevent a general failure to cope with number of patterns created by large windows. This can be seen by comparing the 7×7 and 9×9 windows in figure 5.6; the exponential explosion will eventually overwhelm any reasonable hash table. In addition, consider that when the size of the database is small (and 257 games *is* small for this sort of learning), the number of observed instances of each pattern becomes vanishes; noise overwhelms the signal.

A word of warning: I found, naturally enough, that the collision rate was extremely sensitive to the particular hash function. I experimented with a variety of functions before settling on one which seemed to have a minimal collision rate. In particular, I extract patterns by first transforming them to an intermediate string representation; this is then hashed. Because this string often would differ by only a single character for different classifications, it was necessary to choose the hash function carefully.

One attempt was made to circumvent the collision effect, by adding a *quality* component

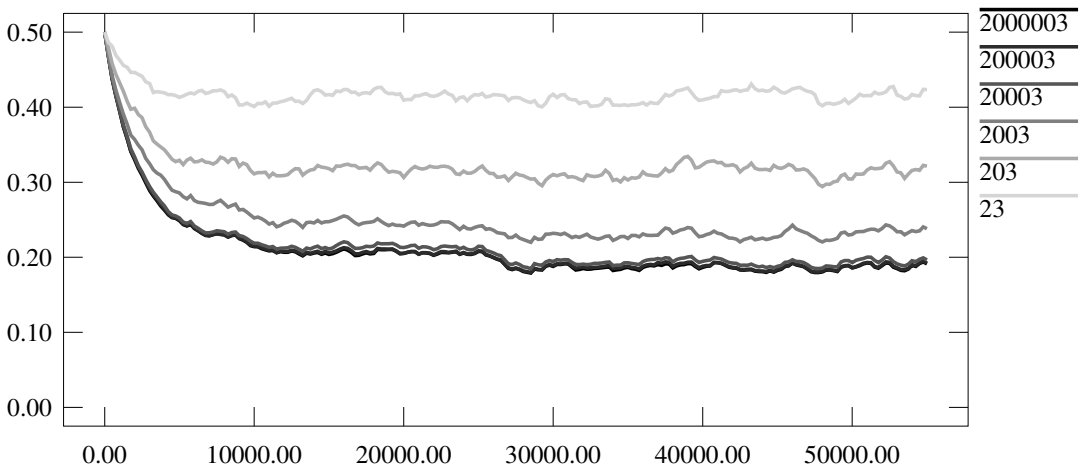

Figure 5.10: A study plot, showing the effect of collisions on hash table method effectiveness. As hash table size increases, probability of collisions between patterns decreases, so the table is a better discriminator between patterns, approaching the effectiveness of the complete map for a large enough hash table.

to each hash entry. The table was hashed into five times using double hashing, and the ultimate evaluation, rather than simply being the value found in the table, was:

$$\frac{\sum_h v_h q_h}{\sum_h q_h}$$

where v_h is the value found in the table for hash h and q_h is the quality component located at that entry. All values are initialized to 0.0, as in the previous technique, but the quality component is initialized to 1.0. The evaluation shown above is a weighted average of multiple values; the quality component is the weight that a value should have when combining it with other values. Initialized to 1.0, this amounts to averaging. Over time, these values should go to zero for values with many damaging collisions, and increase for values with great correlation.

The gradients for this method can be found by chaining from the previous method; for an evaluation e and hash entry i,

$$\frac{de}{dv_i} = \frac{q_i}{\sum_i q_i}$$

and

$$\frac{de}{dq_i} = \frac{v_i}{\sum_i q_i} - \frac{\sum_i v_i q_i}{(\sum_i q_i)^2} = \frac{v_i - e}{\sum_i q_i}$$

and this gradient can be followed as before. This didn't have much effect: addition of a quality component resulted in negligible improvement.

Figure 5.5 shows the very large window (9×9 square) on encountering data for a second time; it has managed to nearly memorize the data, in spite of collisions that must occur on the enormous number of patterns observed in the window. For this reason and the experience with addition of a quality component it would appear that collisions are not the

primary mechanism which is losing discriminatory ability; one would expect the addition of a quality component to have has a noticable constructive effect if collisions were a problem.

5.4 Pattern cache

Another approach to decreasing memory use is to selectively forget patterns. The idea is to maintain a working set of patterns which are effective and toss out patterns which aren't helping to lower the error much. How should patterns to be forgotten be choosen? Here are a few ways:

- The old standby, always replacing the least recently used pattern, akin to standard methods used for paging memory (LRU). This has the advantage of being easy to implement, but has little to recommend it from a go perspective - we wouldn't want a pattern of great utility to be forgotten just because an instance hadn't been encountered recently. On the positive side, patterns of high utility are likely to be so just because they *are* frequently encountered.

- Replacing patterns with values close to the default value used when a pattern is created. The motivation for this is that if we delete patterns that have a value close to the default value choosen when a pattern is introduced into the mapping, then if a future evaluation finds this pattern the value used won't be too different from what it is at the time we are considering replacing it. In my code, the default value was zero (that is, all patterns are "initialized" to zero, the same value as a pass), so this would mean preferentially replacing patterns with low values over high ones.

- Replacing patterns that recently have contributed in a negative manner to the overall error. The program could keep track of difference in effect each pattern would have had on the error if it's value were the default value instead of whatever it happens to be at the moment of each evaluation. Some kind of running average of this effect would allow replacing those patterns that are not contributing to the "greater good".

I tried implementing the second technique. One way this could have been done would be with a priority queue, so that the pattern with the lowest value is always available on demand. To avoid the space taken by a splay tree or other fast data structure, I implemented a simple flushing technique, similar to that used to approximate LRU in paging systems. A maximum and minimum cache size was given by the user. Once per evaluation, the cache is checked to see if it exceeds the maximum cache size. If it does, then all patterns are discarded which are less than a certain value, determined by estimating the number of cache entries needed to reduce the size to the minimum, based on the assumption of a uniform value distribution between the max and min cache values.

This causes the cache size to slowly grow until it reaches the maximal value, at which time enough patterns are deleted to bring it near the minimum size. The choice of a cropping value based on an assumed uniform distribution allows deletion only of patterns which would likely be deleted by a "pure" priority technique.

This did not work well in comparison to hashing. This can explained by the nature of collision between patterns. When a flush occurs in a pattern cache and patterns are

wiped out, this is relatively catastrophic. In comparison, when collisions occur in the hash table, patterns are sharing a value; if their optimal separate values are similar, this may be constructive, and if one occurs frequently with respect to another, the value will tend towards some kind of weighted average of the individual values. In short, the hashing technique degrades gracefully; the pattern cache doesn't.

5.5 Improvements

I tried two techniques to futher improve the performance, *conditional window expansion* and *liberty encoding*.

When a fixed window is used, it suffers from overgeneralization for sparse positions (those with few nearby stones). This was seen in the poor performance of small fixed windows at the beginning of the game. Conditional window expansion uses a fixed shape window, such as a diamond, but adjusts the size to take advantage of information such as the relative position of the edge when pieces are far apart. This was implemented by modifying the parameters given to the diamond window extractor; instead of extracting a window whose radius is fixed and specified by a parameter, the new parameters are the *min* and *max* window sizes and the minimum number of stones which must be seen in the window. In this scheme, extraction proceeds by starting with a window of radius *min*. This window is incrementally increased until either the minimum number of stones criterion is met or the maximum window size is reached.

Another experiment was to add liberty information to the window about groups which are only partly inside the window; whether a group has more liberties outside of the observation window is very important for tactical relationships. For this, an additional parameter was added; the maximum of the parameter and the liberty count of a stone was encoded, so that two otherwise identical positions that differ only in the number of liberties of a group will not be catagorized differently past some set limit of liberties. The rational for this is that once a group is alive, it is alive, and the difference between having five and six liberties isn't likely to be important; however, having or not having a single liberty is extremely important—it is the definition of the life or death of a group.

Figure 5.11 shows that the effect of allowing conditional expansion is dramatic. The point of conditional expansion is to avoid the exponential explosion of classifications in crowded condition, and still achieve the benefits of a large window at the beginning of the game. Addition of liberty information was a slight improvement, but not past discriminating between groups with only a single liberty and those with two or more.

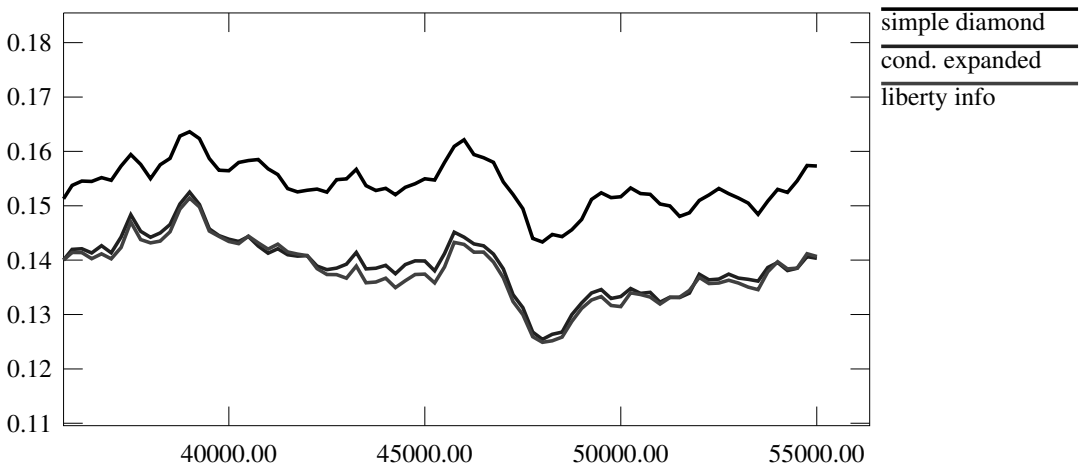

Figure 5.11: Study plot for end of training data, for: a simple diamond of radius two; a diamond conditionally allowed to expand to a maximum radius of five until a stone is seen; and with the addition of identification of groups with a single liberty.

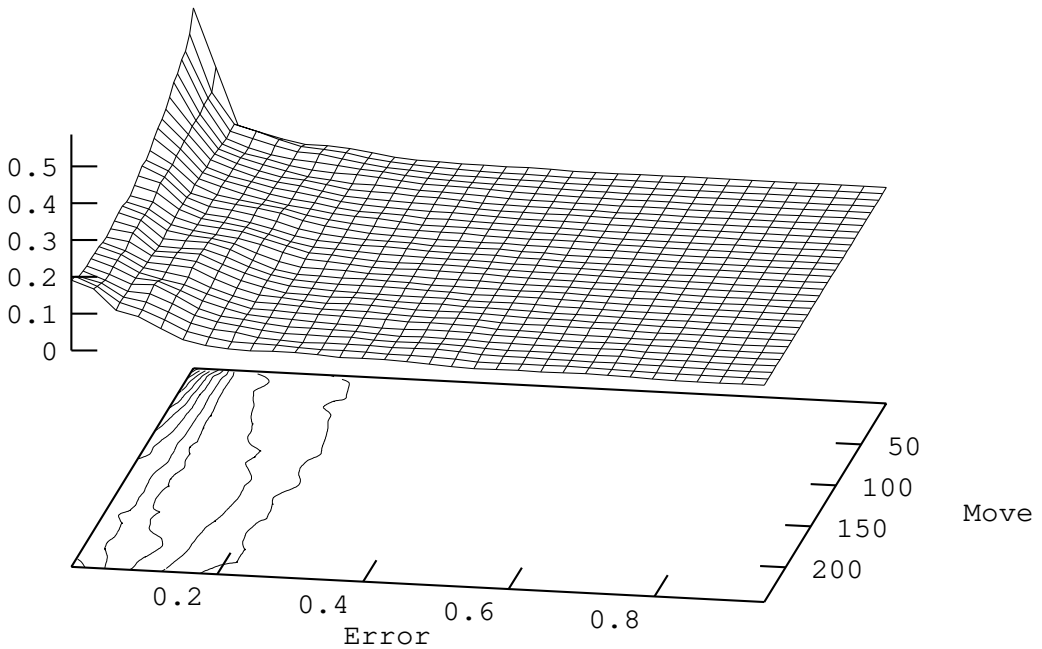

Figure 5.12: NRM plot for the best found classification: conditionally expanded diamond window with liberty information added.

Chapter 6

Example Game

6.1 Evaluation on a master game

As mentioned in section 4.3, it is difficult to correlate the simple performance of a heuristic on training data with real world skill. This section examines pattern preference using a diamond shaped window of minimum diameter 2, conditionally expanded until a stone is seen to a maximum diameter of 5 with single liberty group identification (the classification from the previous section), implemented using a hash table with 2000003 entries (see section 5.3). Here we examine this classification's recommendations on a game which was not used for training.

This illustration game[1] was played between a 9 dan and a 4 dan. Each player had three hours total in which to move; in comparison, the program required slightly under five minutes on a DecStation 3100 to produce these recommendations, including the time used for "learning" from this game for use in future evaluations.

For each move an error is given, which is the error associated with that move, as defined in section 4.1. Moves are specified in a standard notation, where the letter represents the column ('i' is skipped to avoid confusion) and the number the row; for example, 'd4' is the point at the intersection of the lines fourth from the left and bottom. Figures are given for each commented move. Numbers on the boards in the figures represent the rank of the moves which the program evaluated as being as good as, or preferable to, the move the master played; the actual move played is shown in grey. The fewer numbers on the board, the fewer moves were ranked inappropriately higher than the master's move, so the lower the error associated with that move is. The game:

1 (0.011) r16
2 (0.021) d4

The errors of moves at the very beginning of the game suffer from the many symmetric positions. The program identifies the 4–3 point (a move four lines from one edge and three from another) as most desirable. The preference rating between the 4–4 point and the 4–3 is very close; the 4–3 was played in more corner situations in the database.

[1] From the third Fujitsu Cup, June 2, 1990; played by Kobayashi Hoichi (white) and Lee Chang-ho (black). A complete description and expert annotation can by found in *Go World*, No. 61, pages 8–12.

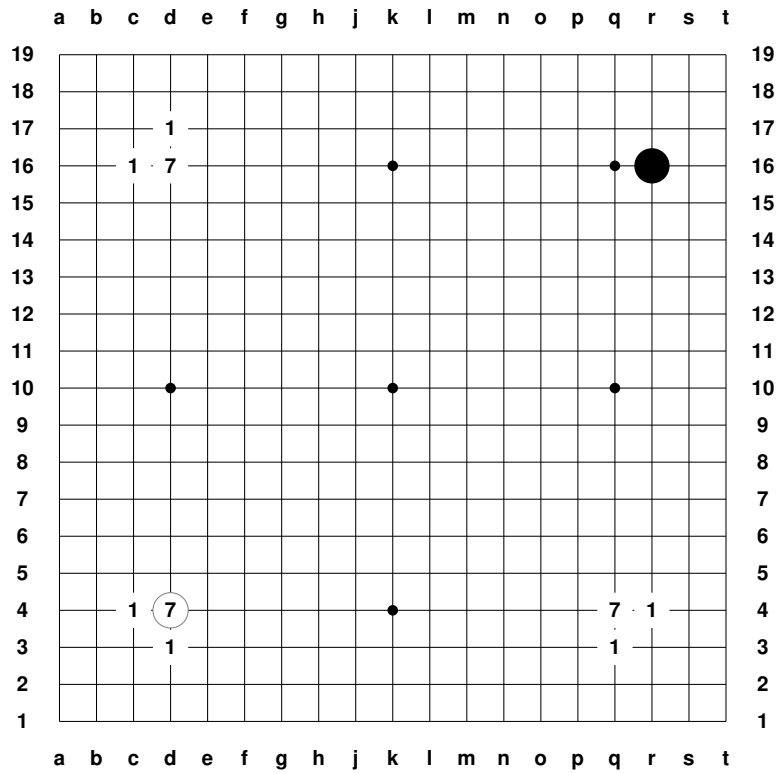

Figure 6.1: Move 2

3 (0.011) q3
4 (0.006) d17
5 (0.095) p17

Two problems with the classification show up here. The rank 21 moves along the middle of the left, bottom, and right sides are move than five positions away from another stone; because a maximum radius of five was imposed on the window, these cannot have different classifications so the relative distance to the corner stones has no effect on the evaluations.

6 (0.223) r6

The first really bad evaluation, in terms of the error function; the master's move ranked 79th out of 356 possible moves, and so would probably not have been considered if this heuristic were being used as a move suggestor. Also note that the rank 4 moves are not discriminated by distance to the other edge; the window stops expanding at radius two because of the presence of the black stone, so the corner move is ranked the same as the outside move.

This type of problem with joseki goes away with large windows which more or less perfectly memorize positions, but this loses the ability to generalize; most go programs have a special joseki library to deal with this aspect of the game.

7 (0.069) q5
8 (0.083) s4
9 (0.242) r8

Another unexpected move, from the program's point of view; it would have extended

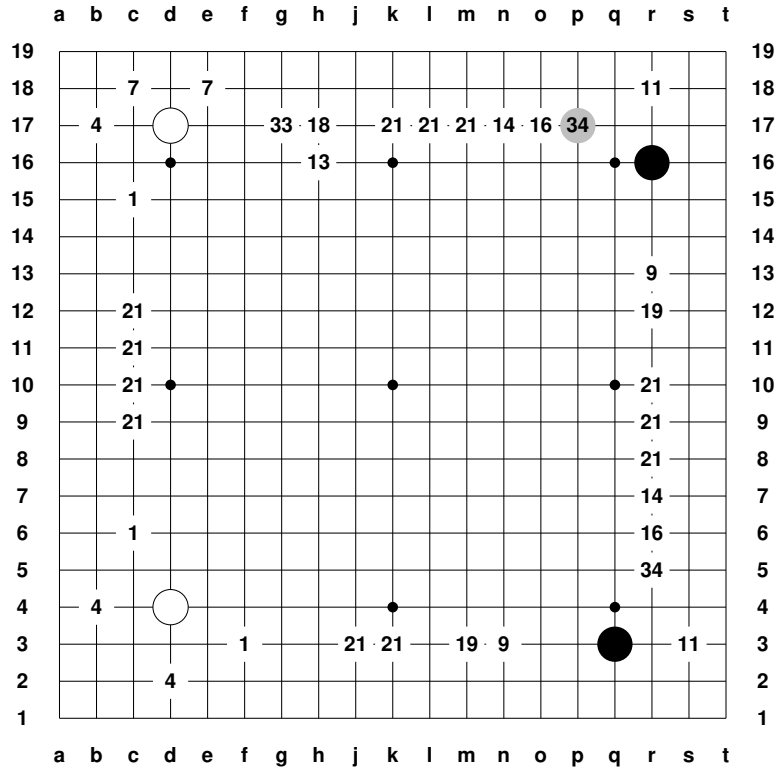

Figure 6.2: Move 5

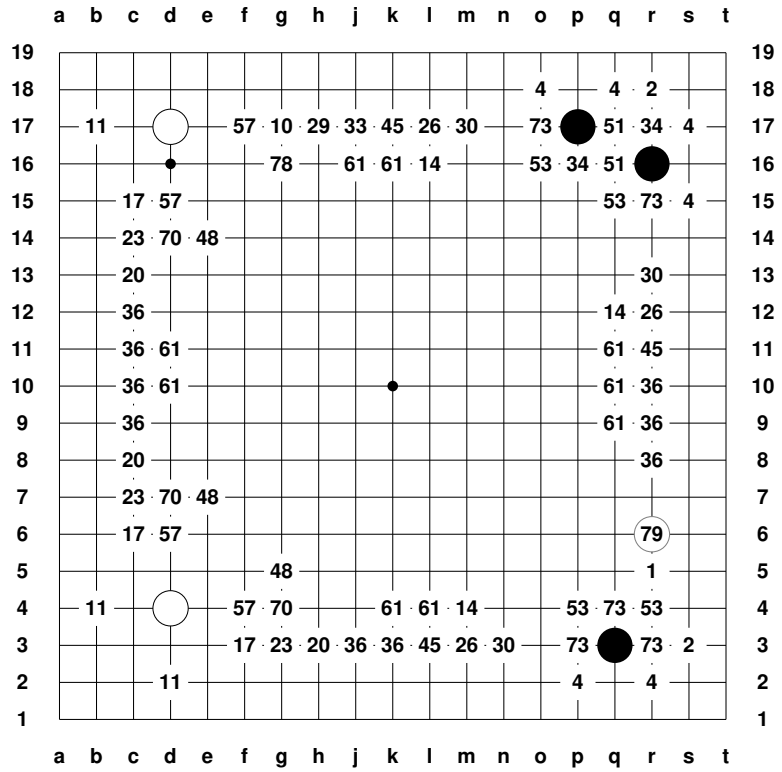

Figure 6.3: Move 6

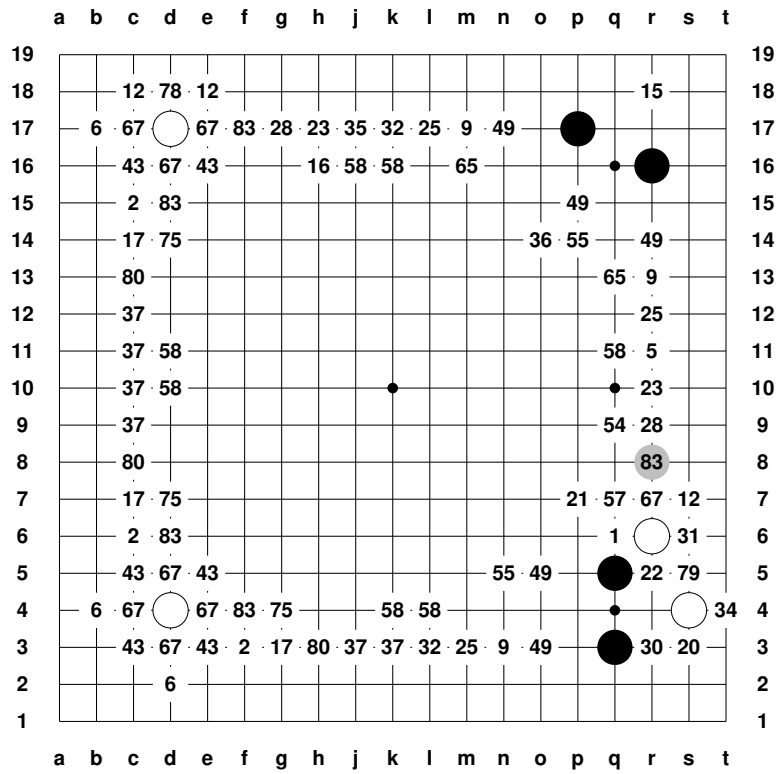

Figure 6.4: Move 9

at 1. Once again, the radius two window can "see' the black stone at q5 and the white at r6, but it has no information about the other stones or its relationship to the corner.

10 (0.126) r3

11 (0.004) r5

The first of four nearly perfect judgements. Each of these is a simple local judgement which pattern matching worked well for, such as the edge extensions of 12 and 14.

12 (0.001) s5

13 (0.001) q6

14 (0.004) q2

15 (0.019) p3

Again, the evaluation at 1 could not see the white stone at r3, so it didn't know that it made 1 premature by threatening q3.

16 (0.004) p2

17 (0.173) n3

18 (0.061) d15

19 (0.203) j16

20 (0.025) q12

Because there are no stones around, the window for move 19 extends to radius five, enough to "understand" the relationship to the edge, but not to the rest of the board. Move 20, by contrast, is neatly a distance of five stones from the two black stones and close enough to the edge to see it.

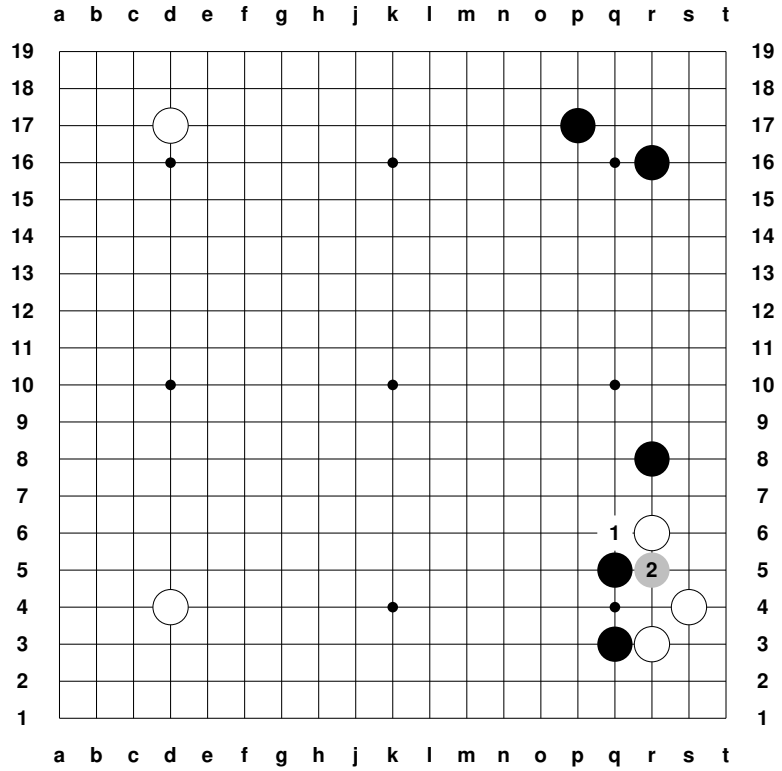

Figure 6.5: Move 11

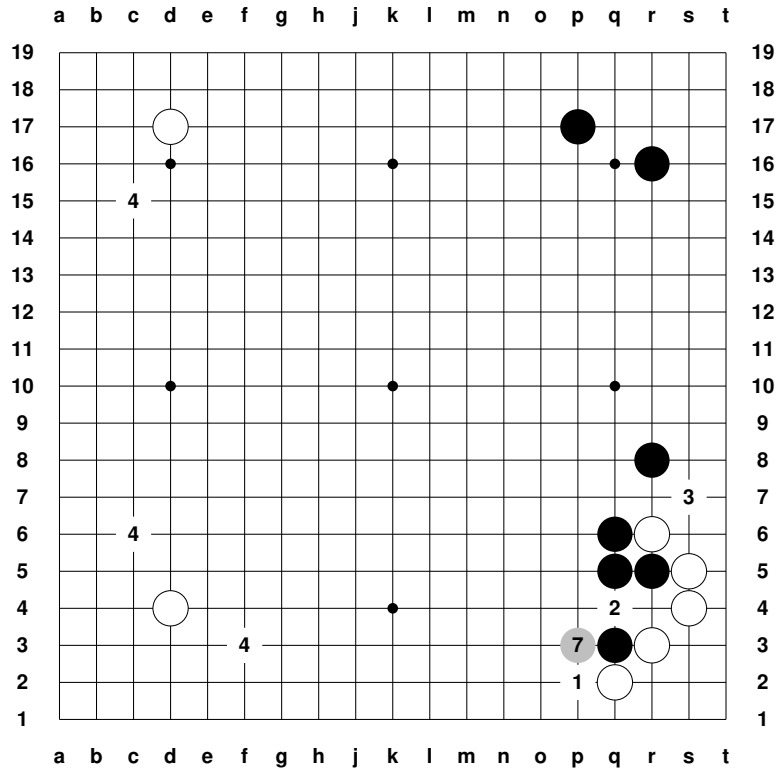

Figure 6.6: Move 15

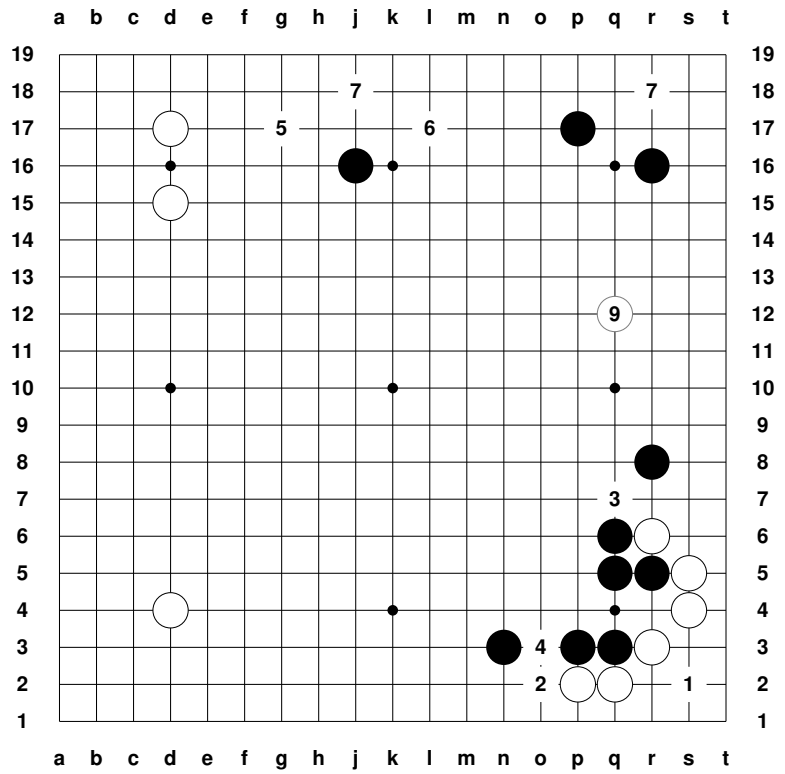

Figure 6.7: Move 20

The move it wants to play (at 1) will not be necessary until move 119, when it is forced by a black play at r2. However, the window cannot see beyond the three white stone which are at radius two.

21 (0.064) r14

22 (0.028) o12

All the moves rated 7 in move 21 share the same classification: a single friendly stone at the tip of the diamond window. This is actually the right move for 22.

23 (0.146) m12

24 (0.434) p9

White later said that attaching at q14 was a much better move, which would make the rank 33rd instead of 141th.

25 (0.072) n9

26 (0.136) r9

27 (0.201) p13

28 (0.007) p12

The rank 1 move at p8 remained the program's favorite for both white and black from 28 through 32, and from then on cropping up repeatedly, interrupted only by obvious contact fights. It sees the white stone above, but not below.

(Further moves without comments can be found in appendix C.)

43 (0.002) d18

This kind of move fits all the criteria needed for a successful classification: there is

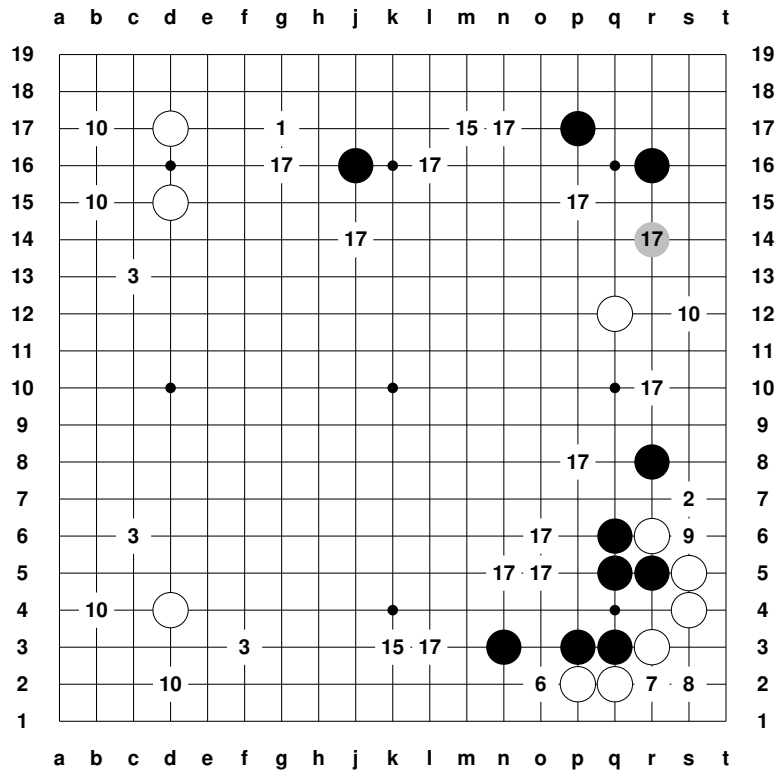

Figure 6.8: Move 21

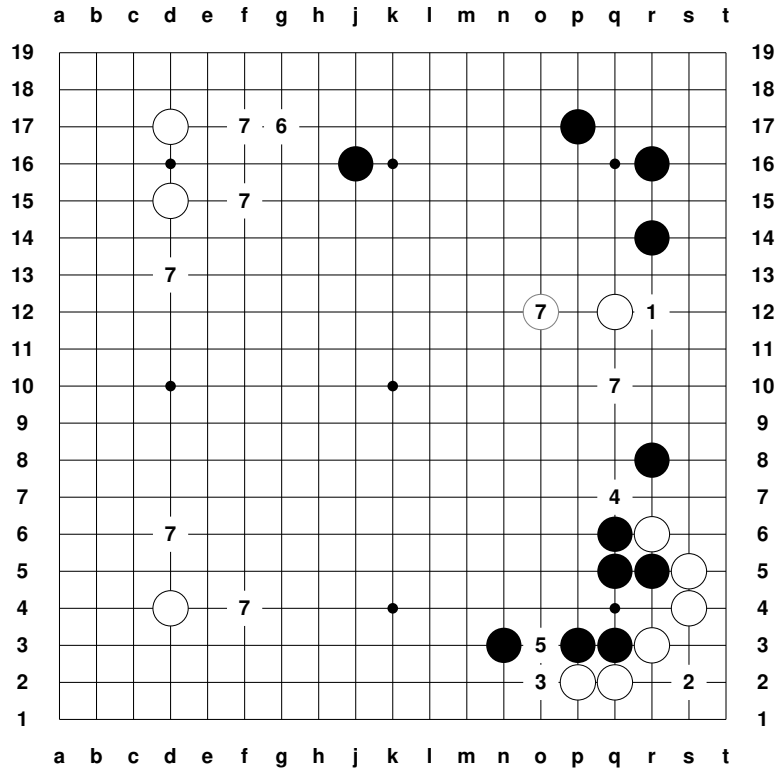

Figure 6.9: Move 22

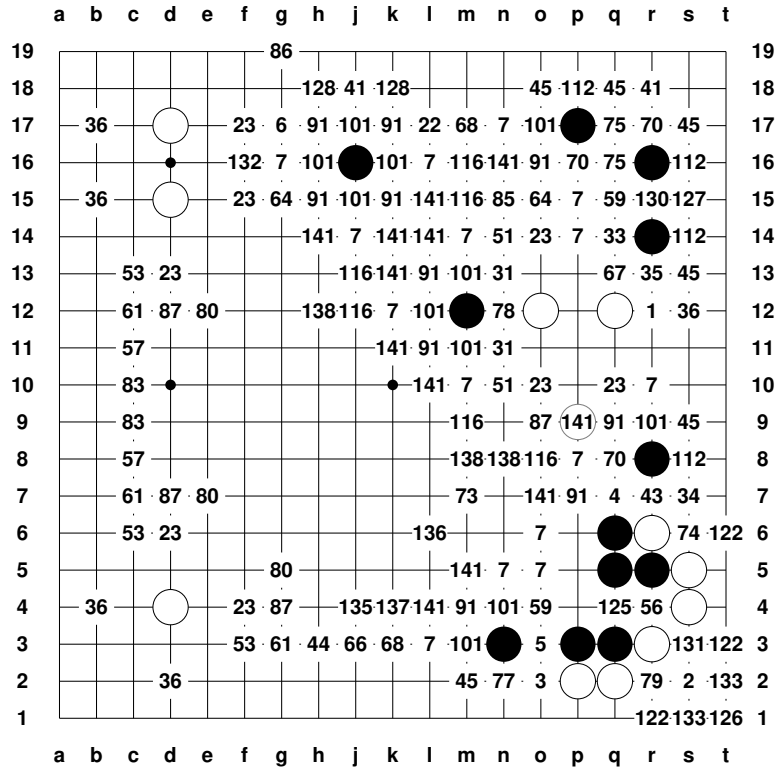

Figure 6.10: Move 24

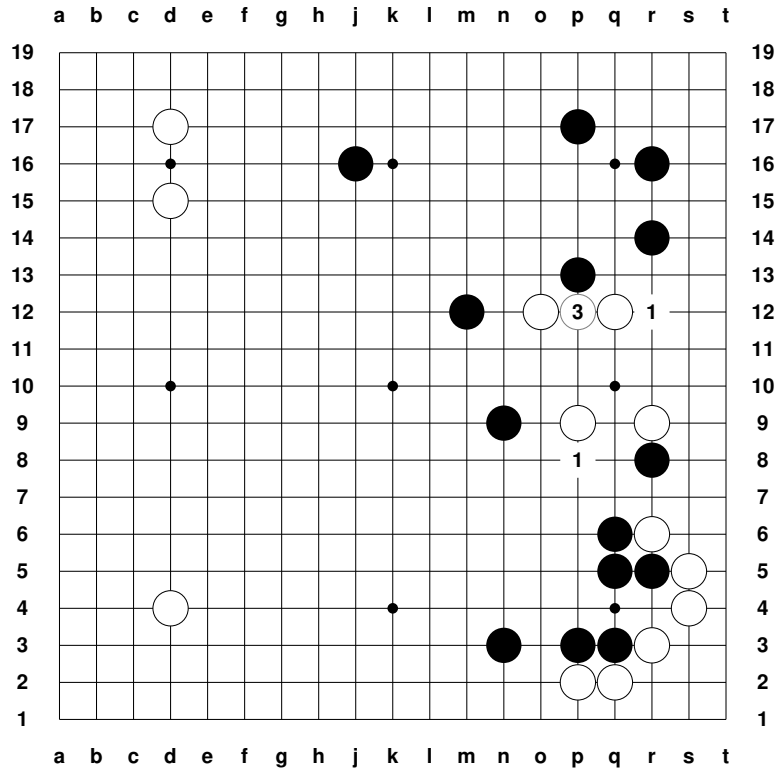

Figure 6.11: Move 28

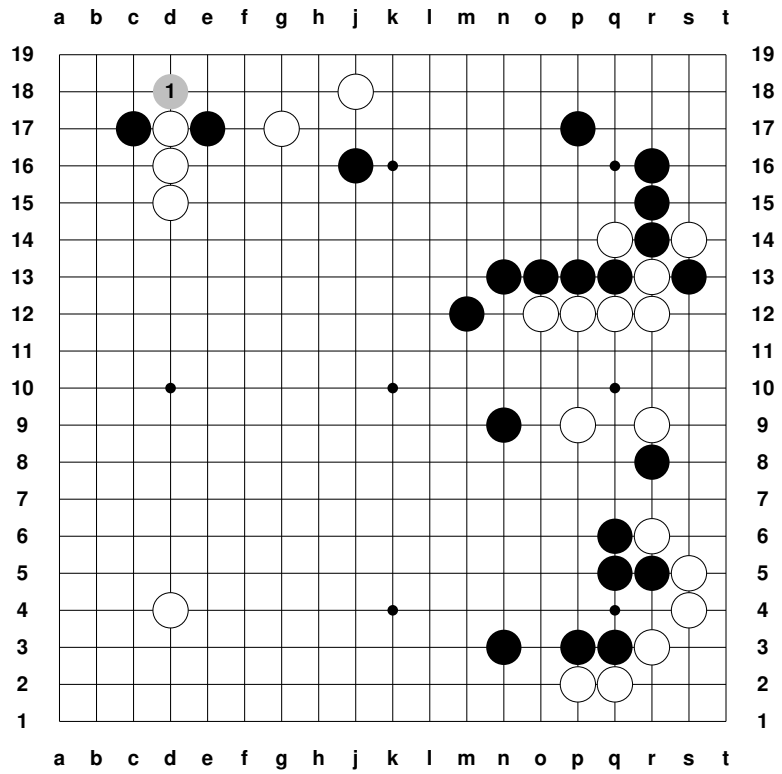

Figure 6.12: Move 43

a definite purpose (connecting), it is a forced move, and everything needed to see this is within the window. Of course, if there were another similar move somewhere else on the board the program would be unable to make any judgement between them based on point value.

44 (0.301) c16
45 (0.083) b17
46 (0.314) c18
47 (0.552) b18

This classification for this move probably did not occur in the database; compare the fleeing moves at 1 and 2 which the program would make. It does properly deal with the immediately following "obvious" follow-ups.

79 (0.757) h2

The evaluation stems from a number of things. The window can see the white stone, but no further, and certainly has no clue about the tight relationship of the black and white groups. As far as the window is concerned, there is nothing on the board aside from the white stone, so there is no point to this move.

92 (0.024) c13

The rated moves shown are a good sample set; using local patterns the program knows to 1 extend, 2 push, 3 capture, 4 connect, and 6 cut. 5 is the old delusion repeatedly seen from move 28 on; the actual move at 7 is made possible by the white stone at d15, which is out of the window.

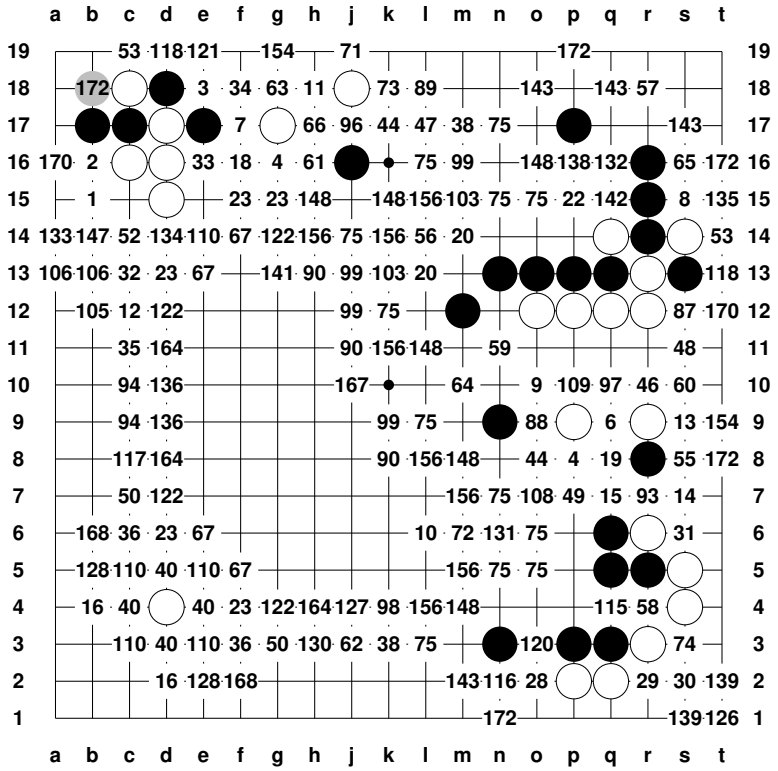

Figure 6.13: Move 47

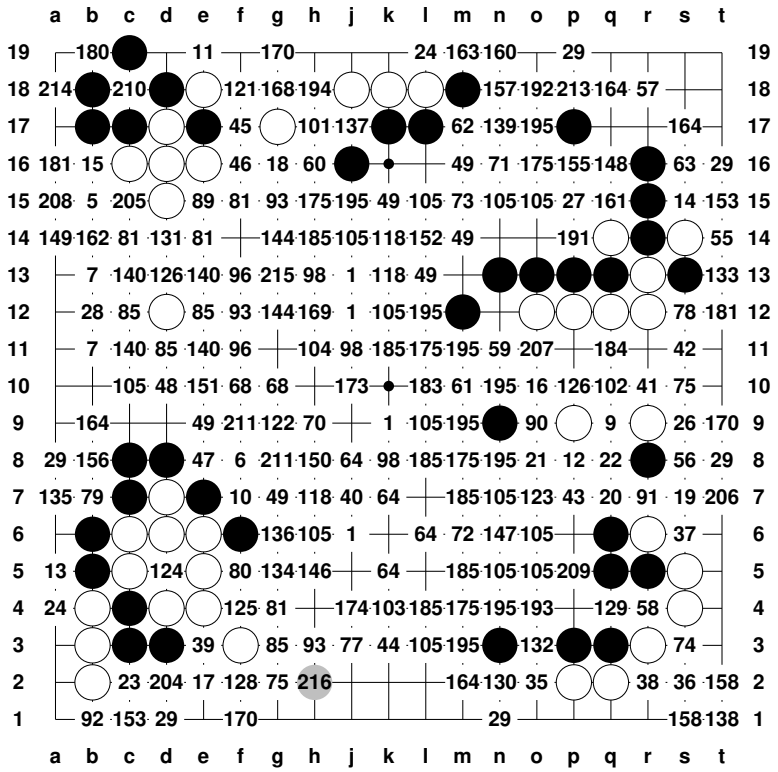

Figure 6.14: Move 79

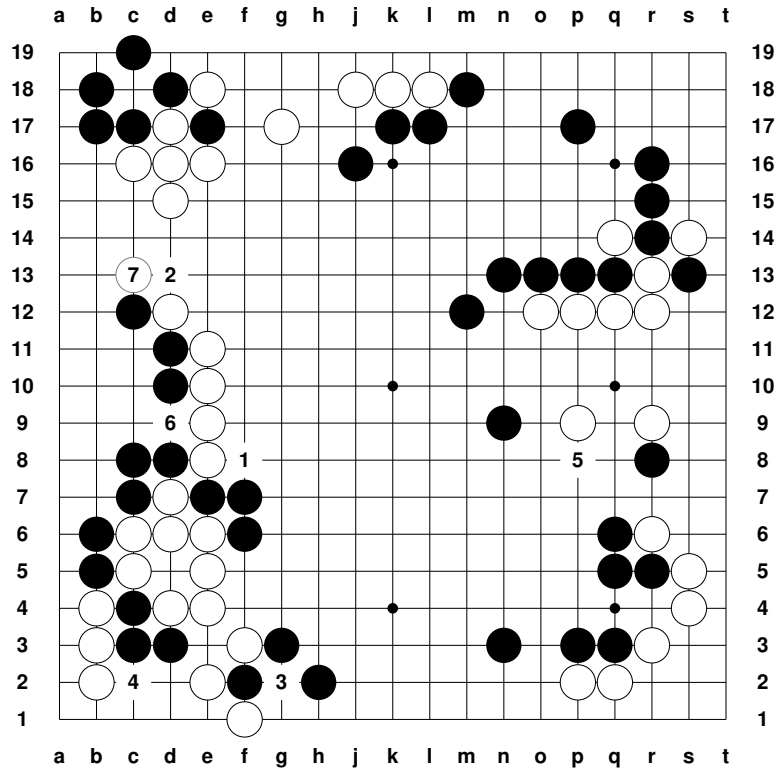

Figure 6.15: Move 92

138 (0.064) l4

The ones here are an interesting phenomena; each of these is a point-wise small move, in which no other stones are in the window for classification.

145 (0.034) l7

White commented that a play at 5 would have probably made the difference and allowed black to win by two and a half points; again it is amusing that this move has a better ranking by the program, although the other poor moves like 3 at j15 would do it in in a real game. Also, the worth of these moves is being decided by careful count of the points on the board, which is miles above the reasoning used by pattern preference.

230 (0.141) e13

White loses by half a point. Notice that most of the considered moves remaining are dame (neutral point) fills and interior space fills, which are necessary under chinese rules. There are exceptions like the attack at 8, but once again harmless under chinese rules.

The actual numeric ratings for the moves ranked 12 and above are below zero, so the program would prefer to pass rather than make those moves, including the shoring up move at 19.

6.2 Discussion

The performance of the heuristic in the sample game as a move suggestor is generally sufficient, but hardly breathtaking. There are several broad reasons which can be given to

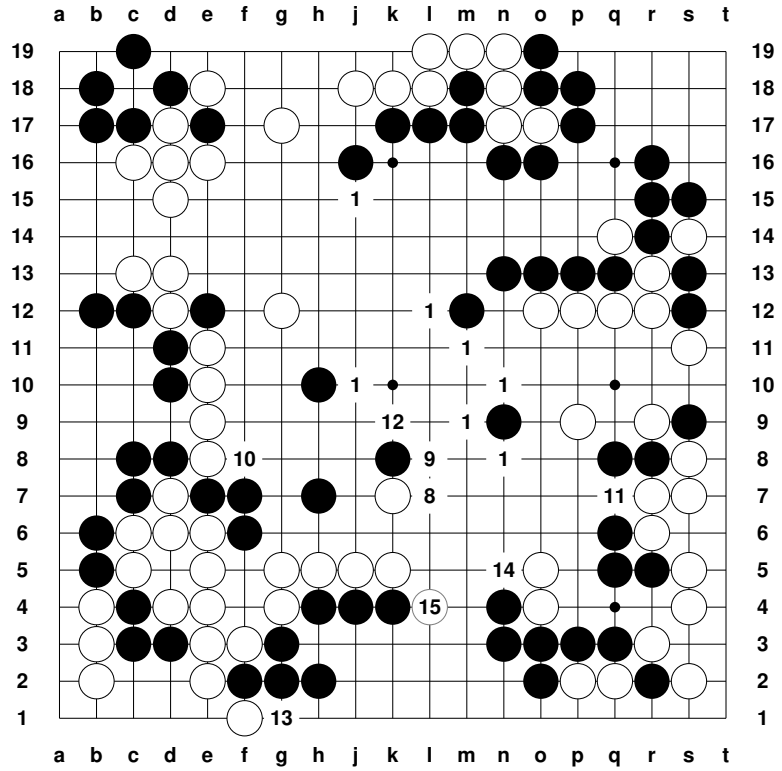

Figure 6.16: Move 138

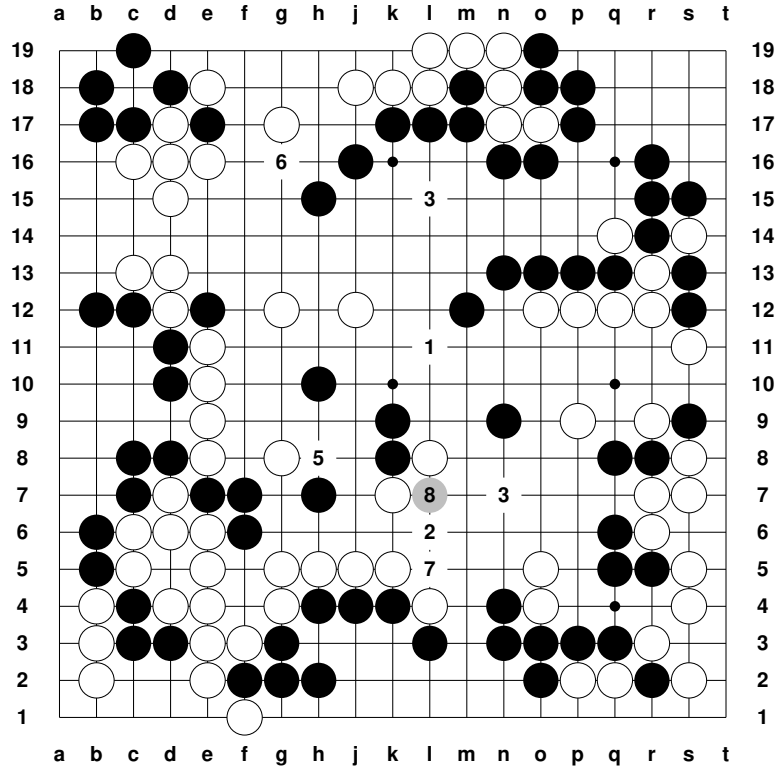

Figure 6.17: Move 145

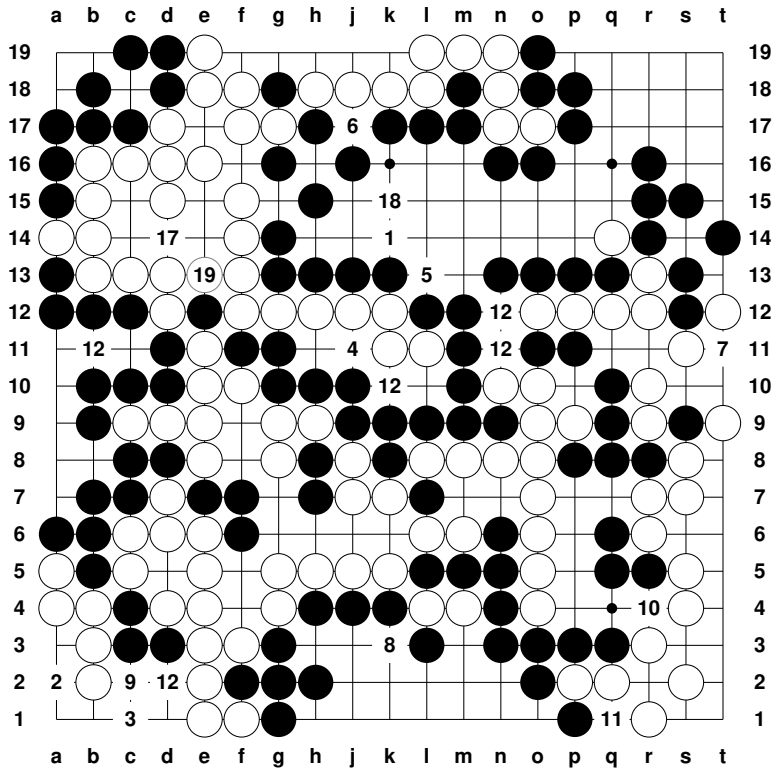

Figure 6.18: Move 230

explain the failures of the heuristic in the previous game:

- Poor choice of training data. The database is entirely composed of games played between masters; there is a stylistic difference between the games of masters and less skillful players. It is possible that this stylistic difference makes this data not useful to a learning program; for example, many of the moves that masters make are the result of sophisticated judgements and may require follow-up moves which a simple opponent may not be capable of making, negating any intrinsic worth of previous moves. In fact, experience with other programs (see section 2.3.7) tends to support the hypothesis that strategy dependent on later follow-up moves is a poor choice. Tesauro carefully built a database by hand to reinforce behavior which was appropriate; he suggests a strategy for constructing expert-like systems by iteratively modifying a database of examples, which seemed to work for backgammon. From the sample games it can be seen that the inclusion of a joseki library would remove much error due to using a small window. Alternatively, a separate pattern preference table using a large window would be able to learn the same information from sample games, when it occurs in the database. However, the existence of large joseki databases make this less attractive.

  Some of the poor results can be explained by the lack of tactical knowledge. In particular, life-and-death battles and the decision of whether or not to pursue ladders and edge extensions clearly cannot be made from local information alone. Pointwise, life-and-death deciding moves are often some of the most important in a game.

Inclusion of these specialized components could improve the performance of both the encompassing go-playing program as well as the pattern preference heuristic itself, because it would not have to train on moves with special needs, improving the quality of the data seen.

- Too little training data. The database had 55,000 moves, not particularly large for as complex a game as go. It is unlikely that this was a problem for the classification technique used because of the apparent convergence in a single pass of the database.

- Poor error function. The defined error function may not relate skill on-line during play with the above metric. For example, if a particular move which a heuristic is able to correctly rate depends for success in play on a following move which the heuristic does not rate well enough to get it played, then playing the first move is of no help, and may even hurt the player! The error function I used assumes independence of moves in this way.

  When the heuristic is being used alone to decide moves to play then its weaknesses, then perhaps the difference between a rank of one and other ranks should become more important to the error function, because the highest ranking move will need to be played. In other words, if a particular move is very important, then it may be less important to know whether there are two or three moves inappropriately ranked better than whether the best move is selected at all.

- Poor classification technique. The classifications that I used evolved in a trial and error ad-hoc way using the metric of performance on data not used to train, in other words, an estimate of the generalization bias. The classification used in the preceding game examples was chosen because it seemed to have a good ability to generalize by this metric. At the same time, it was too small to correctly identify many moves. Increasing the minimum window size generally degraded the generalization bias metric however. This is the most likely to be the major contributor to poor performance.

Chapter 7

Conclusion

7.1 Summary

Go is presently a nearly ideal domain in which to study machine learning. It is easily defined and many human experts are around, yet programs to date have not played particularly well. This can be looked at as a failure of methodology. Other simpler games, such as chess, are amenable to a combination of simple heuristics and brute force search; however, it appears that go is *not*. Some method of automating acquisition of go knowledge is called for.

In section 4.1 a simple error function was defined, the reduction of which approximates good go play. This error function has several distinguishing characteristics. It is simple to compute and relies on no particular go knowledge other than legality of moves. It is nonparametric, so it is robust and tolerant to perturbation. It is based on a database of expert play, so minimizing the error function amounts to approximating the generalization bias of the human experts whose games make up the database.

The simplicity of the error function allows diverse learning techniques to be applied. I tried variations on two themes, neither of which required traditional game-tree searching.

The method I dubbed *pattern-preference* was introduced in section 5.1. It requires the separation of the evaluation into two distinct functions, a classifier, which for each valid move on the board determines a unique class to which that move logically belongs, and a mapping between the set of classes and real numbers, which gives the actual value for a move, as required by the error function. For a fixed classification function, the task of finding a minimizing mapping can be automated; in this report, a function approximating the gradient was derived, which could be followed.

Various classifying functions were tried. Some separately classified every possible pattern observable in a fixed window; this generally worked well, although improvements were possible by including a small amount of go knowledge in the selection of such windows. Others, such as the graph based and conditional expansion windows, used simple heuristics to dynamically change the size of the window to be appropriate.

A simple mapping quickly becomes too large and memory-expensive for all but trivial windows. Two approaches were tried to deal with the exponential explosion of the pattern database. Section 5.3 introduced a hash-based technique which approximated a complete map in a fixed amount of memory was very effective. Another technique which used

a fixed cache of patterns was also tried, but was not as elegant. Specifically, the hash method achieves graceful degradation as the number of patterns increases beyond the size of memory.

7.2 Future work

The present database is kept in Ishi Press standard format, which is ideally suited to keeping records meant for human consumption, but less than ideal for representing go exemplars. Notable exceptions to the format are a way to notate particular moves as being poor (not to be learned from) or excellent. Often games are not fully recorded, and do not present moves all the way to the end of the game; this is a separate case from both players passing, but the information needed to determine this is usually only in the accompanying English comments. Some extensions to the format may be appropriate.

Placing information about the state of each position at the end of the game (i.e., dead or alive, whose territory, or seki) would assist in the construction of a positional whole-board score evaluation rather than just the expected modification to the score achieved by comparing moves alone.

Here, the classification technique has been applied to moves. A view of the board which more closely parallels human representation of the board and would allow for precise worth estimates in points rather than nonparametrically would be classification of each position, rather than each move. For example, the worth of a move positionally is the difference between the sum of the positional estimates before the move is made and after. The worth of a move would then be the difference in the sum over the entire board of $V(C(p,s))$ where p represents a *position* rather than a move. Each position would yield a number in the range [-1,1][1] indicating the expected value of the effect on the final score the position would have. There are a number of enhancements that could be made, such as representing each category as a probability distribution of results rather than merely expected value.

Moves which are often made only because of local tactical importance aren't likely to have a consistent value; it is dependent on the size of the group(s) being attacked or defended. Moves which have purely strategic merit, on the other hand, are likely to have a consistent value. Not throwing away this information and decomposing move worth components into strategic and tactical components by retaining distributions of effect on error rather than mean effect may be worthwhile.

The pattern preference work in chapter 5 was predicated on having a fixed classification function known *a priori*. Graph based windows and the primitive conditional expansion of windows was a step in the right direction, but each was still just a simple simple heuristic (guess) about what might make an appropriate classifier. A dynamic classifier might yield substantial improvement. The biggest obstacle to this is computational; learning for a fixed window is bad enough. Genetic algorithms were attempted to automaticaly derive classifications in [39].

The learning method I used was simply to subtract an estimated gradient from the

[1] Assuming "Chinese" rules. For "Japanese" rules the range would properly have to be expanded to [-2,2] because dead stones count against, whereas in Chinese they do not.

currect point in the search space at each iteration. Modifying the learning rate (a constant multiplied by the gradient) changes the plasticity of the system; a large learning rate picks up new examples quickly, at the expense of prior knowledge. An adaptive learning rate might be beneficial. Similarly, multiple learning instances could be pooled to produce an improved gradient to seek to minimize the effects of noise.

The last idea, of combining gradients calculated from each exemplar to improve the appropriateness of the gradient leads naturally to the idea of *epoch training*, where all instances in the database are considered and pooled at each iteration. On conventional machines, this is not feasible; there are over 55 thousand board instances in my (rather small) database, and gradient evaluations can easily take many hours each. On a massively parallel machine such as a Connection Machine, however, these instances could be distributed among the many processors.

Altogether, the most successfull method (in terms of least ultimate error) was a conditionally expanded window with slight additional liberty information. After learning, the generated heuristic could become a move suggestor for a go program. The heuristic as generated plays some very nice moves, but because it has no life and death searching or knowledge about constructing two eyes, it is not sufficient as a go player. I hope to soon add this information by separating the tactical and strategic components to moves and allowing small local search to augment the tactical component.

Appendix A

Resources

- This thesis and the accompanying C++ source are available by anonymous FTP. It was compiled using GNU's `g++` compiler, available from `prep.ai.mit.edu`. My code is released under the GNU General Public License, which means you can copy, distribute and modify it to your heart's content as long as you don't sell my code. This is to encourage sharing of go code; at the moment decent go sources and game records are impossible or expensive to obtain. Science shouldn't be 100% competitive; if your code and data aren't shared, then in the long run all your work will have contributed nothing to helping us better understand go.
- Some of the games that were used as training data are free; at the time of this writing, Go Seigen and Shusaku games can be obtained by anonymous FTP at `milton.u.washington.edu`, directory `public/go`. Other games were from *Go World on Disk*. These games are copyright and I cannot freely distribute them with this report. They are, however, available from (this is not an endorsement):

  Ishi Press International
  1400 North Shoreline Blvd., Building A7
  Mountain View, CA 94043.

- *Computer Go* is a bulletin prepared by David Erbach. For information, write to

  Computer Go
  71 Brixford Crescent
  Winnipeg, Manitoba R2N 1E1, Canada.

  or EMail `erbach@uwpg02.uwinnipeg.ca`. At the time of this writing, a subscription is $18 for 4 issues.

- You can get information on go players and clubs in your area from

  The American Go Association
  Box 397
  Old Chelsea Station
  New York, New York 10113

- Executables for David Fotland's original program `COSMOS` are available for HP machines by anonymous FTP at `uunet.uu.net`, directory `games/go`, as well as at `jaguar.utah.edu`, directory `pub/go`. Several other mediocre programs (which are nevertheless strong enough for interested beginners) are available from the site `milton.u.washington.edu`, mentioned above.

- Many of the go program authors referred to in this report follow the `USENET` newsgroup `rec.games.go`, which is a good place to contact authors about particular questions about their programs or arrange network go games with other players of similar skill.

- The author can be reached at `daves@alpha.ces.cwru.edu`.

Appendix B

Methods

The study plots in this report were smoothed from raw data by passing through this program (a simple IIR filter):

```
#include \<stdio.h\>
main() {
    float avg=0.5,y;
    int count=0;
    while (scanf("%g\n",&y)!=EOF) {
        avg=0.9995*avg+0.0005*y;
        count++;
        if ((count%250)==1)
            printf("%g\n",avg);
        }
    }
```

The choice of 0.5 as the starting value was motivated by 0.5 being the error each learning technique would theoretically begin with on average. Other filters would be possible; I eyeballed this as being a nice compromise between smoothness and accuracy. The choice of throwing out all but every 250th value was made to reduce output density.

Appendix C

Game record

C.1 Moves from section 6.1

Here are the moves past number 29 which had no comments:

29 (0.019) o13
30 (0.134) r13
31 (0.033) s13
32 (0.008) q14
33 (0.035) q13
34 (0.008) r12
35 (0.220) n13
36 (0.008) s14
37 (0.005) r15
38 (0.145) j18
39 (0.372) e17
40 (0.238) g17
41 (0.148) c17
42 (0.020) d16
48 (0.017) e18
49 (0.002) c19
50 (0.435) e16
51 (0.134) k17
52 (0.050) k18
53 (0.005) l17
54 (0.040) l18
55 (0.005) m18
56 (0.218) f3
57 (0.422) b5
58 (0.070) b4
59 (0.033) c4
60 (0.008) c5
61 (0.018) c3
62 (0.008) b3
63 (0.042) d5
64 (0.015) c6
65 (0.025) d3
66 (0.002) e4
67 (0.005) b6
68 (0.150) b2
69 (0.002) c7
70 (0.002) d6
71 (0.288) e7
72 (0.291) e6
73 (0.005) f6
74 (0.216) d7
75 (0.002) d8
76 (0.203) e5
77 (0.023) c8
78 (0.467) d12
80 (0.023) e8
81 (0.276) f2
82 (0.019) e2
83 (0.005) g3
84 (0.149) f1
85 (0.002) f7
86 (0.027) e9
87 (0.052) d10
88 (0.063) e10
89 (0.078) d11
90 (0.005) e11
91 (0.016) c12
93 (0.006) e12
94 (0.006) d13
95 (0.009) b12
96 (0.106) g4
97 (0.006) h4
98 (0.006) g5
99 (0.032) g2
100 (0.223) e3
101 (0.451) h7
102 (0.105) h5
103 (0.048) k4
104 (0.063) k5
105 (0.002) j4
106 (0.068) j5
107 (0.242) h10
108 (0.430) g12
109 (0.354) s12
110 (0.049) s11
111 (0.073) s15
112 (0.121) r7
113 (0.284) q8
114 (0.002) s8
115 (0.154) s9
116 (0.006) s7
117 (0.123) o2
118 (0.253) o4
119 (0.140) r2
120 (0.055) s2
121 (0.010) n4
122 (0.027) o5
123 (0.273) o3

124 (0.068) n18
125 (0.015) m17
126 (0.006) m19
127 (0.074) o18
128 (0.011) n17
129 (0.316) o19
130 (0.057) n19
131 (0.047) n16
132 (0.058) o17
133 (0.002) o16
134 (0.002) l19
135 (0.028) p18
136 (0.090) k7
137 (0.454) k8
139 (0.091) l3
140 (0.047) l8
141 (0.038) k9
142 (0.318) j12
143 (0.712) h15
144 (0.391) g8
146 (0.016) m8
147 (0.099) p8
148 (0.134) m6
149 (0.595) m9
150 (0.119) n8
151 (0.068) l9
152 (0.031) j7
153 (0.064) h8
154 (0.017) j8
155 (0.126) j9
156 (0.012) g9
157 (0.113) q9
158 (0.012) r10
159 (0.022) l5
160 (0.046) b13
161 (0.373) b10
162 (0.254) m4
163 (0.275) m5
164 (0.219) l6
165 (0.257) n5
166 (0.279) h9
167 (0.038) h13
168 (0.028) h12
169 (0.026) j13
170 (0.105) k12
171 (0.021) k13
172 (0.060) d9
173 (0.299) a13
174 (0.364) c9
175 (0.018) b9
176 (0.013) a14
177 (0.013) a12
178 (0.222) e19
179 (0.089) d19
180 (0.192) f15
181 (0.704) g18
182 (0.194) f18
183 (0.195) h17
184 (0.177) h18
185 (0.192) g16
186 (0.168) f17
187 (0.042) l12
188 (0.014) o6
189 (0.054) q10
190 (0.151) o8
191 (0.214) n6
192 (0.434) o7
193 (0.824) o11
194 (0.384) t12
195 (0.003) t14
196 (0.640) k11
197 (0.079) g14
198 (0.015) f14
199 (0.003) g13
200 (0.120) f13
201 (0.039) j10
202 (0.112) b16
203 (0.058) g10
204 (0.166) o9
205 (0.330) p11
206 (0.096) t9
207 (0.184) f11
208 (0.009) f12
209 (0.044) p1
210 (0.022) l11
211 (0.010) m11
212 (0.010) r1
213 (0.094) a15

214 (0.143) b14
215 (0.003) a16
216 (0.174) f10
217 (0.030) g11
218 (0.127) b15
219 (0.003) a17
220 (0.024) a5
221 (0.105) b7
222 (0.137) o10
223 (0.059) a6
224 (0.017) n10
225 (0.010) m10
226 (0.011) a4
227 (0.067) g1
228 (0.018) e1
229 (0.061) c10

Appendix D

The Code

The environment The graphs in this paper were produced by C++ program `iku`, on UNIX-flavor machines, and compiled by the program `g++`. See appendix A for how to obtain these things.

This code was run at various times on Sun 3/60s, Sparcstations, DecStations, and a Silicon Graphics Iris 4D monster. Many of the plots shown took days (or weeks) of computer time; this was not a small undertaking. One reason for having so many kinds of machines was because I was scambling to parallelize as much as possible.

The code is written using an indentation style which is non-standard but which I prefer; indentation follows the *logical* hierarchy of the code. I have a special emacs mode which allows me to contract away regions, like an outline editor. I also didn't stick to 80 columns, because often logical indentation dictates long lines. Comments follow the same style, and are placed directly in the code.

Here is a brief overview of the class structure of `iku`. The code is liberally documented, but this overview will help prepare the code diver.

The classes Being in C++, the program was constructed using classes. There are several basic groups of classes:

- Go support classes: Board, Move, Pos, and Game.
- Opponents: Opponent and all the derived classes. Each Opponent may use Patterns and Genomes.
- Patterns: Pattern, PatRep and all its derived classes.
- Genomes: Genome, GenRep and derived classes. (Only present in code used for genetic algorithms[39].)

Figure D.1 shows the essential relationships between the classes.

A *Pos* is a position on a regular 19×19 go board. One additional state that a Pos may be in is *invalid*, which it is initialized to. A *Move* is a type of move, such as a play of a stone at some position on the board, a pass, or a resignation. Moves may have associated comments. A *Board* is the state of a game, between moves. It includes the positions of

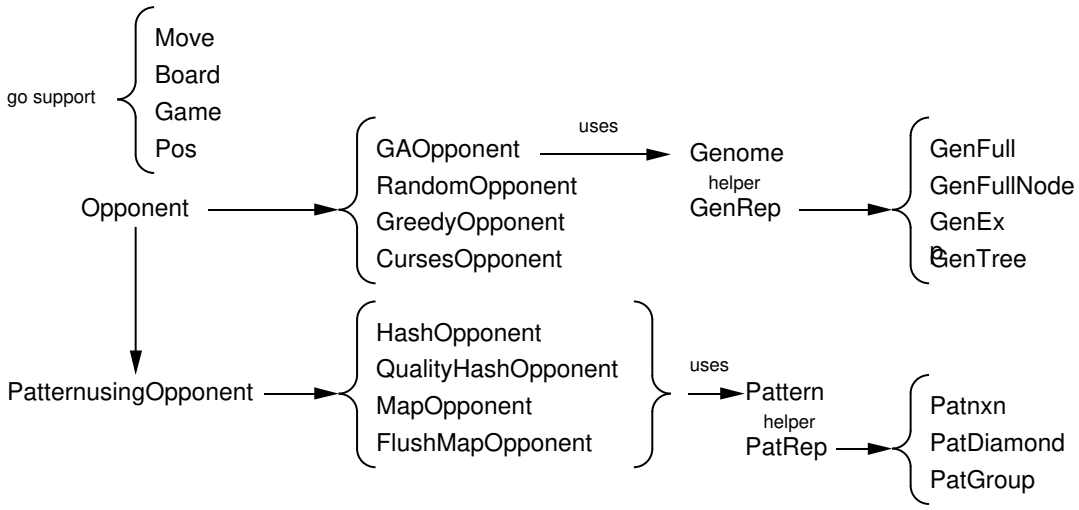

Figure D.1: Classes used in `iku`. Arrows show explicit inheritance; juxtaposed classes have some logical relationship.

stones, whose move it is, how many pieces have been captured by each side, if there is a ko restriction and, if so, where it is. A *Game* is a collection of Moves; any intermediate Board position may be accessed by move number. The constructor Game(String filename) may be used to load a game from a file.

An *Opponent* is just that; it evaluates each playable position, and may be asked to study a given Board/Move combination to try to improve its skill. There are many types of Opponents covered in this thesis. All the PatternusingOpponents use some kind of *Pattern*; the GAOpponent uses some type of Genome.

Constructor dynamics When there is a choice of several kinds of derived types to be constructed, or a numeric parameter is needed (such as a radius for a pattern), this information is constructed from the command line. For example, when the command `iku study hash nxn 2 2000003 allgames` is given, it is parsed as follows:

- `study` means that a number of games are to be presented to an Opponent, and progressive evaluations are to be output as it learns. `study` needs an `Opponent`, the the routine `makeopponent` is called. This gets the next argument from the command line, which is `hash`. `makeopponent` thus calls the constructor for `HashOpponent`. Because `HashOpponent` is derived from `Patternusing-Opponent`, that constructor is called, which goes to the command line and finds `nxn`; this indicates that this opponent will be using a square window, the radius of which is fetched from the command line (2). When the `PatternusingOpponent` constructor finishes, the HashOpponent constuctor decides it needs a hash table size, which it fetches (2000003). Finally, `study` needs a specification for which games to use in learning, which is `allgames`, internally translated to a wild-card specification of all the games available.

Bibliography

1. Benson, D. B., "Life in the game of Go", *Information Sciences*, 10, pp. 17–29, 1976.

2. Benson, D. B., Hilditch, B.R., and Starkey, J.D., "Tree analysis techniques in TsumeGo", *Proceedings of* 6^{th} *IJCAI, Tokyo*, pp. 50–52, 1979.

3. Berlekamp, E., Conway, J. and Guy, R., **Winning Ways**, Academic Press Inc., London, 1982.

4. Bramer, M., **Computer game playing: Theory and Practice**, Ellis Horwood, Chichester, 1983.

5. Boon, M., "A Pattern Matcher for Goliath", *Computer Go*, No. 13, pp. 12–23, 1989–90

6. Chen, K., "Group Identification in Computer Go", in **Heuristic programming in artificial intelligence: the first computer Olympiad**, edited by Levy, D. and Beal, D., Ellis Horwood, Chichester, pp. 195–210, 1989.

7. Chen, K., Kierulf, A., and Nievergelt, J., "Smart Game Board and Go Explorer: A study in software and knowledge engineering", *Communications of the ACM*, 33 no. 2, pp. 152–166, 1990.

8. Chen, K., "The move decision process of Go Intellect", *Computer Go*, no. 14, pp. 9–17, 1990.

9. Liu Dong-Yeh, Hsu Shun-Chin, "The design and construction of the computer go program *Dragon II*", *Computer Go*, No. 10, pp. 11–20, 1988

10. Goldberg, D., **Genetic Algorithms in Search, Optimization and Machine Learning**, Addison-Wesley, 1989.

11. Goldberg, D., **A Note on Boltmann Tournament Selection for Genetic Algorithms and Population-oriented Simulated Annealing**, The Clearinghouse for Genetic Algorithms Report No. 90003, University of Alabama, 1990.

12. High, R., "Mathematical Go", *The American Go Journal*, 24 no. 3, pp. 30–33, 1990.

13. Holland, J., "Processing and processors for schemata", in **Associative Information Processing**, edited by Jacks, E., American Elsevier, New York, 1971.

14. Hsu, Feng-hsiung, Anantharaman, T., Campbell, M., Nowatzyk, A., "A Grandmaster Chess Machine", *Scientific American*, 263 no. 4, pp. 44–50, 1990.

15. Ishida, Y., **Dictionary of Basic Joseki, Vol. 1–3**, Ishi Press (see Appendix A), Tokyo, 1977.

16. Kierulf, A. and Nievergelt, J., "Swiss Explorer blunders its way into winning the first Computer Go Olympiad", in **Heuristic programming in artificial intelligence: the first computer Olympiad**, edited by Levy, D. and Beal, D., Ellis Horwood, Chichester, pp. 51–55, 1989.

17. Kraszek, J., "Heuristics in the life and death algorithm of a Go playing program.", *Computer Go*, No. 9, pp. 13–24, 1988?

18. Kraszek, J., "Looking for Resources in Artificial Intelligence", *Computer Go*, No. 14, pp. 18–24, 1990.

19. Kuhn, T., **The Structure of Scientific Revolutions**, University of Chicago press, Chicago, 1970.

20. Lichtenstein, D. and Sipser, M., "Go is pspace hard", *Proceedings 19th annual symposium on foundations of computer science*, pp. 48–54, 1978.

21. Mano, Y., "An approach to conquer diffficulties in developing a Go playing program", *Journal of Information Processing*, 7, no. 2, pp. 81–88, 1984.

22. Matthews, P., "Inside the AGA rating system", *The American Go Journal*, 24, no. 2, pp. 19, 36–38, 1990.

23. Peterson, J., "A note on undetected typing errors", *Communications of the ACM*, July, 1986.

24. Pearl, J., **Heuristics: intelligent search strategies for computer problem solving**, Addison-Wesley, Massachusetts, 1984.

25. Reitman, W., Kerwin, J., Nado, R., Reitman, J., and Wilcox, B., "Goals and plans in a program for playing Go", *Proceedings of the ACM national conference, San Diego*, pp. 123–127, 1974.

26. Reitman, W. and Wilcox, B., "Perception and representation of spatial relations in a program for playing Go", *Proceedings of the ACM annual conference, Minneapolis*, pp. 37–41, 1975.

27. Reitman, W. and Wilcox, B., "Pattern recognition and pattern-directed inference in a program for playing Go", in **Pattern directed inference systems**, edited by Waterman, D. A. and Hayes-Roth, F., Academic Press, New York, pp. 503–523, 1978.

28. Reitman, W. and Wilcox, B., "The structure and performance of the *Interim.2* Go program", *Proceedings of the of the* 6^th *IJCAI, Tokyo*, pp. 711-719, 1979.

29. Remus, H., "Simulation of a learning machine for playing Go.", *Information processing 1962 (Proceedings of the IFIP Congress, Munich)*, pp. 192–194. North-Holland, Amsterdam, 1962.

30. McClelland, J. and Rummelhart, D., **Parallel Distributed Processing**, MIT Press, London, 1986.

31. Ryder, J. L., **Heuristic analysis of large trees as generated in the game of Go**, PhD. thesis, Stanford University, 1971.

32. Samuel, A., "Some studies in machine learning using the game of checkers", *IBM Journal of Research and Development*, 3, pp.210–229, 1959.

33. Samuel, A., "Some studies in machine learning using the game of checkers—recent progress", *IBM Journal of Research and Development*, 11, pp. 601–617, 1967.

34. Shea, R. and Wilson, R., **The Illuminatus! trilogy**, Dell, New York, pp. 793–4, 1975.

35. Simon, H., "Why should machines learn?", in **Machine Learning: and Artificial Intelligence approach**, edited by Michalski, R. and Carbonall, J. and Mitchell, T., Tioga, Palo Alto California, 1983.

36. Smith, R., **A Go Protocol**, Undergraduate thesis (unpublished), Carnegie–Mellon University, 1989.

37. Shirayanagi, K., **Knowledge representation and its refinement in programming Go**, NTT Software Laboratories, 3-9-11 Midori-cho, Musashino-shi Tokyo 180, Japan, 1989.

38. Shirayanagi, K., "A new approach to programming Go—Knowledge representation and refinement", *Proceedings of the Workshop on New Directions in Game-Tree search*, Edmonton, Canada, May 28–31, 1989.

39. Stoutamire, D., **Machine Learning Applied to Go**, MS thesis, Case Western Reserve University, 1991.

40. Tesauro, G. and Sejnowski, T., "A parallel network that learns to play backgammon", *Artificial Intelligence*, 39, pp. 357–390, 1989.

41. Thorp, E. O. and Walden W., "A partial analysis of Go", *Computer Journal*, 7, no.3, pp. 203–207, 1964.

42. Thorp, E. O. and Walden W., "A computer assisted study of Go on $N \times M$ boards", *Information Sciences*, 4, no. 1, pp. 1–33, 1972.

43. Wilcox, B., "Computer Go", *American Go Journal*, 13, nos. 4–6, nos. 4–6, 44–47 and 48–51; 14, no. 1, 23–28, nos. 5–6; 19, 24–26, 1978–84.

44. Wilcox, B., "Reflections on Building Two Go Programs", *SIGART Newsletter*, 10, No. 94, pp. 29–43, 1985.

45. Yedwab, L., **On playing well in a sum of games**, PhD. thesis, Massachusetts Institute of Technology, 1985.

46. Yoshikawa, T., "The Most Primitive Go Rule", *Computer Go*, No. 13, pp. 6–7, 1989–90.

47. Zobrist, A., **Feature extraction and representation for pattern recognition and the game of Go**, PhD. thesis, University of Wisconson, 1970.