Modelling Uncertainty in the Game of Go

David H. Stern
Darwin College
Cambridge

A dissertation submitted in candidature for the degree of Doctor of Philosophy,
University of Cambridge

Inference Group
Cavendish Laboratory
University of Cambridge

February 2008

Declaration

I hereby declare that my dissertation entitled "Modelling Uncertainty in the Game of Go" is not substantially the same as any that I have submitted for a degree or diploma or other qualification at any other University.

I further state that no part of my dissertation has already been or is being concurrently submitted for any such degree or diploma or other qualification.

Except where explicit reference is made to the work of others, this dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration. This dissertation does not exceed sixty thousand words in length.

Date: .... Signed: ....

David H. Stern
Darwin College
Cambridge
February, 2008

Abstract

Go is an ancient Chinese game of perfect information whose complexity has defeated attempts by Artificial Intelligence researchers to automate play. There is uncertainty about the future course of the game because of the sheer complexity of the game tree and the fact that computers (and humans) have limited computation speed. This thesis presents a number of models which use probability theory to represent and manage this uncertainty.

Firstly, the task of obtaining a probability distribution over available moves in a Go position is considered. A predictive model is trained from records of historical games between expert human players. Each move is represented by the exact pattern of stones in the local neighborhood of the board and millions of these patterns are automatically harvested from the game records. The system can produce a distribution over all the legal moves in a position at a rate of hundreds of positions per second and in 34% of test positions the model perfectly predicts the moves of the expert players. A hierarchical version of the model is also developed with improved performance.

Next, probability theory is used for solving local tactical Go problems. Such problems are solved by searching the game tree until sufficient information is gathered that it is possible to prove by logical deduction whether a particular goal can be achieved. Before a proof is found, probabilities are used to represent the degrees of belief about whether reasoning about a position will lead to a logical proof and these probabilities are used to guide search. The move prediction model of the previous chapter is used to incorporate domain knowledge into search. The system can learn from previous search tasks how to solve future, unseen, problems more efficiently.

Thirdly this thesis considers the task of predicting the final territory outcome of a Go game given a mid-game position. A set of Boltzmann machine models is applied to this task, trained from records of expert games. A generalised version of the Swendsen-Wang sampling algorithm is used to perform efficient inference. Go players confirm that the territory predictions made by these simple models are reasonable.

In the final chapter some alternative Bayesian versions of Monte Carlo planning algorithms are tested on synthetic game trees. With correct setting of the hyperparameters, a simple Gaussian counting model finds the best move faster than the currently popular algorithm (Upper Confidence applied to Trees).

ACKNOWLEDGEMENTS

I am privileged to have had the opportunity to work with David MacKay and Thore Graepel over the last few years. This thesis would certainly not have been possible without their creativity and insight. Their lucid thinking and generous advice has been a source of inspiration.

I also would like to thank the members of the Inference Group at the Cavendish, the Machine Learning and Perception Group at Microsoft Research as well as the members of the computer Go community. I appreciate the helpful discussions with Ryan Adams, Phil Cowans, Julian Dale, Ralf Herbrich, Per Ola Kristensson, Tom Minka, Iain Murray, Scott Sanner, Oliver Stegle, Christian Steinruecken, Philip Sterne, Thomas Thomson, Keith Vertanen, Hanna Wallach, Eric van der Werf, Seb Wills, John Winn and Piotr Zielinski.

Finally, I would like to thank Jane, whose love and support helped enormously during the final months. This thesis is dedicated to my parents, Richard and Susan, and my brother, Jonathan.

This work was supported by a grant from Microsoft Research.

CONTENTS

List of Figures

Notation

Go

N	The size of the Go board (usually 19).
$\mathcal{N} = \{0, \cdots, N\}^2$	The set of points on the Go board.
$\vec{v} \in \mathcal{N}$	A point (vertex) on the Go board $\vec{v} := (v_x, v_y)$.
$\mathcal{C} := \{b, w, e\}$	The set of possible states (black, white, empty) of a point on the board.
$\mathcal{L}(c)$	The set of legal move locations for a board configuration c, $c : \mathcal{N} \to \mathcal{C}$.

Probabilities

$P(Q)$	The probability that proposition Q is true.
$p(x)$	Probability distribution over the value of variable x.
$p(x\|y)$	Probability distribution over the value of x given a value of variable y.
$\mathcal{N}\left(x; \mu, \sigma^2\right)$	Gaussian distribution with mean μ and variance σ^2.
$\Phi\left(x; \mu, \sigma^2\right)$	CDF of the Gaussian distribution with mean μ and variance σ^2.
$\mathcal{N}(\alpha)$	$\mathcal{N}(x) = \mathcal{N}(x; 0, 1)$. By a change of variables, $\frac{1}{\sigma}\mathcal{N}\left(\frac{x-\mu}{\sigma}\right) = \mathcal{N}\left(x; \mu, \sigma^2\right)$.
$\Phi(\alpha)$	$\Phi(x) = \Phi(x; 0, 1)$. By a change of variables, $\Phi\left(\frac{x-\mu}{\sigma}\right) = \Phi\left(x; \mu, \sigma^2\right)$.
$\langle f(x) \rangle_{p(x)}$	Expectation of $f(x)$ for $x \sim p(x)$.
$\mathbb{I}(Q)$	Indicator function, returns 1 if proposition Q is true and 0 otherwise.

Factor Graphs

$\mathcal{V}$	The set of all variables in the model.
$\mathcal{V}_f$	The set of variables that factor f depends on, $\mathcal{V}_f \subset \mathcal{V}$.
$\mathcal{F}$	The set of factors in the model.
$\mathcal{F}_v$	The set of factors that depend on variable v, $\mathcal{F}_v \subset \mathcal{F}$.
$m_{f \to v}$	The message from factor f to variable v.
$m_{v \to f}$	The message from variable v to factor f.

Chapter 1

Introduction

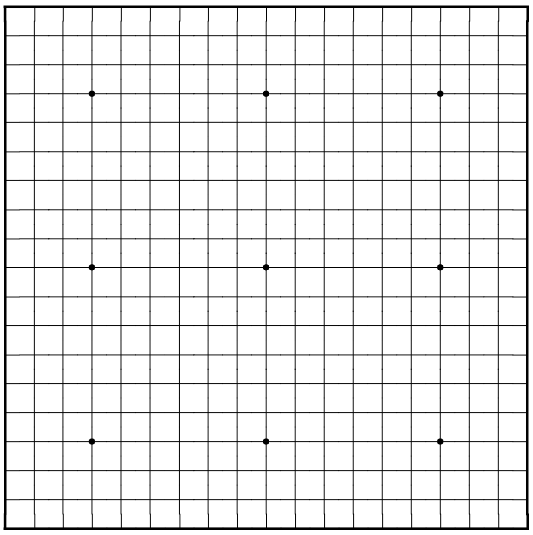


Figure 1.1: The empty Go board.

Long before the development of computers people were fascinated with the possibility of machines playing games of mental skill. In the late 18th century a fake chess playing machine known as the 'Turk' became famous as it was demonstrated throughout the world (Wood, 2002). Since the birth of artificial intelligence (AI), researchers have applied their ideas to game play. The first application of machine learning to a game was by Samuel (1959) for his Checkers player. Game research in the second half of the last century was dominated by brute force Minimax search techniques which proved extremely successful for many games. The defeat of Gary Kasparov by Deep Blue was a well publicised example of the power of this approach (Schaeffer and Plaat, 1997). However, the ancient Oriental game Go has defeated attempts by artificial intelligence researchers to automate play at any level superior to a beginner. Indeed, Go has emerged as one of the current grand challenges of AI research, much like chess was in the 1960s. Many legal moves are typically available and it is difficult to estimate the value of a position. The ensuing defeat of Minimax search forces the pursuit of alternative approaches. Go seems an excellent test-bed for AI ideas, being complex yet well-defined by a small set of simple rules in a fully observed state space. Also, we know that humans are capable of playing Go and learn to do so with relative ease, so any insights we may gain into how to model Go play may help us to understand human intelligence.

1.1 The Game of Go

A great deal of information about Go can be found at `http://gobase.org` (van der Steen, 1994).

1.1.1 History

According to legend, the Chinese game Weiqi originated in the 23rd century BC. One story goes that '[Emperor] Shun regarded his son Shang Jun as stupid and invented Go to instruct him' (Fairbairn, 1995). Another story says the Weiqi board originated as a tool for divination in order to predict the outcomes of future battles during the Shang or Zhou dynasties between the 17th and 6th century BC. At any rate, the view of the Zhou on the cosmos is frequently referred to by later popular writers on the game, for example Ban Gu (32–92 AD) wrote: 'The board must be square and represents the laws of the earth. The lines must be straight like the divine virtues. There are black and white stones, divided like yin and yang. Their arrangement on the board is like a model of the heavens.' (Fairbairn, 1995)

The first known reference to Weiqi is by Confucius in the 6th century BC (Leys, 1997) and the game probably spread to Korea at this time where it was known as 'Baduk' (Fairbairn, 1995). Weiqi became immensely popular in the next millennium and eventually started being played in Japan in the 7th century AD under the name 'Igo' (van der Werf, 2005). The game became known to westerners in the 17th century (Fairbairn, 1995) and some time later took on the English name 'Go'. The number of players in the West is increasing but the game is still most popular in Asia, in particular Korea, Japan and China. There are currently about 60 million players worldwide.

1.1.2 Rules and Definitions

Go is a board game of two players, 'black' and 'white', who take it in turns to place 'stones' (● or ○) on the vertices of an $N \times N$ grid. N is usually 19 but smaller boards may also be used. Once a stone is placed on the board it is never moved but it may be 'captured' (by being surrounded by opponent stones). The aim of the game is for a player to make territory by surrounding areas of the board with their stones. We adapt the Chinese rules of Go (Davies, 1992) for our purposes as follows:

- Go is a game of two players: 'black' and 'white'.
- The Go board is represented as an $N \times N$ graph, $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where the nodes, $\vec{v} \in \mathcal{N}$, correspond to the vertices of a grid so $\mathcal{N} = \{0, \cdots, N\}^2$. We define $\mathcal{E}$ as the set of edges connecting each vertex with its horizontal and vertical neighbors on the board (see Figure 1.1).
- The set of possible states (colours) of each vertex is $\mathcal{C} := \{b, w, e\}$ (black, white, empty) so a configuration (board position) can be given by the function $c : \mathcal{N} \to \mathcal{C}$ (see Figure 1.2).
- A vertex $\vec{v}_1$ of colour $C \in \{b, w\}$ on the grid is said to be 'connected' to a vertex $\vec{v}_2$ (also of colour C) if there exists a path on the graph $\mathcal{G}$ between $\vec{v}_1$and $\vec{v}_2$ passing only through vertices of colour C.

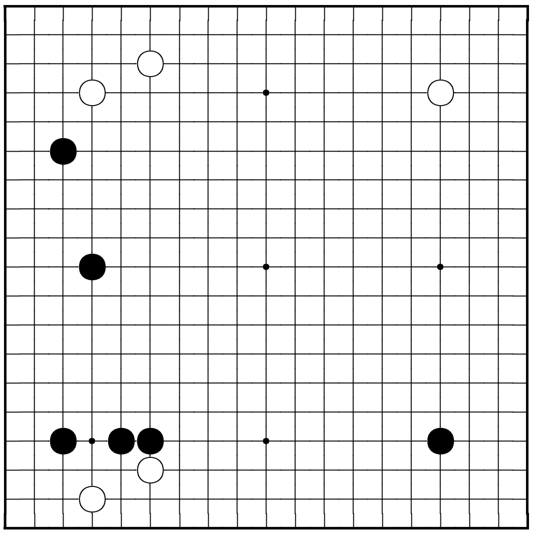
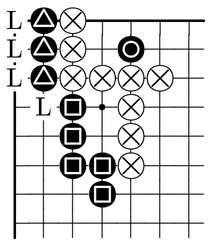


Figure 1.2: Go position during the early stage of a game.

- A set of vertices $\mathcal{B} \subset \mathcal{N}$ entirely connected together and all of colour white or black is called a 'block' or 'chain' (see Figure 1.3).
- A 'liberty' of a chain is an empty node adjacent to the chain on the graph $\mathcal{G}$ (see Figure 1.3).
- Removing a chain, $\mathcal{B}$, means assigning the state of all vertices, $\vec{v} \in \mathcal{B}$, to empty.

Play proceeds as follows:

- Starting with an empty grid, the players alternate turns starting with black.
- A 'move' consists of a player changing the state of one empty vertex to his colour, then removing all chains of the opponent's colour with zero liberties then removing all chains of his colour with zero liberties.
- A 'suicide' move is a move that does not capture an opponent chain and creates a new chain with zero liberties which is immediately removed from the board (see Figure 1.3).
- An 'eye' of colour C is an empty board location where a move would not be suicide for the player of colour C but is a suicide move for the opponent (see Figure 1.4).
- A 'turn' is a move which is not suicide, does not remove an eye of the colour of the player moving and does not repeat an earlier board position (see the ko rules below). If no such moves are available then a player 'passes'.
- After two consecutive passes the game terminates. All empty vertices (the eyes) are changed to the colour of the chain adjacent to them. The score of each player is the number of vertices of their colour on the board. The winner is the player with the highest score.



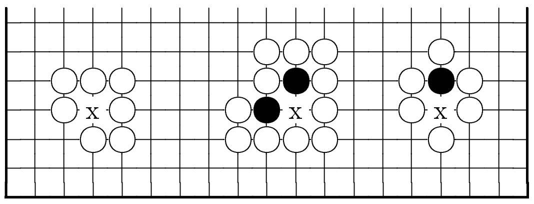


Figure 1.3: **Left**: Example position with the chains labeled. The chain marked with triangles has 4 liberties which are marked 'L'. **Right**: Vertices marked with an x correspond to suicide moves for black (Davies, 1992).

Comparison With Normal Human Rules

In human Go games play will usually stop some time before there are no more moves available. According to the usual rules, a player can choose to pass at each turn and if the two players pass consecutively then the game ends. This raises the difficulty of scoring the game which entails predicting whether each stone would have eventually been captured had the game been played out until no more moves were available. For the purposes of computer play we simplify the rules of the game in that we do not allow a player to pass unless this would mean filling in one of their eyes. Go players will realise that this version of the rules does not handle seki correctly (situations where the player to move first in a local situation will lose a group of stones so neither player makes a move) (Davies, 1992). We ignore this issue as we believe it does not affect the arguments of this thesis.

Life and Death

A set of chains which have eyes in common is called a 'group'. It follows from the rules above that a group of stones with two or more eyes per chain cannot be captured and is said to be a 'living group'. See Figure 1.4 for example configurations of living groups. It requires fewer stones to create a living group at the corner of the board and more stones to create a living group in the center of the board. This is one reason why Go players tend to play in the corners first, then along the sides and finally break out into the middle area of the board.

Often a group can be said to be 'alive' some time before it actually attains two eyes. A set of stones is alive if it can become connected to two eyes in the future course of the game, even taking into account the actions of the opponent. If a stone cannot become part of a living group in the future it is called 'dead'. Predicting the life or death status of stones is an important skill for a Go player to master. Notice that the number of liberties of a chain is the smallest number of opponent stones that must be placed on the board in order to capture the chain, so the number of liberties gives a useful heuristic which indicates how likely the chain is to be alive.

Atari

A chain is in 'atari' if it has one liberty and could thus be captured by the opponent in a single move.

Ladder

A 'ladder' is a forced sequence of direct threats to capture a chain. Let the black chain X be caught in a ladder. In that case, every move by white results in the X having one liberty, forcing black to

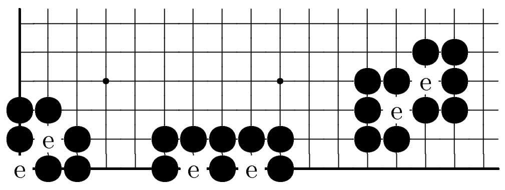


Figure 1.4: Three living black groups with the eyes labeled ‘e’ (van der Werf, 2005).

respond in order to save the chain. Whether the chain will eventually be captured depends on the board position. On an empty board, the chain X will inevitably be captured.

Ko

The ‘ko’ rule is intended to prevent infinite games. We use a version of this rule known as ‘situational superko’. The ‘superko’ rule states that ‘an earlier position must not be repeated’. The ‘situational’ qualifier means that we define a position as the arrangement of stones on the board in conjunction with the colour of the player to move next. The alternative, ‘positional superko’, considers only the arrangement of stones on the board when detecting repetitions (van der Steen, 1994; van der Werf, 2005).

1.2 Minimax Search

The standard algorithm for playing deterministic adversarial games of two players such as chess is *minimax* search (Russell and Norvig, 1995). This algorithm has three aspects: *lookahead*, *evaluation*, and *back-up*. We will refer to the two game players as Max and Min, where Max moves first. Max has the goal to maximise the game score and Min has the goal to miminise the game score.

When in a game position the next move is determined by playing the game forward, alternating play between Max and Min, from the current position a fixed number of steps (N) into the future. Let the number of legal moves available each turn be B (the *branching factor*). The cost of examining the possible positions that can be reached from the first position in this way is $O(B^N)$.

The position at each leaf of this search tree is analysed using a *static evaluation function* which estimates the value of each position. The evaluation function used by a chess program can be straightforward: simply summing up the point values of the pieces on the board seems to work well. The critical thing is that the evaluation is extremely fast: the purpose of lookahead is to insulate us from errors due to the simplicity of the evaluation function.

The value of each position in the tree is determined by propagating information from the leaves of the tree to the root (*back-up*). For each position where Max is to play, the value of the position is taken to be the maximum of the values of each legal successor position (as the Max player will choose the move leading to the most valuable position. For each position where Min is to play, the value of a position is taken to be the minimum of the value of the successors (as the Min player will choose the move leading to the move leading to the position with the most negative value.)

The success of this algorithm for chess is due to the fact that an evaluation function can be written which is fast yet sufficiently accurate that, when combined with a deep-enough search (involving the exploration of billions of positions), leads to good predictions about which moves are valuable. If the evaluation function was slower to calculate or less accurate then minimax search would not be successful for chess.

1.3 Computer Go

A position in Go typically has 220 legal moves available whereas a position in chess has, on average, 35 legal moves. This means that if a Go program could explore the same number of states as a chess program using minimax search the tree would be two thirds as deep ($\frac{\ln 35}{\ln 220}$) and programs would suffer from the horizon effect: important lines of play may be terminated before they are played out, which might lead to a poor estimation of the value of a move (Palay, 1985). However, it is important to realise that minimax search cannot be used to select Go moves even on a small board, where the branching factor is comparable to chess: until recently there were no Go programs capable of play stronger than weak amateur level on a 9×9 board. The real problem is that it is not possible to write a sufficiently accurate yet fast evaluation function for Go. Go programs are typically capable of evaluating 10-20 positions per second whereas a chess program will often analyze millions of positions per second (van der Werf, 2005). This means that chess programs need to be programmed with little knowledge about tactics and strategy; instead they apply brute force search and effectively learn to play the game from scratch with every move. On the other hand, most Go programs tend to harness a great deal of knowledge about the game which must either be pre-programmed by hand or learned automatically offline. This means that Go programs are typically complex (Muller, 2002; Fotland, 1993).

1.3.1 Traditional Approaches to Computer Go

The first computer program to win against a (weak) human Go player was written by Zobrist (1970). Go research accelerated in the 1980s as computers became more affordable and a million dollar prize was offered in 1985 for a program winning against a professional player (the Ing prize). This prize expired, unclaimed, in 2000.

Local Search

Usually a Go program will not use much global search (because of the difficulty of position evaluation as described above). However it is possible to use search to determine certain local properties of the position such as whether a group is alive (Wolf, 1994, 2000), whether a particular stone can be captured, or whether a pair of stones are likely to become connected (Thomsen, 2000) as evaluating these goals is straightforward. This type of local goal-directed search (as opposed to global search) is possible in Go because evaluating simple sub-goals such as 'capture stone A' or 'connect stones A and B' is simple and fast, unlike estimating the heuristic 'probability of win'. Another application of search to Go is for solving the late end-game when the game tends to break down into a number of subgames and can thus be solved by a divide-and-conquer approach (decomposition search) (Muller, 1999).

Static Knowledge

A common method for representing static knowledge about the game is by matching patterns of stones on the board which are labeled with various properties such as the life/death status of stones, good locations for moves or connectivity between stones (Boon, 1990; Bouzy and Cazenave, 2001; Muller, 2002). Also, mathematical methods have been developed for the determination of living groups (Benson, 1976) and the analysis of end-games (Berlekamp and Wolfe, 1994).

The Combination: an Evaluation Function

A combination of the results of goal-directed search tasks and additional static (rule based or heuristic based) knowledge is typically used to build a detailed hierarchical abstract representation of the board position (Fotland, 1993). This representation is used by a move generator to select possible moves. A candidate set of moves may then be generated and the resulting positions evaluated, which will often involve additional goal directed searches to determine life and death as well as further static analysis. An 'influence function' may be used to predict territory (Zobrist, 1970).

1.3.2 Machine Learning Approaches to Go

These Go programs use many hand tuned rules and potentially unreliable heuristics. It is notoriously difficult to tune such rule-based expert systems to work well. There is no principled way of handling conflicting rules or taking account of the fact that a rule may not always be correct depending on the wider context (Russell and Norvig, 1995). The more complex the expert system becomes the more difficult it becomes to tune. Programming Go knowledge by hand takes a great deal of time and the well known traditional Go programs took years to develop (see Fotland (1993) for example). Also, it is difficult for a human to determine by hand what knowledge is most useful for a Go program to perform well: even strong Go players cannot easily describe what thought processes they are going through when selecting a move.

For these reasons a number of approaches to automatically adding knowledge to Go programs and tuning them have been investigated. The sources of this knowledge fall into three main categories. Firstly, as recognised by Samuel (1959), it is possible to store information gained by exploring the state space generated by the rules of a game, so that in future it is possible to predict the result of exploration without actually having to carry it out, in effect compressing the game tree and reducing the state space. Secondly, humans have learned a great deal about Go over the last few millennia and this knowledge is implicit in abundant records of human games. A particularly large resource is games that have been played online on the Internet Go servers. Thirdly, a program can learn from Go positions manually labeled with important properties.

Some attempts have been made to extract rule-based knowledge from the first source by offline retrograde analysis of local searches (Cazenave, 1996; Bouzy, 2001; Cazenave, 2001). However, possibly because of the problems with rule-based systems mentioned above, more impact has been made by work on non rule-based machine learning techniques. Reinforcement learning from self play using the temporal difference (TD) algorithm famously produced a world-class backgammon player (Tesauro, 1994). Inspired by this success a number of researchers have applied this idea to Go. Schrauldolph *et al.* (1994) trained a convolutional neural network to evaluate Go positions by TD learning from self-play. Following this work, more success with self-play learning was achieved by Enzenberger (1996, 2003) who incorporated prior game knowledge into his neural network architecture. In particular he took account of the common fate property of chains. This neural network was the key component of his Go program, NeuroGo, which was capable of competing with the mid-level traditional Go programs of the time. More recently Silver *et al.* (2007) trained by self play a system that represented a position as a combination of local patterns of stones.

The approaches to learning from unlabeled game records have been directed at the task of move prediction. One method is to extract salient patterns of stones from positions in the games and to learn properties of these patterns based on the decisions made by the human players (Stoutamire, 1991; Bouzy and Chaslot, 2005; de Groot, 2005). Neural networks using a number of pre-processed features as their inputs have also been trained to predict moves in game records (van der Werf *et al.*, 2002). Dahl (1999) used a hybrid approach in order to produce a surprisingly strong Go player: Honte. His system was composed of several neural networks, one for generating candidate moves which was trained from expert game records and two others for evaluating Go positions which were trained by TD under self-play. Another recent approach to train a neural network from game records is by Wu and Baldi (2007).

One way that labeled positions can be used by learning algorithms is for learning to solve local (tactical) Go sub-tasks. Sasaki *et al.* (1999) describe an application of neural networks to solving problems of life and death in Go where the task is to choose the best move to kill a group in a given position (so called *tsume* go problems). Their training data was in the form of positions labeled with the correct move in each case. Graepel *et al.* (2001) made use of the common fate property of chains (in a manner similar to Enzenberger (1996)) to construct an efficient graph-based representation of the board. They trained a Support Vector Machine to use this representation to solve Go problems, again training from labeled positions. In his PhD thesis van der Werf (2005) describes applications of neural networks to the Go sub-tasks of territory prediction, scoring final positions and life and death determination.

1.4 Monte Carlo Search and the State-of-the-Art in Go

I suggest there is a continuum between *static* and *dynamic* approaches to position evaluation. A dynamic evaluation method makes use of the rules of the game and learns about a position by exploring the game state space. A pure static evaluation method does not take into account the rules of the game and does not attempt to explore the state space, but instead applies general knowledge associated with game states.

An example of a pure dynamic technique is *Monte Carlo planning* which takes into account only the rules of the game (Brügmann, 1993). In this approach games are played from the current position to the end of the game, using a stochastic procedure (*policy*) to choose the move at each position in the game. The final win/loss outcomes of these games after they are completely played out are taken to be samples from the distribution over the value of the initial position. In this way Monte Carlo planning provides a stochastic, dynamic, evaluation function for Go positions. In contrast, Minimax search is only partially dynamic as it uses the rules of the game in order to search into the future but also has a crucial static component: the evaluation function.

A pure static evaluation method for Go might take into account features such particular patterns of stones which appear on the board or static features of moves such as the value of moving at the edge of the board and so on.

1.4.1 Monte Carlo Planning with UCT

As discussed in the previous section, successful Go programs have predominantly used static knowledge because no fast evaluation function has been found to make minimax search effective, although some dynamic evaluation is included in the form of local tactical searches. However, recently Gelly *et al.* (2006) discovered that an adaptive Monte Carlo planning algorithm, 'Upper Confidence applied to Trees' (UCT) (Kocsis and Szepesvari, 2006) could be very successfully applied to Go on small sized boards and this has led to a sudden shift in the focus of Go research from static knowledge-based approaches towards techniques which are almost entirely dynamic in nature. As in the older Monte Carlo method described above, each iteration of UCT involves playing a game from the current position until the end of the game (this is called a 'rollout'). In this way the game tree is explored and each game position which is encountered along the way is valued by taking the mean of the win/loss values of the rollouts passing through that position. The difference with UCT is that the policy used to select the moves during the rollouts is adapted based on the outcome of previous rollouts. As further iterations are carried out more valuable positions are explored more frequently.

UCT provides a unified attack against the two key problems of computer Go discussed at the start of this chapter: determining the evaluation function and exploring the huge game tree usefully. Typically, when UCT is implemented, the two tasks of evaluation and search are separated out as in minimax search.

Position Evaluation

In Monte Carlo planning, the positions at the leaves of the search tree are evaluated by playing rollouts from these positions. This avoids the most challenging problem of computer Go: writing a static evaluation function. This is because at the end of each rollout evaluation is trivial (just count the score as in Section 1.1.2). Since this evaluation method is noisy, standard minimax search cannot be applied. Also, since the game tree is so large in Go it is unknown if it could be applied usefully anyway. However, UCT provides a method of searching the game tree which solves these problems.

Game Tree Exploration vs Exploitation

The tree search part of the algorithm requires that the part of the game tree which has been explored so far is stored in memory. Each iteration of the algorithm involves walking down the tree from the current position to a leaf (a previously unexplored position) and then adding this leaf position to the tree and sampling its value by performing a stochastic Monte Carlo rollout. UCT provides the method to explore the tree so as to efficiently assign computational resources. As we will see in Chapter 6 the algorithm achieves this by treating each node as a separate multi-armed bandit (where each arm corresponds to a legal move) (Kocsis and Szepesvari, 2006). Moves are chosen so as to balance 'exploitation' and 'exploration' (Sutton and Barto, 1998). Exploitation means further examining parts of the tree that have been seen before and already predicted to be valuable, this is useful to do because it means that nodes higher up in the tree can have their estimates improved. Exploration means to examine parts of the tree that have not been investigated previously in case they turn out to be valuable. In contrast to minimax search, no information is propagated up the tree in a step-wise fashion. Rather, the value of each node is estimated directly as the mean of all the outcomes of the rollouts passing through that node. The trick which makes UCT converge to the minimax solution is that this average becomes dominated by the results of following better and better move sequences until, ultimately, only the correct sequence of moves is being followed. Chapter 6 covers this in more detail.

1.4.2 Combining Knowledge with Monte Carlo Search

In order to make Monte Carlo planning competitive with the traditional Go programs it was found that is was necessary to include some static game knowledge in the form of rules and local patterns (although it is important to point out the knowledge included in these programs is still extremely simple compared to the knowledge included in the traditional Go programs). Gelly *et al.* (2006) initially used small local patterns to bias play during the rollouts to make them more informative. Next, additional domain knowledge was included in order to initialise the values of newly encountered positions (Gelly and Silver, 2007) in the game tree. Some other Go-specific heuristics were also included which led to greater performance increase. Currently the resulting Go program, MoGo, is the worlds strongest Go program on the 9×9 Go board and plays at a level comparable with strong amateur Go players. On the 19×19 board MoGo has achieved a similar level to the traditional Go programs but is still weak compared to human play. Another successful UCT based Go program is Crazystone which also combines static Go knowledge (in the form of patterns of stones) with UCT (Coulom, 2007) and is of a similar strength to MoGo. At the time of writing, one of the biggest questions which computer Go researchers are trying to answer is if UCT can be applied successfully to the full size Go board.

1.5 Limited Rationality and Uncertainty

Typically in machine learning, uncertainty is said to derive from observation 'noise', due to unpredictable aspects of the data-set in question. For example, imagine we are interested in building a model for predicting the future price of a stock given the historical sequence of previous prices for that stock. The future price will depend on many external factors such as news events, the success of rival companies or macroeconomic changes. All of the information needed to predict the future evolution of the price will probably not be contained in the price history. In fact, it would be extremely difficult to determine all of the relevant information. It may be possible to predict stock prices to some degree of accuracy but some uncertainty we always remain, however good the model is, due to the fact we only have imperfect information to base our prediction on.

In Go, uncertainty comes from another source: limitations on computing power (also termed 'limited rationality') (Simon, 1982; Russell and Wefald, 1991). A board position, in conjunction with the rules of the game, contains all of the information necessary for perfect play. However, the sheer complexity of the game tree prevents sufficient exploration of the state space, resulting in uncertainty about the future course and outcome of the game. The Japanese have a word, *aji*, much used by Go players. Taken literally it means 'taste'. Taste lingers and likewise the influence of a Go stone lingers (even if it appears weak or dead) because of the uncertainty about the effect it may have in the future. In this thesis I propose that it is useful to represent and manage this uncertainty using probability theory. A number of models are presented which demonstrate this principle.

1.6 Bayesian Uncertainty

I propose that progress can be made if we adopt a unified, rigorous and consistent mechanism for representing and managing the uncertainty that results from the complexity of the game tree: probability theory.

Following Jaynes (2003) I view probabilities as an extension of logic in which degrees of plausibility are represented as probabilities (also known as 'degrees of belief'). Cox (1946) stated a set of axioms under which degrees of belief can be represented by real numbers such that the representation satisfies desired conditions of consistency and correspondence with intuitions about the way a rational agent should reason. The degree of plausibility of a proposition Q being true is represented as the real number, $P(Q)$, the probability of Q. If Q is known to be certainly true then $P(Q) = 1$. If Q is known certainly to be false then $P(Q) = 0$. The following conditions hold for probabilities:

$$P(Q) \geq 0$$

$$P(Q) + P(\neg Q) = 1$$

In general we are interested in representing uncertainty about the possible values of numerical variables which may be continuous or discrete. If X is a variable then the probability $P(X = x)$ represents the degree of belief that X will take on the value x. A probability distribution is a function which returns the probability of each of the possible assignments to its input variable(s) so $p(x) = P(X = x)$. In this thesis I will often make use of the abbreviated notation that the name of a variable also represents its value. A probability distribution, $p(x)$, must satisfy:

$$p(x) \geq 0, \forall x$$

$$\sum_x p(x) = 1$$

and for continuous variables:

$$\int p(x)dx = 1$$

where

$$\int_a^b p(x)dx = P(a \leq x \leq b).$$

Probability distributions may be used to represent the probability of the joint assignment of a set of variables, for example, $p(x, y)$ returns $P((X = x) \wedge (Y = y))$, the joint probability that $X = x$ and $Y = y$. Note that

$$\sum_x \sum_y p(x, y) = 1$$

with the sums replaced by integrals for continuous variables. This leads to the 'marginalisation' property:

$$p(x) = \sum_y p(x, y).$$

The probability that a variable has value $X = x$ given the value of another variable $Y = y$ is known as a conditional probability. The conditional distribution, $p(y|x)$, is a probability distribution over the possible values of Y given that X takes on the particular value, x. If $p(x|y) = p(x)$ then X and Y are independent. The conditional distribution is related to the joint distribution by the 'product rule':

$$p(x, y) = p(y|x)p(x).$$

Noting that $p(x, y) = p(x|y)p(y)$ we arrive at Bayes' rule (Bayes, 1763):

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)} = \frac{p(x|y)p(y)}{\sum_y p(x|y)p(y)}.$$

If we observe X to be equal to the value a then this equation enables us to update our 'prior' belief distribution, $p(y)$, over the value of y in light of this new observation and thus infer a new 'posterior' distribution, $p(y|x = a)$. This is an example of Bayesian inference. The distribution $p(x|y)$ is known as the 'likelihood' function which returns, for a particular observed value of the variable x, the likelihood of each possible setting of y (the probability of x given y). The task of 'probabilistic modeling' is to produce a probability distribution which places high probability on future empirical data, often given some parameters. An important application of Bayesian inference is to infer posterior probability distributions over the values of the parameters of a model in the light of observed data, given some prior belief about their plausible values.

1.7 Training Data

One advantage of the game of Go as a test-bed for machine learning techniques is that a large number of games are available for training data. I obtained games from two main sources. Firstly I obtained a collection of historical expert games, 'Games of Go on Disk' (GoGoD)[1] by T. Hall and J. Fairbairn. The version of this database used here contained 22,000 Go games. A second source is games played on the various internet Go servers. Because Go players are distributed quite sparsely across the world these servers are extremely popular and many high quality games records are available from them. I obtained most of these games from a pre-compiled collection which is commercially available: BiGo[2]. This contains about 1,000,000 games in total although only a subset were used for the experiments in this thesis.

1.8 Thesis Outline

In this thesis I present a number of applications of machine learning to modeling aspects of Go play. In Chapter 2 I introduce the probabilistic modelling techniques and mathematical results that are used throughout the rest of the thesis. Chapter 3 covers an approach for ranking patterns in order to predict human move choices in records of games between expert players. Next, in Chapter 4, I apply probability theory to solve local tactical Go problems and to learn how to solve future, unseen, problems more efficiently. In Chapter 5 I focus on predicting the final point-wise territory outcome of games given a mid-game position using a simple conditional Markov random field model. Recently there has been success with applying Monte Carlo planning techniques to computer Go (Gelly *et al.*, 2006). Chapter 6 presents some initial findings into alternative approaches to the algorithms currently used for this task. Finally, in Chapter 7 we present some conclusions and ideas for future work.

[1]GoGoD is available at `http://www.gogod.co.uk`.

[2]BiGo is available at `http://bigo.baduk.org`.

Chapter 2

Representing Uncertainty

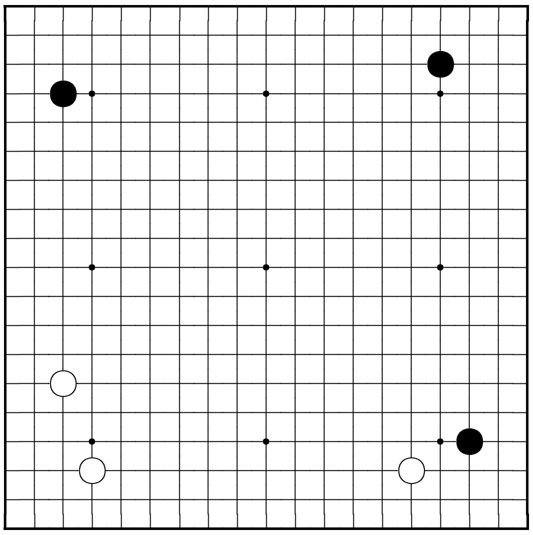


Figure 2.1: *fuseki*

In this thesis I present a number of applications of probability theory to represent and manage the uncertainty that results from the complexity of the Go game tree. This chapter introduces the technique of probabilistic modelling and a number of mathematical results are derived which will be invoked throughout the rest of the thesis.

In general, a probabilistic model is a probability distribution over the values of a set of variables, $\mathcal{X} = \{x_1, x_2, \cdots, x_n\}$. These variables may represent observable features of the real world or they may be unobservable 'latent' variables. An assignment of a value to each of these variables, $\{x_1 = a, x_2 = b, .., x_n = c\}$, represents a hypothesis about a plausible assignment of their values. The probability distribution, $p(x_1, x_2, x_3, ...)$, returns, for each possible hypothesis, a probability representing our degree of belief that this hypothesis is true.

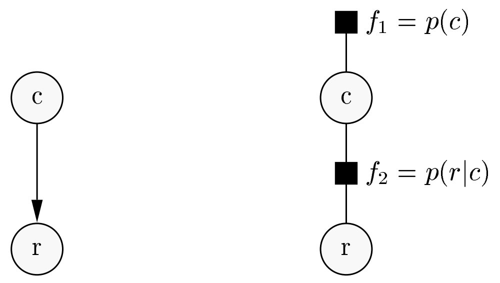


Figure 2.2: Clouds and rain model. Left: Bayesian network. Right: Factor graph.

2.1 Graphical Models

Graphical models represent relations between variables using graphs. They were independently developed in several fields and have been adopted for various applications such as expert systems, data mining, control tasks and error correcting codes. For a comprehensive treatment of the methods as they are used in machine learning I refer the reader to Pearl (1988), MacKay (2003) and Bishop (2006).

2.1.1 Bayesian Networks

Clouds and Rain Example

As a first example, imagine we are interested in modeling the joint plausibility of two variables, c: whether it is cloudy and r: whether it is raining. We choose to model the joint distribution as the product of two terms: a prior belief about whether it is likely to be cloudy and a conditional probability of rain given that it is cloudy:

$$p(c, r) = p(c)p(r|c)$$

If we observe that it is raining we can infer a posterior distribution over the probability of clouds by Bayes' rule:

$$p(c|r = \text{TRUE}) = \frac{p(c)p(r = \text{TRUE}|c)}{\sum_c p(c)p(r = \text{TRUE}|c)} \tag{2.1}$$

The dependencies between the variables can be depicted graphically by drawing a 'Bayesian network' (Pearl, 1988) (Figure 2.2 Left).

Sprinkler Example

A more complex model extends the above to give the joint distribution of clouds (c), rain (r), wet grass (w), a sprinkler running (s) and there being lots of traffic on the roads (t). Here we assume that the joint distribution is given by

$$p(c, r, t, w, s) = p(c)p(r|c)p(s)p(w|r, s)p(t|r). \tag{2.2}$$

The Bayesian network for this model is given by Figure 2.3 (left). By decomposing the joint distribution into a product of factors, each only operating on a subset of the variables, we are making 'conditional independence' assumptions. For example, take the three variables t, r, and c. If we observe r then c and t become independent or in other words, $p(t|r, c) = p(t|r)$.

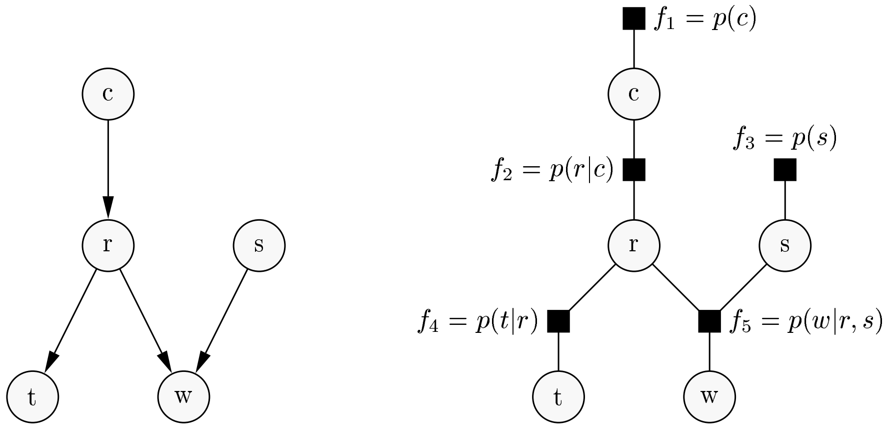


Figure 2.3: Sprinkler example. **Left:** Bayesian network. **Right:** Factor graph.

If we observe that the the grass is wet ($w = \text{TRUE}$) we can calculate a posterior marginal probability of clouds by:

$$\begin{aligned} p(c|w = \text{TRUE}) &= \frac{p(c, w = \text{TRUE})}{\sum_c p(c, w = \text{TRUE})} && (2.3)\\ &= \frac{p(c, w = \text{TRUE})}{Z} && (2.4)\\ &= \frac{1}{Z}\sum_s \sum_r \sum_t p(c, r, t, s, w = \text{TRUE}) && (2.5) \end{aligned}$$

The cost of this summation scales exponentially with the number of unobserved variables in the model. However, we can take advantage of the distributive law to give:

$$\begin{aligned} p(c|w = \text{TRUE}) &= \frac{1}{Z}\sum_s \sum_r \sum_t p(c)p(r|c)p(s)p(w = \text{TRUE}|r, s)p(t|r) && (2.6)\\ &= \frac{1}{Z}p(c)\sum_r p(r|c)\sum_s p(s)p(w = \text{TRUE}|r, s)\sum_t p(t|r) && (2.7)\\ &= \frac{1}{Z}p(c)\sum_r p(r|c)\cdot m_1(r)\cdot m_2(r) && (2.8) \end{aligned}$$

where

$$m_1(r) = \sum_s p(s)p(w = \text{TRUE}|r, s) \qquad (2.9)$$

and

$$m_2(r) = \sum_t p(t|r). \qquad (2.10)$$

This calculation is called ‘variable elimination’ and its computational cost depends on the order in which variables are summed out or ‘eliminated’. Notice how the inference problem decomposes into a sequence of smaller sums and products.

2.1.2 Factor Graphs

A factor graph $\{\mathcal{V}, \mathcal{F}, \mathcal{E}\}$ represents a factorised function as a bipartite graph with *factor* nodes $f \in \mathcal{F}$ and *variable* nodes $v \in \mathcal{V}$ (represented by squares and circles respectively) (Kschischang *et al.*, 2001). Each factor node in the graph corresponds to a factor in the function and each variable node to a variable. The edges $e \in \mathcal{E}$ in the graph indicate which variables each factor is a function of. In general a factor graph can be used to represent a distribution of the form

$$p(\mathcal{V}) = \frac{1}{Z} \prod_{f \in \mathcal{F}} f(\mathcal{V}_f),$$

where $\mathcal{V}_f$ is the set of variables that factor f depends on. The distribution is the product of all the factors renormalised (by the factor $\frac{1}{Z}$). A Bayesian network can be represented as a factor graph where each factor corresponds to one of the conditional probabilities or prior probabilities in the Bayesian network and in that case the renormalisation factor Z is equal to 1 (see Figures 2.2 and 2.3). However, not all factor graphs have a Bayesian network equivalent. Factor graphs are a generalisation of many other earlier methods, for example 'Tanner Graphs' for the decoding of error correcting codes (Tanner, 1981) as well as Markov Random Fields.

2.2 Message Passing On Factor Graphs

2.2.1 The Sum Product Algorithm

Equation (2.8) shows how marginalisation can be simplified as a sequence of less costly sums and products due to the conditional independence assumptions made when constructing the model. In a factor graph with no cycles each variable, v, *separates* the graph into a set of subgraphs, one for each factor, $f \in \mathcal{F}_v$, connected to the variable in question, v (MacKay, 2003). Since no variable occurs in more than one of these sub-graphs the marginal $p(v)$ can be calculated as the product of separate smaller summations: one for each $f \in \mathcal{F}_v$. This idea leads to a message passing implementation of variable elimination called the *sum product algorithm* (Kschischang *et al.*, 2001) for calculating marginal distributions of variables in a model. This algorithm is a generalisation of Belief Propagation (BP) by Pearl (1988) as well as algorithms for decoding error correcting codes (Tanner, 1981). See Kschischang *et al.* (2001) for a more detailed discussion of the origins of the algorithm and how it relates to other methods.

Messages are propagated about the graph according to the following pair of equations:

$$m_{f \to v}(v) = \sum_{\mathcal{V}_f \setminus v} f(\mathcal{V}_f) \cdot \prod_{v_i \in \mathcal{V}_f \setminus v} m_{v_i \to f}(v) \tag{2.11}$$

$$m_{v \to f}(v) = \prod_{f_i \in \mathcal{F}_v \setminus f} m_{f_i \to v}(v) \tag{2.12}$$

and the marginal distribution of a variable $v \in \mathcal{V}$ is calculated by

$$p(v) = \frac{\prod_{f \in \mathcal{F}_v} m_{f \to v}(v)}{Z} \tag{2.13}$$

where $\mathcal{F}_v$ is the set of factors connected to variable v, $m_{f \to v}(v)$ is a *message* from factor f to variable v and $m_{v \to f}(v)$ is a message from variable v to factor f and Z is a normalisation constant. For an

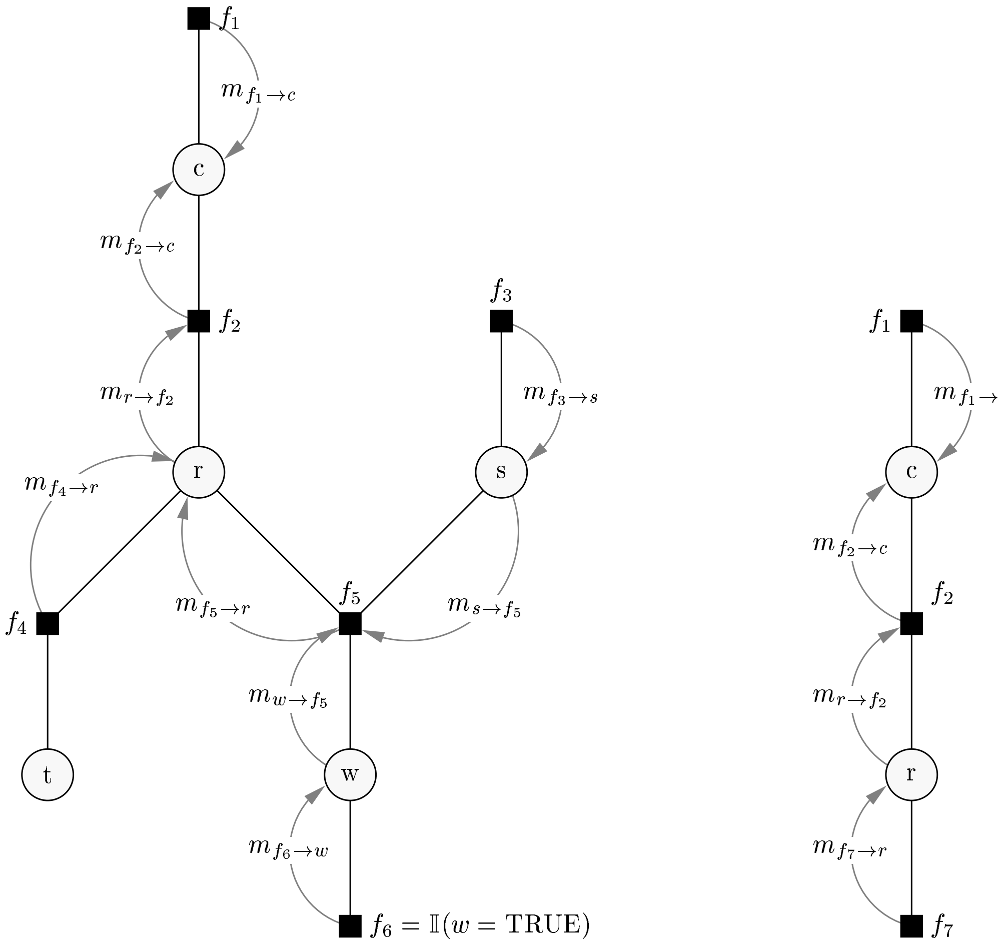


Figure 2.4: **Left**: Sprinkler factor graph. The grass is observed to be wet by including the factor $f_6(w) = \mathbb{I}(w = \text{TRUE})$. Messages are propagated in order to calculate the posterior marginal, $p(c|w = \text{TRUE})$, as shown. **Right**: The top part of the sprinkler model. The factor f_7 corresponds to the evidence propagated from the lower part of the full model, summarised in the message $m_{f_7\to r} = m_{f_4\to r}(r) \cdot m_{f_5\to r}(r)$. If r is observed (for example to be true by setting $f_7 = \mathbb{I}(r = \text{TRUE})$) then c becomes independent of the other variables and then we have the rain model of Figure 2.2.

acyclic graph there is an update schedule such that each message needs to be calculated only once in order to determine the marginal distributions of all the variables exactly. On a tree this means that firstly the messages are calculated in a sweep from the leaves to the root of the tree and in a second sweep the messages from the root to the leaves are calculated.

Observations

Empirical evidence in the form of observations of variable values or constraints can be taken into account by including observation factors of the form

$$f_o(\mathcal{V}_{f_o}) = \mathbb{I}(Q(\mathcal{V}_{f_o}))$$

where $\mathbb{I}(Q(\mathcal{V}_{f_o}))$ is the *indicator function* which returns 1 if logical proposition, $Q(\mathcal{V}_{f_o})$, is true and 0 otherwise. For simple observations of the value of a variable, v, we let $f_o(v) = \mathbb{I}(v = a)$ where a is the observed value of v. Another type of observation used in this thesis is an inequality constraint, for example $f_o(v) = \mathbb{I}(v > b)$.

Sprinkler Example

For the sprinkler example the messages are shown in Figure 2.4 (left). The grass is observed to be wet and this is indicated by the factor $f_6(w) = \mathbb{I}(w = \text{TRUE})$. Message calculations proceed starting with the observation:

$$m_{f_6 \to w}(w) = f_6(w) = \mathbb{I}(w = \text{TRUE}),$$

(from equation 2.11). Next we calculate,

$$m_{f_3 \to s}(s) = f_3(s) = p(s),$$

the prior on s (also from 2.11), now using (2.12) we calculate,

$$m_{w \to f_5}(w) = m_{f_6 \to w}(w) = \mathbb{I}(w = \text{TRUE}),$$

and

$$m_{s \to f_5}(s) = m_{f_3 \to s}(s) = p(s).$$

At this point we have all the messages needed to calculate the message from the factor f_5 to the variable r (using 2.11):

$$\begin{aligned} m_{f_5 \to r}(r) &= \sum_w \sum_s f_5(w, r, s) \cdot m_{s \to f_5}(s) \cdot m_{w \to f_5}(w) \\ &= \sum_w \sum_s p(w|r, s) \cdot p(s) \cdot \mathbb{I}(w = \text{TRUE}) \\ &= \sum_s p(w = \text{TRUE}|r, s) \cdot p(s). \end{aligned}$$

Now from (2.11) and (2.12) :

$$m_{f_4 \to r}(r) = \sum_t p(t|r).$$

Notice that this message equals unity and will have no effect on the final result. Continuing, we use (2.12) to calculate

$$m_{r \to f_2}(r) = m_{f_5 \to r}(r) \cdot m_{f_4 \to r}(r) = \sum_s p(w = \text{TRUE}|r, s) \cdot p(s) \cdot \sum_t p(t|r).$$

So the bottom half of the model effectively can be replaced by a single factor $f_7(r) = m_{f_5 \to r}(r) \cdot m_{f_4 \to r}(r)$ (See Figure 2.4 Right). Now, equation 2.11 gives us,

$$m_{f_2 \to c}(c) = \sum_r f_2(r, c) \cdot m_{r \to f_2}(r) = \sum_r p(r|c) \cdot \sum_s p(w = \text{TRUE}|r, s) \cdot p(s) \cdot \sum_t p(t|r).$$

We also have that

$$m_{f_1 \to c}(c) = f_c(c) = p(c),$$

the prior on clouds, c, so the marginal posterior is given by equation (2.13):

$$\begin{aligned} p(c|w=\text{TRUE}) &= \frac{1}{Z}\cdot m_{f_1\to c}(c)\cdot m_{f_2\to c}(c) \\ &= \frac{1}{Z}\cdot p(c)\cdot \sum_r p(r|c)\cdot \sum_s p(w=\text{TRUE}|r,s)\cdot p(s)\cdot \sum_t p(t|r). \end{aligned}$$

Notice how we recover the result of Equation (2.8).

Cost

The computational cost is exponential in the number of variables in the largest factor, not the number of variables in the model so in principle models of arbitrary size are possible as long as the individual factors remain simple and there are no cycles in the corresponding factor graph.

Separation

Figure 2.4 (right) shows the top part of the sprinkler model with the bottom half of the model replaced by a single factor node, $f_7(r) = m_{f_4\to r}(r)\cdot m_{f_5\to r}(r)$. The variable, r, separates the other variables in the graph because of conditional independence: if r is observed by setting $f_7(r) = \mathbb{I}(r=\text{TRUE})$ then c becomes independent of the other variables and the model reduces to the simple rain/clouds model of figure 2.2. In this case the posterior $p(c|r=\text{TRUE})$ is given by

$$\begin{aligned} p(c|r=\text{TRUE}) &= \frac{1}{Z} m_{f_2\to c}(c)\cdot m_{f_1\to c}(c) \\ &= \frac{1}{Z}\sum_r \mathbb{I}(r=\text{TRUE})\cdot p(r|c)\cdot p(c) \\ &= \frac{p(r=\text{TRUE}|c)\cdot p(c)}{\sum_c p(r=\text{TRUE}|c)\cdot p(c)}, \end{aligned}$$

which is Bayes' rule (compare with equation 2.1).

The factor graph methodology in conjunction with the sum-product algorithms gives us a unified method for performing inference on a composite model with several conditionally independent sub-components. Beliefs propagated from other components are summarised as messages depicted as factor nodes on the recipient model. Being able to construct composite models of this type is a key advantage of using probabilities to represent uncertainty.

In this thesis we will often follow the following process for performing inference on probabilistic models:

1. Write down the Bayesian network corresponding to the model.
2. Write down the corresponding factor graph, including any observation or constraint factors.
3. Calculate any desired marginal distributions by message passing using (2.11), (2.12) and (2.13).

2.2.2 Loopy Belief Propagation

If the factor graph contains cycles then the separation property cannot apply for the variables in each cycle. This means that there is no schedule by which the local messages can be all updated using the sum-product algorithm in such a way that we can obtain the correct marginal distributions when the schedule terminates. However, the algorithm can still be run, with the messages repeatedly passed around each cycle. This process is called *loopy belief propagation* (MacKay, 2003) and was originally suggested by Pearl (1988). If the message densities converge to a fixed point then this is a stationary point of the Kikuchi free energy (Yedidia *et al.*, 2002) and the marginal distributions calculated using the resulting messages can be used as approximations to the true distributions. This issue will be re-visited in Chapter 5.

2.3 Parametric Message Passing

It is often useful to represent the messages in parametric form and in this section we discuss two simple representations. The Bernoulli distribution is used to represent binary messages and the Gaussian distribution is used to represent continuous messages.

2.3.1 Bernoulli Message Passing

In the example above all the variables we have considered are Boolean. If we represent these as binary variables then all the messages take on the form of a Bernoulli distribution:

$$\text{Ber}(x;q) = q^x \cdot (1-q)^{1-x},\, x \in \{0,1\}\,,\, q \in [0,1]\,, \tag{2.14}$$

the distribution over the value of a binary variable, x. A conditional binary distribution, $p(y|x)$, can be represented as:

$$p(y|x) = \left[a^y(1-a)^{1-y}\right]^x \cdot \left[b^y(1-b)^{1-y}\right]^{1-x}$$

where

$$a = p(y=1|x=1)$$

and

$$b = p(y=1|x=0).$$

Beta-Bernoulli Model

Let the prior on the parameter q be a Beta distribution:

$$p(q) = \text{Beta}(q;\alpha,\beta) \propto q^{\alpha-1} \cdot (1-q)^{\beta-1}.$$

Now the predictive distribution for x is

$$\begin{aligned} p(x=1) &= \int \text{Ber}(x=1;q)\text{Beta}(q;\alpha,\beta)dq \\ &= \frac{\int q \cdot \text{Beta}(q;\alpha,\beta)dq}{Z(\alpha,\beta)} \\ &= \frac{\alpha}{\alpha+\beta}, \end{aligned} \tag{2.15}$$

the mean of the Beta distribution.

We also have that the posterior distribution over the parameter q, given an observation of the value of x is also a Beta distribution. This property is called *conjugacy.* The Beta distribution is the conjugate prior on the parameter of the Bernoulli distribution:

$$\begin{aligned} p(q|x=X) &= \frac{p(x=X|q)p(q)}{p(x=X)} \\ &= \frac{\text{Ber}(X;q)\cdot\text{Beta}(q;\alpha,\beta)}{Z(\alpha,\beta)} \\ &= \frac{q^X\cdot(1-q)^{1-X}\cdot q^\alpha\cdot(1-q)^\beta}{Z(\alpha,\beta)} \\ &= \text{Beta}(q;\alpha+X,\beta+1-X). \end{aligned} \tag{2.16}$$

This useful property means that after making an observation of $x = X_1$ we can use the posterior $p(q|x = X_1)$ as the prior for the next observation without making inference more complex at the next stage.

2.3.2 Gaussian Message Passing

A particularly useful representation for messages in models with continuous variables is the Gaussian function:

$$m_{f\to x}(x) = \mathcal{N}(x;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\}. \tag{2.17}$$

This has a number of useful properties: firstly, the distribution over the value of a linear combination of Gaussian distributed random variables is a Gaussian, secondly the Gaussian distribution is the conjugate prior for the mean of a Gaussian likelihood and thirdly the product of Gaussian functions yields a Gaussian function. It is important to realise that the messages are not necessarily Gaussian probability distributions since the variance of a message may be negative, in which case the function cannot be normalised. When multiplying the messages together to generate marginal distributions according to (2.13) the resulting normalised density will be a Gaussian probability distribution with positive variance. A number of exact relationships hold if we assume that all messages are represented as Gaussian densities as follows.

Variable to Factor Messages

Figure 2.5 (top) shows a node, x, connected to 3 factors of the form

$$f_i(x) = \mathcal{N}(x;\mu_i,\sigma_i^2).$$

The node to factor messages are calculated by

$$\begin{aligned} m_{x\to f_1}(x) &= \prod_{i\neq 1} m_{f_i\to x}(x) \\ &= \prod_{i\neq 1}\mathcal{N}(x;\mu_i,\sigma_i^2) \\ &= \mathcal{N}(x;\mu',\sigma'^2) \end{aligned} \tag{2.18}$$

where the update equations for multiplying Gaussian densities are:

$$\sigma'^2 = \frac{1}{\sum_{i\neq 1}\sigma_i^{-2}},$$

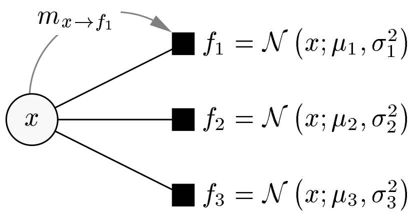
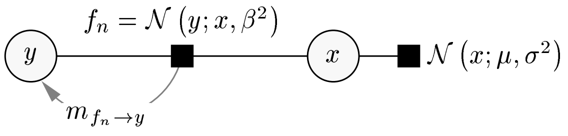
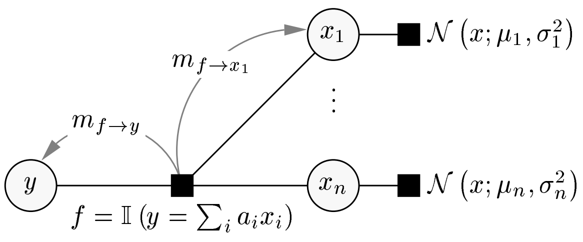
 $m_{x\to f_1}$ x $f_1 = \mathcal{N}\left(x;\mu_1,\sigma_1^2\right)$ $f_2 = \mathcal{N}\left(x;\mu_2,\sigma_2^2\right)$ $f_3 = \mathcal{N}\left(x;\mu_3,\sigma_3^2\right)$	$\begin{aligned} m_{x\to f_1}(x) &= \mathcal{N}\left(x;\mu',\sigma'^2\right) \\ \sigma'^2 &= \frac{1}{\sum_{i\neq 1}\sigma_i^{-2}} \\ \mu' &= \sigma'^2\sum_{i\neq 1}\frac{\mu_i}{\sigma_i^2} \end{aligned}$
 $f_n = \mathcal{N}\left(y;x,\beta^2\right)$ y x $\mathcal{N}\left(x;\mu,\sigma^2\right)$ $m_{f_n\to y}$	$m_{f_n\to y}(y) = \mathcal{N}\left(y;\mu,\beta^2+\sigma^2\right)$
 x_1 $\mathcal{N}\left(x;\mu_1,\sigma_1^2\right)$ $m_{f\to x_1}$ $m_{f\to y}$ y x_n $\mathcal{N}\left(x;\mu_n,\sigma_n^2\right)$ $f = \mathbb{I}\left(y=\sum_i a_i x_i\right)$	$m_{f\to y}(y) = \mathcal{N}\left(y,\sum_i a_i\mu_i,\sum_i a_i^2\sigma_i^2\right)$

Figure 2.5: Useful Gaussian message passing update equations. **Top:** Node connected to a set of Gaussian Factors. **Middle:** Gaussian Noise Factor **Bottom:** Linear Projection Factor

$$\mu' = \sigma'^2 \sum_{i \neq 1} \frac{\mu_i}{\sigma_i^2}.$$

Gaussian Noise Factor

Figure 2.5 (middle) shows a Gaussian factor connecting two variables x and y of the form

$$f_n(x, y) = \mathcal{N}\left(y; x, \beta^2\right).$$

This factor corresponds to the addition of Gaussian noise to the variable x, or alternatively f_n can be thought of as a Gaussian model with parameter $\mu = x$ for an observed variable y. The message from this factor to the variable y is calculated as

$$\begin{aligned} m_{f_n \to y}(y) &= \int f_n(x, y) \cdot m_{x \to f_n}(x) dx \\ &= \int \mathcal{N}\left(y; x, \beta^2\right) \cdot \mathcal{N}\left(x; \mu, \sigma^2\right) dx \\ &= \mathcal{N}\left(y; \mu, \beta^2 + \sigma^2\right). \end{aligned} \tag{2.19}$$

Note that the direction of the conditional distribution implied by f_n does not matter since $\mathcal{N}(y; x, \beta^2) = \mathcal{N}(x; y, \beta^2)$.

Linear Projection Factor

Figure 2.5 (bottom) shows a factor, f, which enforces the constraint that the variable y must be equal to a particular linear combination of the x_i variables:

$$f(y, x_1, \cdots, x_n) = \mathbb{I}(y = \sum_i a_i \cdot x_i).$$

The message from the factor to the variable y is calculated as:

$$\begin{aligned} m_{f \to y}(y) &= \int_{x_1} \cdots \int_{x_n} f(y, x_1, \cdots, x_n) \prod_i m_{x_i \to f}(x_i) dx_1 \cdots dx_n \\ &= \int_{x_1} \cdots \int_{x_n} \mathbb{I}(y = \sum_i a_i \cdot x_i) \prod_i \mathcal{N}\left(x_i, \mu_i, \sigma_i^2\right) dx_1 \cdots dx_n \\ &= \mathcal{N}\left(y, \sum_i a_i \mu_i, \sum_i a_i^2 \sigma_i^2\right). \end{aligned} \tag{2.20}$$

Simply re-arranging the factor and using the above gives the messages to the other variables if a Gaussian factor $\mathcal{N}(y; \mu_y, \sigma_y^2)$ is added to the graph:

$$m_{f \to x_i}(x_i) = \mathcal{N}\left(x_i; \frac{1}{a_i}\left(\mu_y - \sum_{j \neq i} a_j \mu_j\right), \frac{1}{a_i^2}\left(\sigma_y^2 + \sum_{j \neq i} a_j^2 \sigma_j^2\right)\right).$$

2.4 Expectation Propagation on Factor Graphs

2.4.1 Expectation Propagation Message Passing

We have seen that using the Gaussian density to represent messages in the sum-product algorithm has the advantage that the message updates are often simple to calculate (Section 2.3.2) and the messages are easy to represent. However, for most types of factors, the factor-to-variable messages will not be Gaussian (after calculating them using equation 2.11), leading to an intractable message-passing scheme with messages taking on complex forms which are difficult to represent, and making the required integrals non-analytic. To deal with this we use an approximation scheme called Expectation Propagation (EP) (Minka, 2001), applied as a message passing algorithm (Minka, 2005), which gives us a unified approximate inference scheme in which we represent all messages in continuous models as Gaussians. EP was first developed in order to improve the performance of Assumed-Density Filtering (ADF), which will be covered later in this chapter. EP is discussed in the context of ADF and developed from it by Minka (2001).

The equation for calculating exact factor to variable messages for continuous variables is:

$$m^{\text{true}}_{f\to v_j}(v_j) = \int_{\mathcal{V}_f\setminus v_j} f(\mathcal{V}_f)\cdot \prod_{v_i\in\mathcal{V}_f\setminus v_j} m_{v_i\to f}(v_i)d\mathcal{V}_f. \tag{2.21}$$

We also have from (2.12) and (2.13) that

$$p(v_k) \propto m^{\text{true}}_{f\to v_k}(v_k)\cdot m_{v_k\to f}(v_k). \tag{2.22}$$

This is typically non-Gaussian so we approximate it with the Gaussian $\hat{p}(v_k)$ such that the Kullback-Leibler (KL) divergence is minimised:

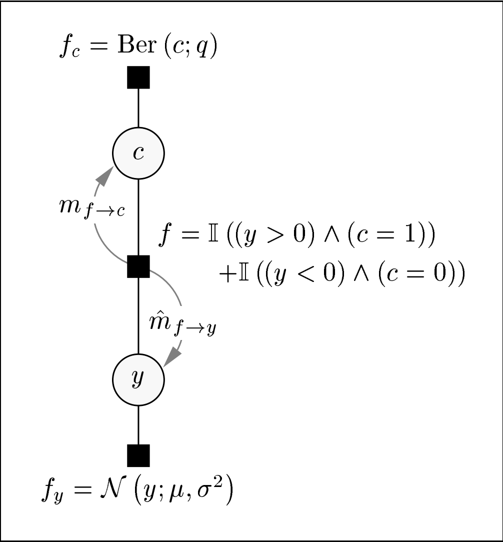
$$\text{KL}(p||\hat{p}) = \int_v p(v)\log\frac{p(v)}{\hat{p}(v)}dv.$$

For a Gaussian $\hat{p}(v)$ the KL divergence is minimised when the first and second moments of $\hat{p}(v)$ are set to the corresponding moments of $p(v)$ so we have

$$\begin{aligned}
\hat{p}(v_k) &= \text{argmin}_{\hat{p}'}\text{KL}(p||\hat{p}') \\
&= \text{MM}\left[p(v_k)\right] \\
&= \text{MM}\left[m^{\text{true}}_{f\to v_k}(v_k)\cdot m_{v_k\to f}(v_k)\right] \qquad (2.23)\\
&= \mathcal{N}(v_k;\mu_q,\sigma_q).
\end{aligned}$$

where 'MM[.]' denotes 'moment match'. Now the approximate Gaussian factor-to-variable message can be calculated by dividing out the variable-to-factor message:

$$\begin{aligned}
\hat{m}_{f\to v_k}(v_k) &\propto \frac{\hat{p}(v_k)}{m_{v_k\to f}(v_k)} \qquad (2.24)\\
&= \frac{\text{MM}\left[m^{\text{true}}_{f\to v_k}(v_k)\cdot m_{v_k\to f}(v_k)\right]}{m_{v_k\to f}(v_k)} \qquad (2.25)\\
&= \frac{\mathcal{N}(v_k;\mu_q,\sigma_q^2)}{\mathcal{N}(v_k;\mu_{v_k\to f},\sigma^2_{v_k\to f})} = \mathcal{N}\left(v_k;\mu',\sigma'^2\right), \qquad (2.26)
\end{aligned}$$


$f_c = \text{Ber}\left(c; q\right)$
c
$m_{f \to c}$
$f = \mathbb{I}\left((y > 0) \wedge (c = 1)\right)$
$+\mathbb{I}\left((y < 0) \wedge (c = 0)\right)$
$\hat{m}_{f \to y}$
y
$f_y = \mathcal{N}\left(y; \mu, \sigma^2\right)$

$$m_{f \to c}(c) = \text{Ber}\left(c; \Phi\left(0, \mu, \sigma^2\right)\right)$$

$$\hat{m}_{f \to y}(y) = \frac{\mathcal{N}(y; \mu', \sigma'^2)}{\mathcal{N}(y; \mu, \sigma)}$$

For μ' and σ'^2 see equations (2.29), (2.30) and (2.31).

Figure 2.6: Factor graph showing a coupling between a binary variable c and a continuous variable y. The message formulae are shown on the right.

where the update equations for dividing Gaussian densities are:

$$\sigma'^2 = \frac{1}{\sigma_q^{-2} - \sigma_{v_k \to f}^{-2}},$$

$$\mu' = \sigma'^2 \left(\frac{\mu_q}{\sigma_q^2} - \frac{\mu_{v_k \to f}}{\sigma_{v_k \to f}^2} \right).$$

2.4.2 Coupling Binary and Continuous Models

An important application of the moment matching method in this thesis is to couple a model of a set of binary variables with a model of a set of continuous variables. In the binary model we pass messages with the parametric form of a Bernoulli distribution and in the continuous model we represent the messages as Gaussian densities. In particular, imagine we have a Bernoulli distributed binary variable, c, and a continuous variable y. Let us place a Gaussian prior on y so $p(y) = f_y(y) = \mathcal{N}(y; \mu, \sigma^2)$. This prior could be the result of message passing on an arbitrarily complex model. We let the conditional distribution for c given y be

$$p(c|y) = \mathbb{I}((y > 0) \wedge (c = 1)) + \mathbb{I}((y < 0) \wedge (c = 0)),$$

or in other words we enforce the constraint that if $c = 1$ then y must be positive and if $c = 0$ then y must be negative. Also, we do not assume that c is observed directly (although it can be as a special case) but other observed variables are dependent on it. The evidence passed down from these observations can be summarised by the belief $f_c = \text{Ber}(c; q)$. The situation is illustrated in Figure 2.6.

We are interested in determining the messages from the coupling factor f. The message to the binary node, $m_{f\to c}(c)$ is calculated from equation 2.11 as

$$\begin{aligned}
m_{f\to c}(c) &= \int f(c,y)\cdot m_{y\to f}(y)dy \\
&= \int \mathbb{I}\left((y>0)\wedge(c=1)\right)\mathcal{N}(y;\mu,\sigma^2)dy + \int \mathbb{I}\left((y<0)\wedge(c=0)\right)\mathcal{N}(y;\mu,\sigma^2)dy \\
&= \mathbb{I}\left(c=1\right)(1-\Phi(0;\mu,\sigma^2)) + \mathbb{I}\left(c=0\right)\Phi(0;\mu,\sigma^2) \\
&= \mathrm{Ber}\left(c;(1-\Phi(0;\mu,\sigma^2))\right) \qquad (2.27)
\end{aligned}$$

The exact message to the continuous node is calculated by

$$\begin{aligned}
m_{f\to y}(y) &= \sum_c f(c,y)\cdot m_{c\to f}(c) \\
&= \sum_c f(c,y)\cdot q^c(1-q)^{1-c} \\
&= q\cdot\mathbb{I}(y>0)+(1-q)\cdot\mathbb{I}(y<0).
\end{aligned}$$

As mentioned above we wish to represent all the messages in the continuous part of the model as Gaussian densities so we must approximate this exact message by a Gaussian using the scheme described above. From 2.13 we have that

$$p(y|q)=\frac{[q\mathbb{I}(y>0)+(1-q)\mathbb{I}(y<0)]\cdot\mathcal{N}(y;\mu,\sigma^2)}{Z(\mu,\sigma,q)}.$$

We approximate this using the Gaussian distribution, $\hat{p}(y)$, with the same moments as the true marginal so

$$\begin{aligned}
\hat{p}(y) &= \mathrm{MM}\left[p(y|q)\right] \\
&= \mathcal{N}(y;\mu',\sigma'^2), \qquad (2.28)
\end{aligned}$$

where

$$\begin{aligned}
\mu' &= \frac{1}{Z}\cdot\left[\Phi\left(\frac{\mu}{\sigma}\right)(2q\mu-\mu)+\mathcal{N}\left(\frac{\mu}{\sigma}\right)(2q\sigma-\sigma)+\mu-q.\mu\right] \qquad (2.29)\\
\sigma'^2 &= \frac{1}{Z}\cdot\left[\Phi\left(\frac{\mu}{\sigma}\right)\left(\mu^2+\sigma^2\right)(2q-1)+\mathcal{N}\left(\frac{\mu}{\sigma}\right)\sigma\mu\,(2q-1)+\left(\mu^2+\sigma^2\right)(1-q)\right]-\mu'^2 \qquad (2.30)\\
Z &= (1-q)\cdot\left(1-\Phi\left(\frac{\mu}{\sigma}\right)\right)+q\cdot\Phi\left(\frac{\mu}{\sigma}\right). \qquad (2.31)
\end{aligned}$$

The derivation of this result is given in Section A.1. The approximate Gaussian message can now be calculated by dividing out the incoming message using equation 2.26:

$$\begin{aligned}
\hat{m}_{f\to y}(y) &= \frac{\hat{p}(y)}{m_{y\to f}(y)} \\
&= \frac{\mathcal{N}(y;\mu',\sigma'^2)}{\mathcal{N}(y;\mu,\sigma^2)}.
\end{aligned}$$

2.5 Assumed-Density Filtering

Now, instead of assuming all the data is available simultaneously, we look at learning on-line from one data point at a time. We assume that each data point in a data-set, $\mathcal{D}$, is independent so the likelihood factorises as a product of separate terms for each data-point. The posterior distribution over the parameters is given by Bayes' rule:

$$p(\theta|\mathcal{D}) = \frac{1}{Z} p(\theta) \prod_{d_i \in \mathcal{D}} p(d_i|\theta), \tag{2.32}$$

where θ is the set of parameters and $p(\theta)$ is the prior on the parameters. If we wish to learn on-line and update the parameters each time we observe another data-point then the update after each observation is

$$p(\theta|S_{i+1}) = \frac{1}{Z_i} p(\theta|S_i) p(d_i|\theta). \tag{2.33}$$

where $S_i = \{d_1, \cdots d_{i-1}\}$ the set of data seen up to element $i-1$, $S_i \subset \mathcal{D}$. If this equation is applied for each data-point in turn, calculating the posterior given that data-point and then using this posterior as the prior for the next point the outcome will be the same result as applying the batch update 2.32.

If we place a Gaussian prior $p(\theta)$ on the parameters and the posterior $p(\theta|d_1)$ is calculated using equation 2.33 the result will not be Gaussian for most forms of $p(d_1|\theta)$. This means that inference will be more complex for the next data point because the prior will no longer be Gaussian. However if at each step we approximate the true posterior by an assumed parametric density (for example a Gaussian distribution) then the inference procedure remains the same for each data-point and the posterior is simple to represent. The approximate distribution at each step is calculated as

$$q(\theta|S_{i+1}) = \mathrm{MM}\left[\frac{1}{Z} q(\theta|S_i) p(d_i|\theta)\right]$$

and this approximate posterior is used as the prior for the next data-point. This method is called 'Assumed-Density Filtering' (ADF) and was independently developed in the statistics, artificial intelligence and control fields (Minka, 2001). Although EP was developed to improve the performance of ADF in the case where all the data is available simultaneously ADF is still useful when we wish to learn 'online' because we do not have random access to the data.

2.6 Monte Carlo Methods

An alternative to performing approximate inference in order to solve an otherwise intractable inference problem is to obtain samples from the distribution (MacKay, 2003). If sufficient independent samples $\{\mathbf{x}_i\}_{i=1}^n \sim p(\mathbf{x})$ are obtained we can determine expectations under the distribution $p(\mathbf{x})$ by the estimator

$$\langle f(\mathbf{x}) \rangle_{p(\mathbf{x})} = \frac{1}{n} \sum_{i=1}^{n} f(\mathbf{x}_i) \tag{2.34}$$

which gives the exact result as $n \to \infty$. This is called a Monte Carlo (MC) method and we will apply this type of technique in Chapters 5 and 6. Monte Carlo methods were developed and popularised by physics researchers (for example, (Metropolis and Ulam, 1949)) and became used more commonly as digital computers became available for simulations. A famous early application of the methods was during the Manhattan Project.

Gibbs Sampling

In order to perform inference on a graphical model, a standard MC approach is Gibbs sampling. The algorithm proceeds by drawing samples from each of the variables in turn. The distribution used to draw each sample is the conditional probability distribution over the possible states of that variable, given the states of variables it depends on (connected to it via factors in the factor graph):

$$p(v = a | \mathcal{V} \setminus v) = \frac{\prod_{f \in \mathcal{F}_v} f(v = a, \mathcal{V}_f \setminus v)}{\sum_v \prod_{f \in \mathcal{F}_v} f(v, \mathcal{V}_f \setminus v)}.$$

In this way a Markov chain of samples is generated, each one of which can be used to calculate the expectation of equation 2.34. Usually it is necessary to discard the first few samples from the Markov chain as they will be dependent on the initialisation of the variables, and hence not drawn from the correct distribution (burn-in). Gibbs sampling is useful for models where EP or loopy BP would give poor performance. The graphical models covered in chapter 5 contain many tight loops so Gibbs sampling provided a better trade off between speed and accuracy than BP methods. However, for most of the models in this thesis, Expectation Propagation and message passing was sufficiently accurate and fast. A common alternative to Gibbs sampling is to use the Metropolis-Hastings method (see MacKay (2003)), although this is not applied in this thesis.

2.7 Alternative Methods

This chapter is intended as a reference for the rest of this thesis, not as a complete review of machine learning techniques. Many alternatives to the techniques presented in this chapter exist and for a comprehensive treatment I suggest reading Bishop (2006) and MacKay (2003). One alternative method which is particularly worth mentioning is the variational approximation which can also be formulated as a message passing algorithm: Variational Message Passing (Winn and Bishop, 2005). Also, it is possible to combine the different methods to perform inference in different parts of a single model.

Chapter 3

Pattern Ranking for Move Prediction[1]

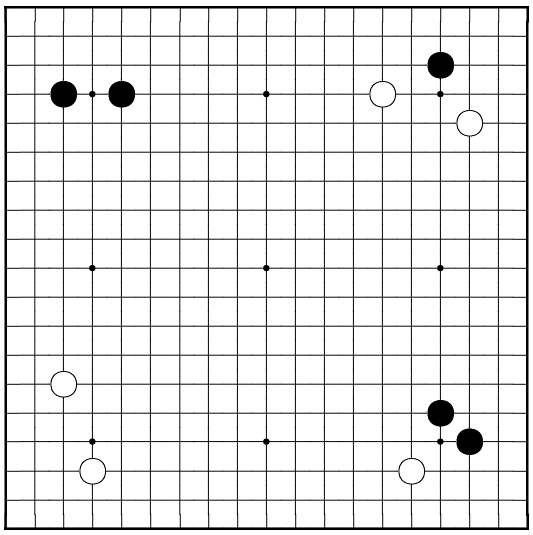


Figure 3.1: *Taisha joseki*

Faced with a board position, what is the best move to make? That is the key question a Go player must answer and here we focus directly on that issue. To be precise we are interested in obtaining a probability distribution over legal moves in a particular board position. On its own, this distribution would provide us with a standalone Go player but it could also be used as a sub-component within a Go program. For example, it could be used to prune a search tree or to pre-select moves before more computationally expensive analysis is performed. This distribution also may provide a study tool for Go.

Over the last few millennia humans have learned a great deal about playing Go. This knowledge

[1]The content of this chapter is built on work done in collaboration with Thore Graepel and Ralf Herbrich from Microsoft Research. That work was presented at the International Conference of Machine Learning 2006 (Stern *et al.*, 2006).

has been passed down through the generations by verbal and written means as well as by the dialogue implicit in play between two players. Along the way many useful strategies and guiding principles have been developed and this information is contained in abundant records of human games. In this chapter we are interested in predicting the moves in game records.

An example of the type of Go knowledge which has been developed over the years is the concept of *shape.* A shape is a local pattern of stones which is created by making a move. A 'good' shape will use stones efficiently in order to maximise the possibility of them establishing good connections, making eyes and increasing liberties while remaining flexible. Another aspect of Go play which is strongly influenced by history and is dominated by standard sequences is the opening or *fuseki.* Following the fuseki, a second set of standard sequences are often played out, called the *joseki* moves (Figure 3.1).

A lot of the information needed to select a move is contained in the configuration of stones local to the move in question. In this chapter we take advantage of this locality by matching exact patterns of stones surrounding potential moves. By matching a hierarchy of nested patterns of different sizes, the system provides a unified representation of local shapes and tactical patterns as well as standard opening plays and joseki sequences (de Groot, 2005; Stoutamire, 1991).

3.1 Pattern Matching

Computer Go has similarities to computer vision. Superficially, the goals of these fields seem well aligned: the task of a Go program is to take a board configuration (which can be represented as an array of pixels) and return a label (which location should a move be made). Similarly, in computer vision a common task is to ascribe labels to parts of images. Therefore, Schrauldolph *et al.* (1994) seemed to be on the right track when evaluating Go positions with a model previously used for character recognition (Lecun *et al.*, 1989). However, images usually have the property that changing the state of a single pixel makes little difference to the meaning of the image whereas in Go changing the state of a single vertex can completely change the value of a move. Exact pattern matching seems appropriate to capture this discontinuous nature of pattern recognition in Go and has the additional advantage that it can be achieved with low computational cost. Also, Go knowledge in reference books for players is frequently represented in terms of exact arrangements of stones in diagrams so this type of representation seems appropriate for Go.

Exact pattern matching has previously been used as a representation of various types of knowledge in computer Go programs such as connectivity between stones, group safety, territory or potential moves (Bouzy and Cazenave, 2001; Muller, 2002). A pattern is usually defined by three properties: firstly the *template,* the area within which points on the board are considered when testing for a match, secondly, additional context constraints which must be satisfied for the pattern to match such as liberty counts of chains on the boundary of a pattern and thirdly, the *pattern information* which is the knowledge which can be applied if the pattern matches, such as good move locations or group safety information (Muller, 2002). These patterns are usually hand-crafted (Boon, 1990; Fotland, 1993). Some work has gone into automatically generating the patterns from search trees (Cazenave, 1996, 2001) though more success has been achieved by extracting patterns from records of expert games (Bouzy and Chaslot, 2005; Stoutamire, 1991; de Groot, 2005).

Following Stoutamire (1991) and de Groot (2005) we define a pattern as the exact arrangement of stones within a sub-region of the board (the template) surrounding the (empty) location of a potential move. We define in advance a nested sequence of symmetrical, roughly circular templates of increasing

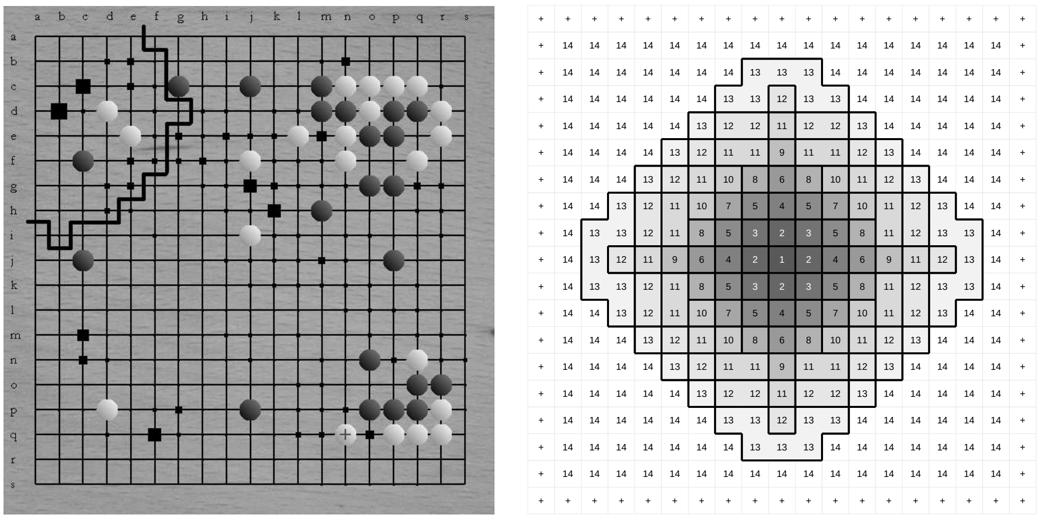


Figure 3.2: **Left:** Screen shot of the pattern system showing a board configuration from an expert game. The area of the black squares indicates for each vertex the probability of being the next black expert move under the model. In the top left corner pattern template T_{11} is shown centred about the lower 2-4 (b,d) point of that corner. **Right:** The sequence of nested pattern templates T_i with $i \in \{1, \ldots, 14\}$. Template T_{14} is a 39×39 square which covers the full board for every possible move location so it extends beyond the plot as indicated by "+". These pattern templates are similar to the pattern templates as used by de Groot (2005).

size so a large pattern can be used if a large local context has been seen previously but a small pattern can be used where necessary. Restricting ourselves to exact pattern matching within fixed, symmetrical templates means that the matching process is extremely efficient. This efficiency allows us to harness a good proportion of the vast amount of available game records (we use over a hundred thousand records of games between professional players). In addition, the low computational cost opens up a set of applications that would otherwise be impossible, such as pruning search trees and guiding Monte Carlo planning algorithms. The patterns are generated by *harvesting* them automatically from game records as described in Section 3.1.4. Given the move selection choices present in the game records we can learn values for the patterns in a separate distinct training process (see Section 3.2). As a result of this procedure the patterns can be used to predict the moves in game records.

3.1.1 Representation

In order to include the edge of the board in the representation we define an extended set of board vertices $\hat{\mathcal{N}} := \left\{\vec{v} + \vec{\delta} : \vec{v} \in \mathcal{N}, \vec{\delta} \in \mathcal{D}\right\}$ where $\mathcal{D} = \{-(N-1), \cdots, (N-1)\}^2$ is a set of vectors representing offset locations $\vec{\delta}$ from a move location $\vec{v}$. We also define an extended set of board states as $\hat{\mathcal{C}} = \{b, w, e, o\}$ (black, white, empty, off-board) generated by the function $\hat{c} : \hat{\mathcal{N}} \to \hat{\mathcal{C}}$. A pattern template, $T \subset \mathcal{D}$, is a subset of the set of offset vectors which defines the region of the board relative to a particular move location, $\vec{v}$, which is considered when determining if a pattern is present. Each pattern, $\pi : T \to \hat{\mathcal{C}}$, $\pi \in \Pi$, represents a particular colouring of the vertices of a template. For a pattern π to *match* at location $\vec{v}$, it must hold that $\hat{c}(\vec{v} + \vec{\delta}) = \pi(\vec{\delta})$ for all $\vec{\delta} \in T$.

We denote the template of size i as T_i. T_1 is the smallest template (just the vertex where the move is made) and T_{14} is the largest template (a $(2N + 1) \times (2N + 1)$ square which covers the full board for every possible move location). The pattern templates are illustrated in Figure 3.2 and have the following properties:

- The templates are a nested sequence of increasing size so for two templates T and T' we have that $T \subset T'$or $T' \subset T$ and also that $i < j \Rightarrow T_i \subset T_j$.
- The templates are rotation-symmetric and mirror-symmetric.

The meaning of a pattern of Go stones does not change if it is rotated or mirrored. In addition, if we are to use patterns to represent moves, the meaning of the pattern does not change if we reverse the colour of the move and the colour of all the stones in the pattern. Therefore we wish the pattern matcher to be invariant to these transformations. To achieve this goal we define an *invariant pattern*, $\tilde{\pi}$, as the set of patterns $\tilde{\pi} \subset \Pi$ which are equivalent under the 8-fold symmetry of the square and reversal of all the stone colours. For the colour reversal property to hold we set $\pi(\vec{0}) = b$ for patterns corresponding to black moves and $\pi(\vec{0}) = w$ for patterns corresponding to white moves. An invariant pattern, $\tilde{\pi}$, matches for a move in a position if one of its constituent non-invariant patterns $\pi \in \tilde{\pi}$ matches for that move in the same position.

3.1.2 Local Features

In order to extend the predictive power of the smaller patterns we can incorporate additional constraints which must be satisfied for a pattern to match. Guided by van der Werf *et al.* (2002) we selected the following features which provide us with 8 binary features in total:

- **Liberties of new chain (2 bits)** The number of liberties of the chain of stones produced by making the move. Values are $\{1, 2, 3, > 3\}$.
- **Liberties of opponent (2 bits)** The number of liberties of the closest opponent chain after making the move. Values are $\{1, 2, 3, > 3\}$.
- **Ko (1 bit)** Is there an active Ko?
- **Escapes atari (1 bit)** Does this move bring a chain out of atari, either by connection, extension or by capturing an opponent chain?
- **Distance to edge (2 bits)** Distance of move to the board edge. Values are $\{< 3, 4, 5, > 5\}$.

We define an index over the set of features as $F = \{1, \cdots, 8\}$. The function $f : F \rightarrow \{\text{TRUE}, \text{FALSE}\}$ maps each feature to its logical value. For the larger patterns these features are already seen in the arrangement of stones within the template region so the larger patterns are less likely to be altered by the addition of these features.

3.1.3 Pattern Matching and Storing

The patterns are not represented explicitly but instead we define a hash function for patterns and store the pattern information in a hash table. This saves memory and decreases the computational cost of pattern matching. The hash function is a variant of Zobrist hashing (Zobrist, 1990). We generate four sets of 64-bit random numbers, $h_a : \mathcal{D} \rightarrow \{0, \cdots, 2^{64} - 1\}$, $a \in \hat{\mathcal{C}}$, so there are four numbers for each possible offset vector from the pattern center. The hash-key k_π of a pattern π is calculated by the XOR ($\oplus$) of the corresponding random numbers:

$$k_\pi := \bigoplus_{\vec{\delta}\in T} h_{\pi(\vec{\delta})}(\vec{\delta}).$$

Both adding and removing a stone of colour $a \in \{b.w\}$ at position $\vec{\delta}$ correspond to the same operation $k_{\text{new}} \leftarrow k_{\text{old}} \oplus h_a(\vec{\delta}) \oplus h_e(\vec{\delta})$. Due to the associativity and commutativity of the XOR operation the hash-key can be calculated incrementally as stones are added or removed from a pattern and the order in which stones are added does not matter.

As mentioned in Section 3.1.1 we wish to match patterns which are invariant to the 8-fold symmetry of the square and to colour reversal so we actually store the invariant patterns $\tilde{\pi}$. In order to calculate the hash-key for an invariant pattern, $\tilde{\pi}$, we calculate the hash-key for every symmetry variant of the pattern and then choose the minimum of those keys, that is, $k_{\tilde{\pi}} := \min_{\pi\in\tilde{\pi}} k_\pi$.

We also generate a random number for each of the features (see Section 3.1.2), $l : F \rightarrow \left\{0, \cdots, 2^{64}-1\right\}$. The hash key for the feature configuration is given by the XOR of the keys for the active (TRUE) features:

$$k_f = \bigoplus_{i\in F} l(i)\mathbb{I}\left(f(i)\right).$$

Therefore, the hash-key of the complete pattern, including feature constraints, is generated by:

$$k_{\tilde{\pi},f} := k_{\tilde{\pi}} \oplus k_f.$$

3.1.4 Harvesting

The hash-table of patterns is filled by automatically harvesting invariant patterns (for all the template sizes) corresponding to the moves played in a database of Go game records. A game has about 250 moves on average and there are 14 different pattern sizes so a database of 180,000 game records yields roughly $180,000 \times 14 \times 250 = 600,000,000$ potential patterns. In order to reduce storage and to ensure that we store only significant patterns we retain only the patterns which appear in the game records more than b times where $b = 2$ for the full database and $b = 1$ if we use only a subset of the database. Testing for this condition is potentially difficult as storing 600 million 64-bit hash keys in memory is impossible. In order to get around this difficulty we use a *spectral Bloom filter* (Cohen and Matias, 2003) to count the number of times each pattern has been seen as we play through the game records.

A spectral Bloom filter is a generalisation of the Bloom filter (Bloom, 1970) and gives an approximate test for the minimum number of times each element of a set has been seen previously with minimal memory footprint. The spectral Bloom filter contains of a array of m integers, each entry of which is indexed by a key, $k \in \{1, \cdots, m\}$. Let the value of each of the entries in the array be given by the function $B : k \rightarrow \{0, \mathbb{Z}^+\}$ (initialised to zero for an empty Bloom filter). For each item to be added to the Bloom filter we generate a set of hash keys, K, and to insert the item we increment each of the corresponding entries in the array, that is, $B(k) := B(k) + 1, \forall k \in K$. To test the number of times an item has been previously inserted we calculate

$$t = \min_{k\in K} B(k).$$

If we use $t > b$ as a success criterion (that is we wish to test if an item has been inserted more than

b times) then the false positive rate is the same as that for the standard Bloom filter and can be estimated as $(1 - \exp(-\frac{|K|n}{m}))^{|K|}$ (Cohen and Matias, 2003). We choose $|K| = 4$, and $m = 2^{32}$ which gives us an error rate of 3% for $n = 600 \times 10^6$ insertions. In practice, the false-positive rate seems far lower than this, possibly because the figure $n = 600$ million is an over-estimate. The false negative rate is zero so although we may end up with some unnecessary, rarely seen patterns in the hash-table we will not eliminate any important, frequently seen patterns. The hash-keys are generated as described in Section 3.1.3.

3.2 Move Ranking Models

3.2.1 Bayesian Ranking Model

The model for ranking the patterns is based on a model for ranking players after observing a sequence of game outcomes (Herbrich *et al.*, 2007). Each position in a game record contains a set of available legal moves, only one of which was chosen by the player whose turn it was to move. We assume that each move in the set of legal moves, $\vec{v} \in \mathcal{L}(c)$, in a position, c, has some 'urgency value', $u(\vec{v}, c)$, which is defined such that a player will always choose the move with the highest urgency value. If we denote the vector of move urgencies by $\mathbf{u}$, the probability distribution over move locations for a position, c, is given by

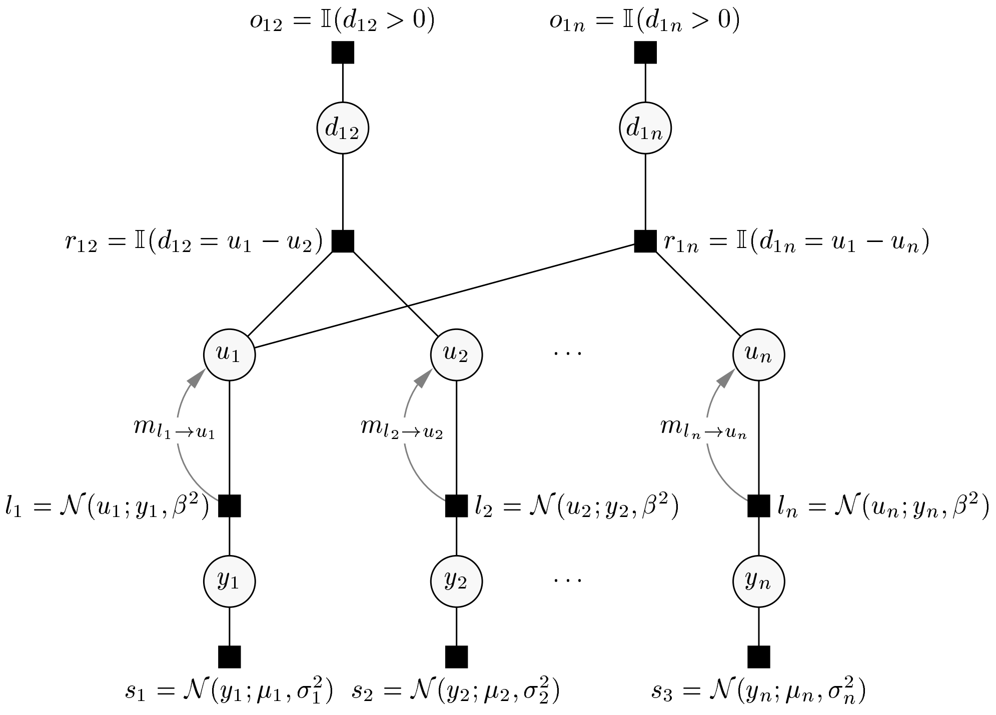
$$P(\vec{v}|c, \mathbf{u}) := P\left(\vec{v} = \underset{\vec{v}' \in \mathcal{L}(c)}{\operatorname{argmax}} \{u(\vec{v}', c)\}\right). \tag{3.1}$$

We model each move's urgency, $u(\vec{v}, c)$, as a Gaussian distribution with an underlying latent mean value, $y(\vec{v}, c)$, and fixed variance, β^2, that is, $p(u|y, \beta^2) = \mathcal{N}(u; y, \beta^2)$. We use $\mathbf{y}$ to denote the vector of the underlying move mean values. We place a separate Gaussian prior on each move's underlying value, $p(y(\vec{v}, c)) = \mathcal{N}(y(\vec{v}, c); \mu_{\vec{v},c}, \sigma^2_{\vec{v},c})$. If the parameters of this Gaussian prior model are shared between moves with the same characteristics in different positions then it is possible to generalise and hence to learn to predict moves in future, unseen positions, from positions in the training data. The simplest way we do this is to make the value of a move depend on the largest matching pattern in the hash table. We denote the largest matching pattern for a move at location $\vec{v}$ in position c as $\pi(\vec{v}, c)$ so, $y(\vec{v}, c) := y(\pi(\vec{v}, c))$. This is the method we use for the first experiments. The noise term with variance β^2 takes into account variability due to the prior model on $y(\vec{v}, c)$ not taking into account all aspects of the board position (because the largest pattern will usually not be the full board pattern) as well as the fact that different Go players have different playing styles so may prefer different moves in the same context.

Inference

Equation 3.1 shows that if a move is played it must have higher urgency than every other available move. This means that observing a move being played in a position is equivalent to observing the set of constraints enforced by the fact that the urgency of the played move must be higher than the urgencies of each of the other legal moves. Figure 3.3 shows the factor graph corresponding to this situation, including the observed constraints. Each element of $\mathbf{u}$ is denoted u_i and each element of $\mathbf{y}$ is denoted y_i where i is an index over the available legal moves in a position.

The goal of learning is to infer a posterior distribution over the move values, $\mathbf{y}$. For a single board position and chosen move the posterior, $p(\mathbf{y}|\vec{v}, c)$, can be calculated by message passing ac-



$o_{12} = \mathbb{I}(d_{12} > 0)$
$o_{1n} = \mathbb{I}(d_{1n} > 0)$
d_{12}
d_{1n}
$r_{12} = \mathbb{I}(d_{12} = u_1 - u_2)$
$r_{1n} = \mathbb{I}(d_{1n} = u_1 - u_n)$
u_1
u_2
$\cdots$
u_n
$m_{l_1 \to u_1}$
$m_{l_2 \to u_2}$
$m_{l_n \to u_n}$
$l_1 = \mathcal{N}(u_1; y_1, \beta^2)$
$l_2 = \mathcal{N}(u_2; y_2, \beta^2)$
$l_n = \mathcal{N}(u_n; y_n, \beta^2)$
y_1
y_2
$\cdots$
y_n
$s_1 = \mathcal{N}(y_1; \mu_1, \sigma_1^2)$
$s_2 = \mathcal{N}(y_2; \mu_2, \sigma_2^2)$
$s_3 = \mathcal{N}(y_n; \mu_n, \sigma_n^2)$

Figure 3.3: Factor graph corresponding to the joint distribution of $\mathbf{u}$ and $\mathbf{y}$ for a single Go position. The move with index 1 is chosen. This is equivalent to observing the set of constraints such that move 1 has a higher urgency, u_1, than the other available moves.

cording to the sum product algorithm (see Section 2.2.1). The messages from the factors, o_{1i} (the sign constraints), to the difference variables, r_{1i}, are non-Gaussian step functions (equal to the factors o_{1i} themselves). These are approximated by Gaussian densities using the EP scheme described in Section 2.4 and the message update is given by equation 2.28 for the special case where $q = 1$. All the other factor-to-variable messages on the graph are exactly Gaussian (assuming Gaussian input messages) and the update equations are given in Figure 2.5. Since the messages from the factors o_{1i} are approximations which depend on the upward messages into the factors, which in turn depend on (approximate) messages from the other o_{1j} factors, we must iterate the message passing in the part of the graph above the u_i variables until convergence before passing the messages down to the y_i variables. To summarize, we first calculate the upward (exact) messages $m_{l_i \to u_i}$ using equation (2.19). Next we iterate messages passing in the top part of the graph. Then we calculate the messages $m_{u_i \to l_i}$ and the messages $m_{l_i \to y_i}$ so we can finally update the posterior from equation 2.13 as

$$p(y_i|\vec{v}, c) = \frac{1}{Z} m_{s_i \to y_i} \cdot m_{l_i \to y_i}.$$

This posterior is obviously only an approximation to the true posterior because the message $m_{o_{1i} \to r_{1i}}$ is only approximately Gaussian. For the next move/position pair in the dataset we use this posterior as the prior on the latent move values, y_i, and perform the same message passing procedure for the next position. Therefore we are performing online learning across the set of game positions using Assumed Density Filtering (ADF) (Section 2.5).

3.2.2 Independent Bernoulli Model

As a simpler alternative model, we assume that each move has a certain independent probability of being played, regardless whether another move is already played in that position. Therefore the

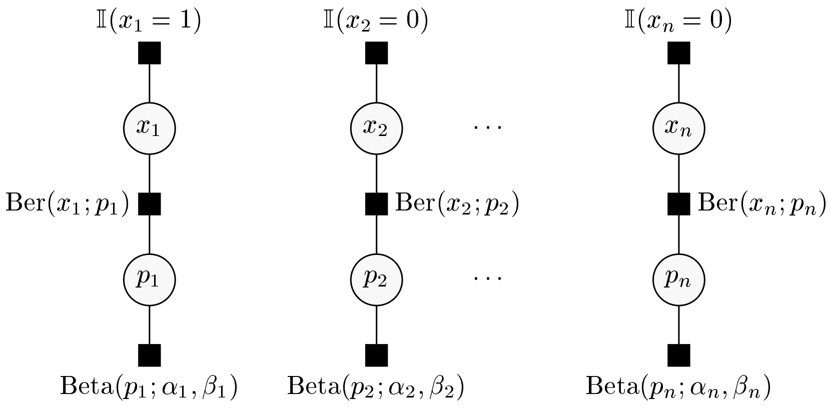


Figure 3.4: Independent Bernoulli Model.

probability of the location of a move being $\vec{v}$ in position c is

$$p(\vec{v}|c, \mathbf{p}) = p_{\vec{v},c} \cdot \prod_{\vec{v}' \in \mathcal{L}(c) \setminus \vec{v}} (1 - p_{\vec{v}',c}).$$

where $\mathbf{p}$ is a vector of probabilities, $p_{\vec{v},c}$. Again, the simplest way to map a move to a probability (the element of $\mathbf{p}$) is by the largest pattern which matches for the move, that is, $p_{\vec{v},c} = p_{\pi(\vec{v},c)}$. This is the method we adopt for our experiments. The uncertainty on the probability $p_{\vec{v},c}$ is modeled by a conjugate Beta prior $p(p_{\vec{v},c}) = \mathrm{Beta}(p_{\vec{v},c}; \alpha_{\vec{v},c}, \beta_{\vec{v},c})$ (see Section 2.3.1) so the marginal probability of a move is

$$p(\vec{v}|c, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \int p(\vec{v}|c, \mathbf{p}) \prod_{\vec{v}' \in \mathcal{L}(c)} \mathrm{Beta}(p_{\vec{v}',c}; \alpha_{\vec{v}',c}, \beta_{\vec{v}',c}) d\mathbf{p} \tag{3.2}$$

$$= \frac{\alpha_{\vec{v},c}}{\alpha_{\vec{v},c} + \beta_{\vec{v},c}} \cdot \prod_{\vec{v}' \in \mathcal{L}(c) \setminus \vec{v}} \left(1 - \frac{\alpha_{\vec{v}',c}}{\alpha_{\vec{v}',c} + \beta_{\vec{v}',c}}\right) \tag{3.3}$$

If we map to the parameters via the largest pattern match then $\alpha_{\vec{v},c}$ is equal to the number of times the corresponding pattern matched in the training data for moves played by the human players and $\beta_{\vec{v},c}$ corresponds to the number of times the corresponding pattern matched for moves which were available to be played in the training games but were not chosen. Thus learning in this model involves simple counting. For comparison the factor graph for this model for the same example situation as we used for the ranking model is shown in Figure 3.4.

This model does not produce a normalised distribution over the moves in a position and valid samples from this distribution may contain zero or indeed multiple moves per position. This means that the probability of the expert moves in the test data as calculated using equation (3.2) is lower for this model (as we will see in Section 3.2.3) than for the full ranking model. The move prediction probability could be improved by re-normalising the probabilities produced by equation 3.2 per position.

3.2.3 Experiments

In order to evaluate the pattern representation and compare the ranking models we carried out the following set of experiments.

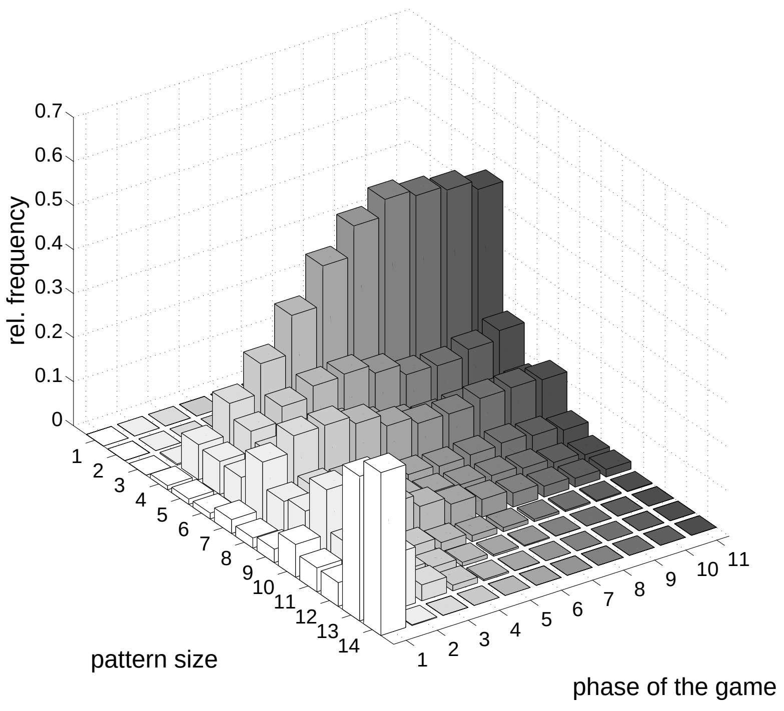


Figure 3.5: Relative frequency of the different pattern sizes matching as a function of game phase. Each game phase is 30 moves. These patterns were harvested from 181,000 Go games with the Bloom filter cut-off set to $b = 2$.

Harvesting

Patterns were harvested from a training set of 181,000 Go games between professional Go players. We retained all patterns seen 3 or more times so the Bloom filter cut-off was set to $b = 2$. This gives a table of 12 million patterns. Figure 3.5 shows the relative frequency of the different pattern sizes being matched at each phase of the game. In this plot (and in the following results where the term is used) each game 'phase' is 30 moves. Large patterns match at the beginning of the game when the board is mostly empty and play is proceeding according to standard lines of play (the *fuseki* and *joseki*) whereas later in the game only smaller patterns match.

Pattern Ranking

Next, values were learned for each pattern using the models described above. Each move was represented by the largest pattern found in the hash table matching at the move location, that is, parameters were shared between moves with the same largest matching pattern. Since we harvest only patterns seen sufficient times in the training games as described in Section 3.1.4, using the biggest matching pattern to represent a move is equivalent to the 'backing off' scheme well known to the language modelling community (Katz, 1997). In such a scheme smaller and smaller contexts are considered until the context is sufficiently small that it has been seen a significant number of times in training so we can use its value with confidence.

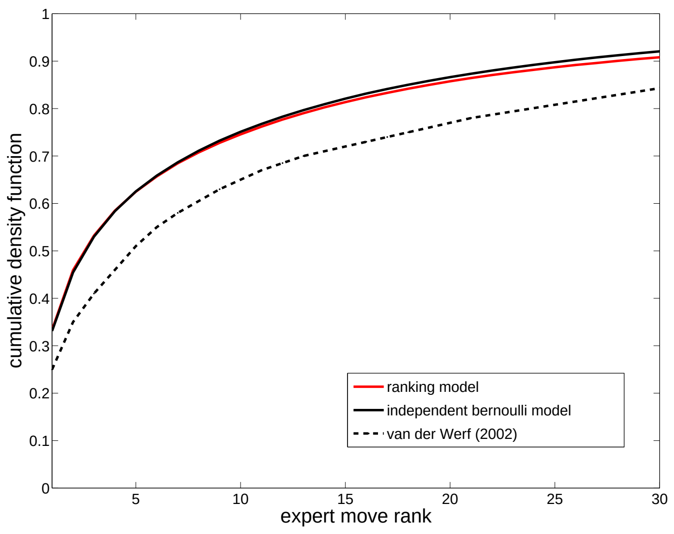
For the full ranking model the prior parameters were set to $\mu = 0$ and $\sigma = 1$ for all of the patterns. For the independent Bernoulli model the prior parameters were set to $\alpha = \beta = 1$ (corresponding to two initial pseudo-observations of the pattern appearing, one in which it was played and one in which it was not played.)

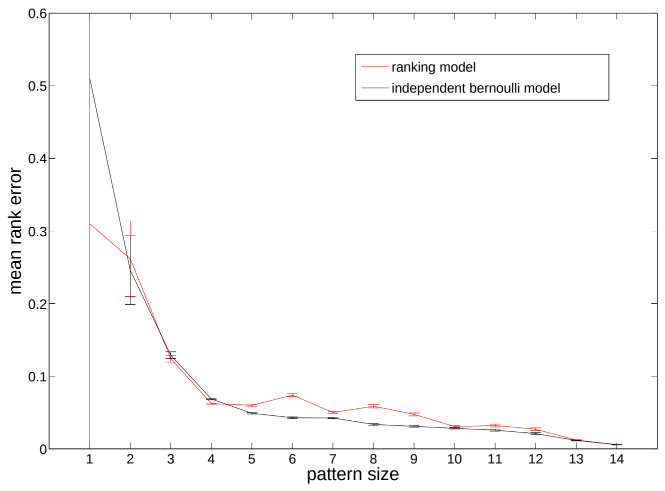
For the testing we ranked all the moves played in 1352 separate expert games by matching the largest pattern for every possible move and ranking the moves according to the μ values in the case of the full ranking model and the mean probabilities, $\frac{\alpha}{\alpha+\beta}$, in the case of the independent Bernoulli model. Figure 3.6 (top) shows that the full ranking model ranks 34% of all expert moves first, 66% in the top 5, 76% in the top 10 and 86% in the top 20. The graph illustrates that we have significantly improved on the performance of van der Werf *et al.* (2002) who used a neural network model, taking a number of hand crafted features as the input. Expert human players will often disagree about which is the best move on the board so a possible direction for future work would be to evaluate human performance at this task since this result would yield a useful benchmark for our models.

Three Ranking Scenarios

Figure 3.6 also shows that the two move ranking models appear to perform similarly. This is perhaps surprising as the full ranking model should be able infer a ranking from more limited observations than the independent model. Considering three hypothetical scenarios which might occur in training as presented in Figure 3.9 provides a possible explanation of the similar performance of the two models. Assume we wish to infer a global ranking of four patterns from limited observations of pattern pairs where the true values are ordered such that $A > B > C > D$. Each factor in the figure represents an observation of the ordering of a pair of patterns.

The first graph shows a situation where we observe three preference pairs in a linked chain: $A > B$, $B > C$, and then $C > D$. The full ranking model could infer a global ranking from





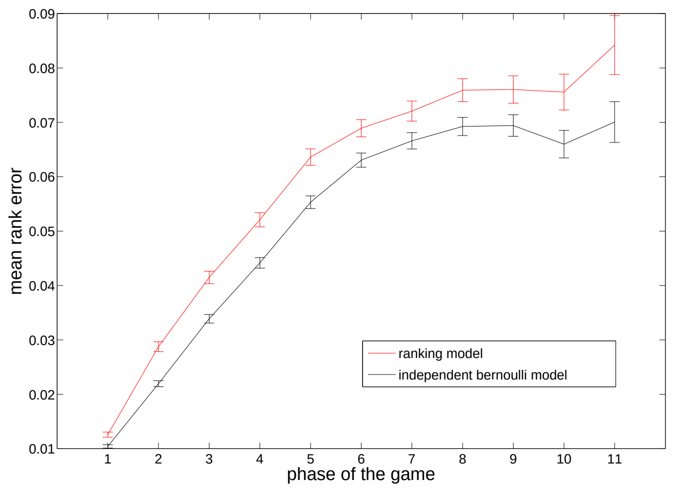


Figure 3.6: Comparing the performance of the two ranking models on 1352 unseen expert games. The error bars indicate +/− one standard deviation. **Top**: Cumulative distribution of the ranks the models assign to the moves played by expert players. For comparison the corresponding curve from van der Werf *et al.* (2002) is included which was obtained on games from the same collection. **Middle**: Mean rank error for each pattern size. Rank error is $\frac{\text{expert move rank}}{\text{number available moves}}$. **Bottom**: Mean rank error for different phases of the game (each phase is 30 moves).

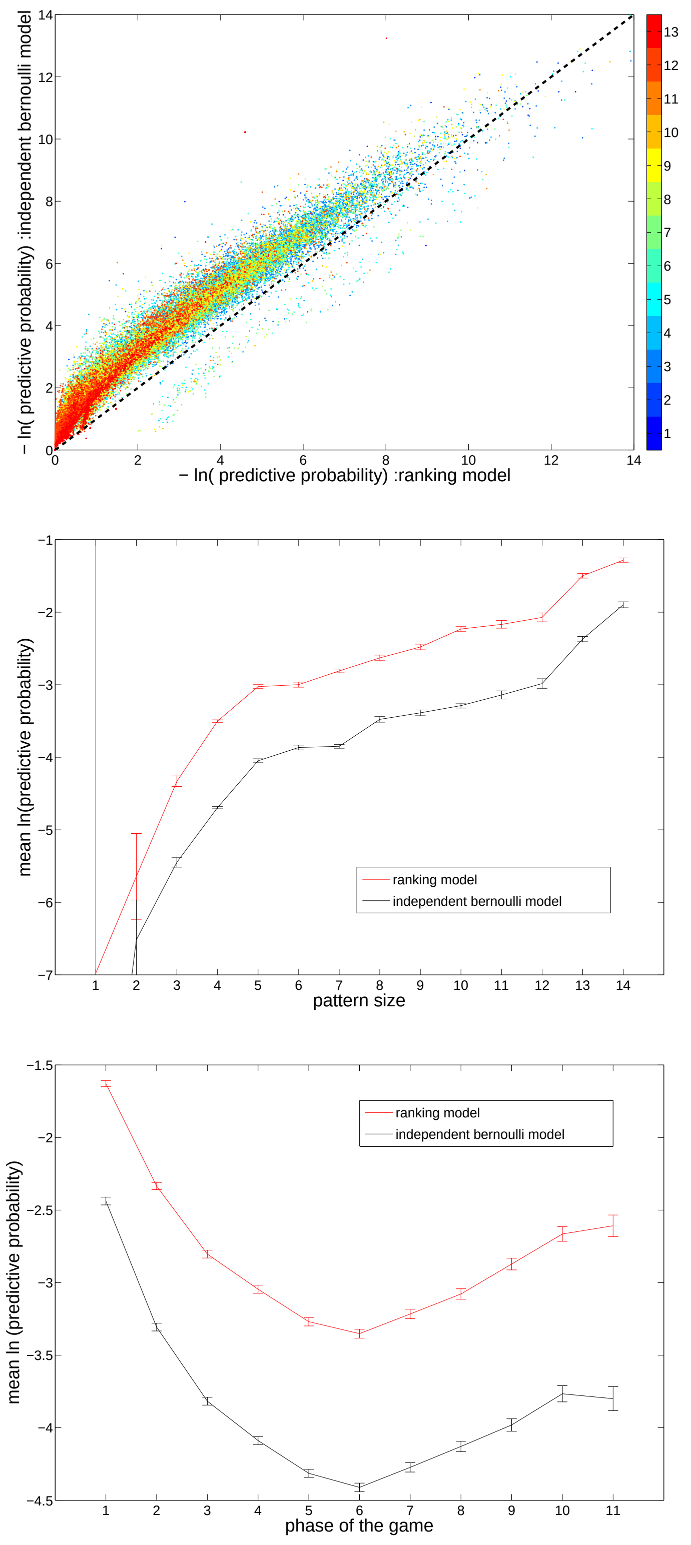
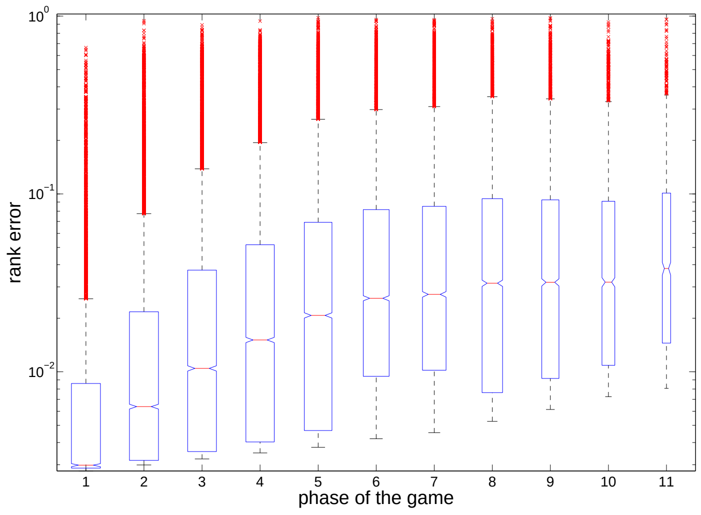


Figure 3.7: Comparing the probability of the expert moves in 1352 unseen expert games according to the two ranking models. **Top**: Scatter Plot. Each colour corresponds to a different maximum pattern size. **Middle**: Mean predictive probability for each pattern size. **Bottom**: Mean predictive probability for different phases of the game (each phase is 30 moves).



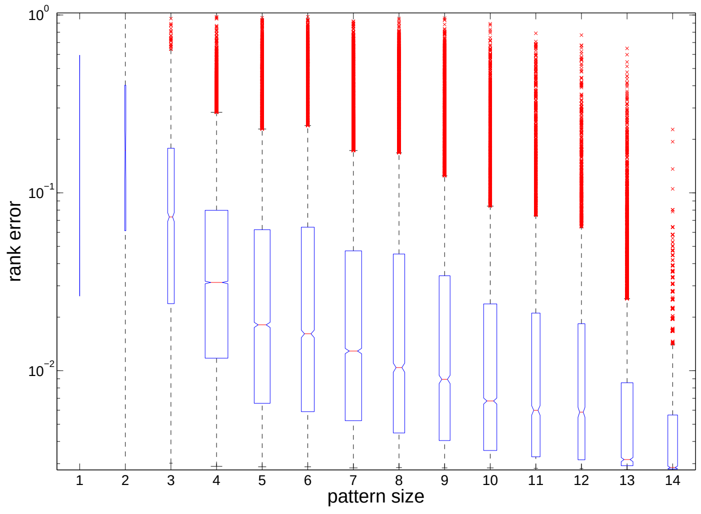


Figure 3.8: Box plots showing test performance of the full ranking model on 1352 unseen games. Lower and upper sides of box: quartiles; horizontal line across middle: median; width: number of data points; whiskers (dashed lines): extent of data within $3 \times$ Inter-Quartile Range (IQR); dots: data points outside $3 \times$ IQR. **Top:** Box plot of the rank error for different phases of the game, each phase corresponding to an interval of 30 moves. Rank error is $\frac{\text{expert move rank}}{\text{number available moves}}$. **Bottom:** Box plot of the rank error for different pattern sizes.

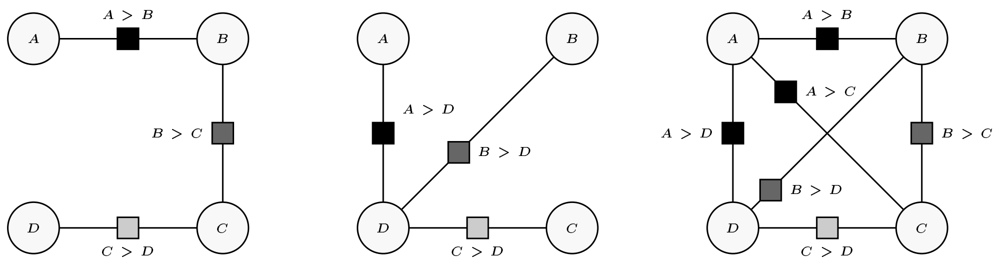


Figure 3.9: Each factor graph represents a set of observations of the ordering of pairs of variables. The global preference order of these variables is $A > B > C > D$ and the goal of a ranking algorithm is to infer this order from the observations.

these observations. In contrast, the independent Bernoulli model could only learn the probabilities $\{A, B, C, D\} = \left\{\frac{1}{1}, \frac{1}{2}, \frac{1}{2}, \frac{0}{1}\right\}$ where each fraction is (times played)/(times seen).

In the second graph we observe three preference pairs: $A > D$, $B > D$, and $C > D$. Neither ranking model can learn a global ranking from these three observations. This set of observations is typical of what is observed during a *fuseki* or *joseki* sequence in Go. During such a sequence the patterns in areas of the board not involved in the joseki will remain unchanged as no new stones will be placed in these areas of the board (corresponding to the pattern D in the example). However as each move in the joseki sequence is added to the board the pattern for that move disappears and by the placement of the new stone the next pattern in the sequence is created. The joseki move-patterns correspond to A, B, and C in the example. Despite the fact that a global ordering of the joseki patterns cannot be inferred (as they are unlikely to co-occur in a single position in the training games) the system can learn the joseki sequence by merely learning that all joseki patterns have high value - the ordering of the moves follows automatically from the rules of the game.

The third graph shows a situation in which we observe a preference ordering for all pairs of patterns. In that case both of the ranking models can learn a global ranking of the patterns with the independent Bernoulli model learning $\{A, B, C, D\} = \left\{\frac{3}{3}, \frac{2}{3}, \frac{1}{3}, \frac{0}{3}\right\}$. This scenario is typical of an end game situation in Go where many competing move-patterns co-occur on the board.

So in summary it is only the first situation (the chain of preference pairs) where the full ranking model can learn a better ordering of moves than the independent model. However, this situation does not correspond (at least not in any obvious way) to a real scenario likely to be seen in Go games so this might explain the similar performance of the two models in terms of move ordering.

Despite this similarity in performance, the full ranking model has the advantage that it provides a normalised probability distribution over moves in a given position whereas the independent model does not because it lacks the constraint that only exactly one move-pattern is played in a given position as shown by the two factor graphs (Figures 3.3 and 3.4). Since the full ranking model concentrates its probability mass on only the possible set of outcomes, we would expect the model evidence, $p(\vec{v}|c)$, to be greater, which indeed is the case. Figure 3.7 compares the probability of the expert moves in the test set according to the two ranking models. The full ranking model outperforms the independent model at every stage of the game and for all pattern sizes. It is worth noting, however, that the move probabilities produced by the Bernoulli model could be renormalised for each position which would improve the performance of this model in terms of move probability. Both models appear to perform

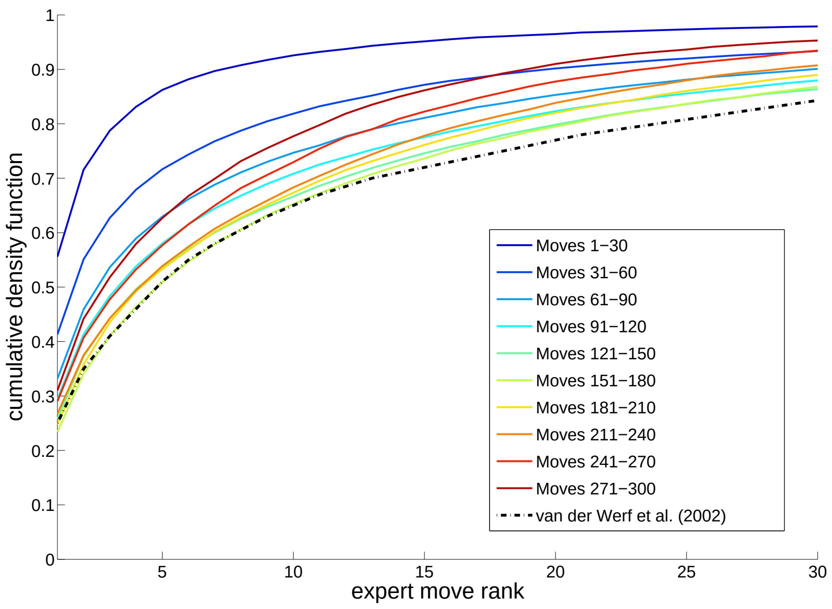


Figure 3.10: Cumulative distribution of the ranks the models assign to the moves played by expert players. A separate plot is shown for each phase of the game. For comparison the corresponding curve from van der Werf *et al.* (2002) is included which was obtained on games from the same collection.

best at the start of the game and at the end of the game. The increase in performance at the end of the game could be explained by the fact that at the end-game play starts to become dominated by standard plays, characterised by standard patterns, and hence becomes more predictable. The figure also demonstrates that larger patterns yield better performance.

Because the full ranking model gives a normalised probability distribution over moves and hence predicts the actual expert moves with a higher probability, I did not consider the independent Bernoulli model further. Another reason for this choice is aesthetic: the full ranking model seems the 'correct' model for learning the underlying values of moves, based on an observed partial ranking and seems a more principled approach than the bernoulli counting model. After all, it is a ranking we observe, so a ranking model seems appropriate. Also, learning a ranking of moves is exactly what is needed if we are to use the system to prune the huge Go game tree, which is ultimately what we are interested in.

Game Phase

The ranking performance at different stages of the game is compared in Figures 3.8 (top) and 3.6 (bottom). The system performs extremely well at the early stages of the game where moves more commonly correspond to standard plays. Figure 3.8 (bottom), Figure 3.6 (middle) and Figure 3.5 provide an explanation for this: the system is more likely to match larger patterns with greater discriminative power earlier in the game. For the first 30 moves the system frequently matches full board patterns (size 14) which correspond to the standard fuseki opening moves and for the next 30 moves large patterns are still often matched (size 12 and 13) which correspond to the standard joseki corner plays. The combination of matching large patterns and the systematic assignment of their values means that the system reliably predicts entire joseki sequences. Later in the game only smaller, less discriminative, patterns are matched so the move prediction performance decreases. However, the fact that the system gradually automatically backs off to smaller patterns as the game progresses means that the performance degradation is graceful.

Figure 3.10 shows the cumulative density function of the rank of the expert move according to the full ranking model for different stages of the game, again showing that performance is much stronger at the start of the game. However, performance never falls below that of the neural network model of van der Werf *et al.* (2002).

Effect of Training Set Size

Figures 3.11 and 3.12 show the effect of changing the number of training games on performance. There is a large performance difference, presumably because larger patterns are found more frequently if the set of patterns is larger. This suggests that better results might be obtained if an even larger training set was used. There is a great deal more data available from the online Go servers than we have made use of so far.

Example Self-Play Game

I also tested the ability of the system to play Go and an example game of the system against itself is shown in Figures 3.13 and 3.14. Each move is selected by matching the largest pattern in the pattern table for each available legal move and then selecting the move with the largest mean pattern value. The system appears to play along standard *fuseki* and *joseki* lines during the early part of the game (up to move 32).

The system takes about 10ms to generate each move on a standard 2GHz PC. This speed can be increased to the order of 1ms by restricting the size of the largest pattern (as in the next chapter).

Games Against Human Opponents

Move prediction and actual game play are quite different challenges. It seems likely that a model could perform extremely well at the task of move prediction (in terms of fraction of moves correctly predicted) while being an extremely poor Go player. Imagine, for example, the case where every 10th move of a perfect expert move predictor was replaced by a random move: such a move predictor would predict 90% of expert moves perfectly but it would probably be a poor Go player because the random moves would leave urgent threats un-answered. Interestingly, we have (albeit rather limited) evidence that the pattern system described in this chapter can actually provide a reasonable (and fun) opponent for both weak and strong Go players.

Several human players played against the pattern system. Go players are rated according to a standard system of *kyu* and *dan* values where *kyu* ratings (lower is better) are for student players (like the coloured belts in martial arts) and the *dan* ratings (higher is better) are for serious players (like the black belt). A Go player with a strong rating of 2 *dan* estimated the pattern system should be rated about 8 kyu (a weak but not beginner rank, comparable to the Go program GnuGo). The system was able to play each move in about 10ms on a standard 2GHz PC so if the thinking time of the human player was restricted the computer would have a considerable advantage. This speed also means that the computer could, in principle, play hundreds of human opponents in parallel.

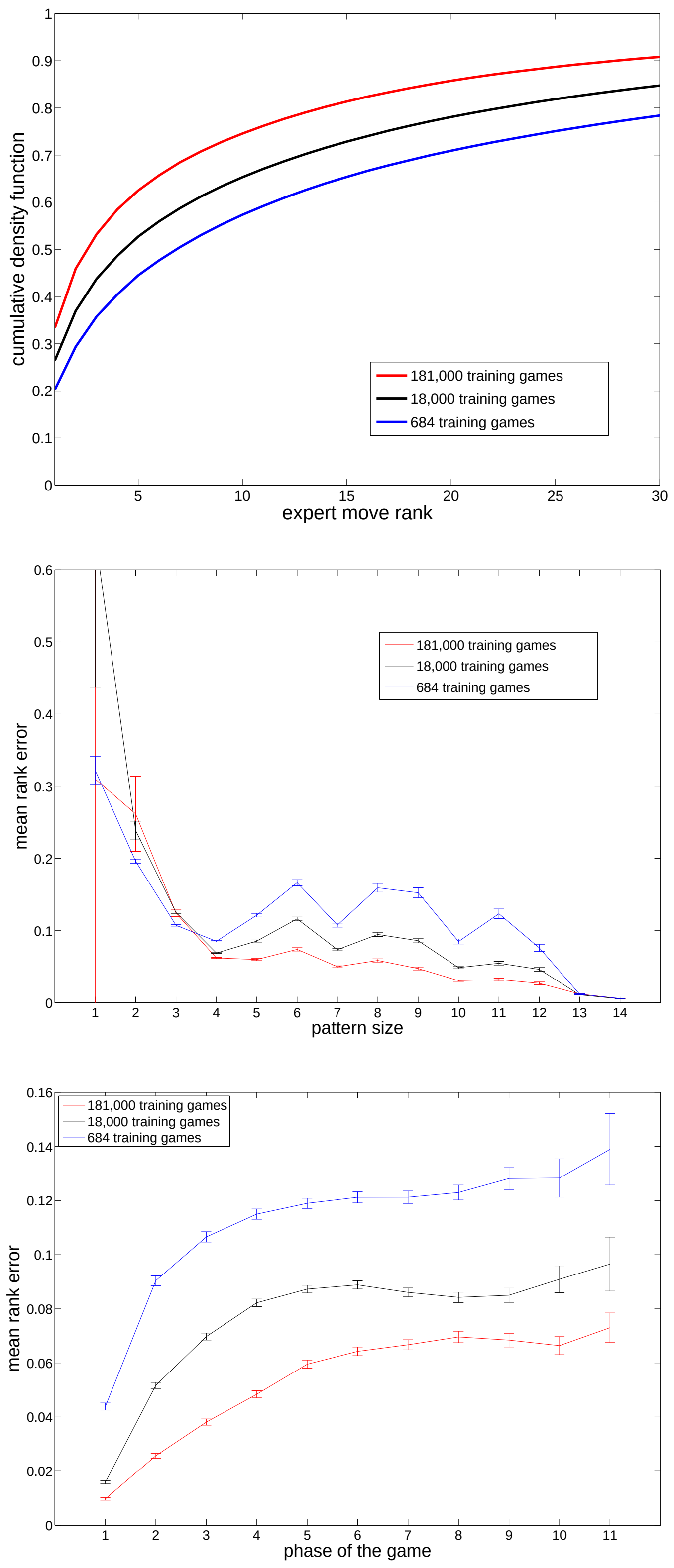
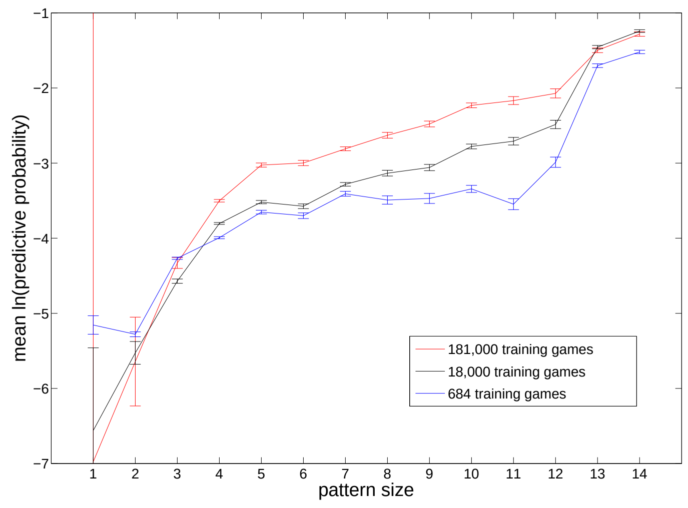


Figure 3.11: Comparing the performance of the full ranking model for different sizes of training data sets. **Top**: Cumulative distribution of the ranks the models assign to the moves played by expert players. **Middle**: Mean rank error for each pattern size. Rank error is $\frac{\text{expert move rank}}{\text{number available moves}}$. **Bottom**: Mean rank error for different phases of the game (each phase is 30 moves).



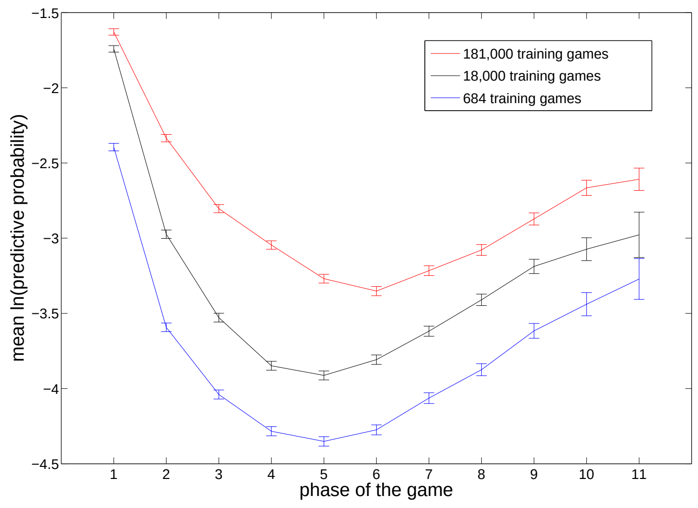


Figure 3.12: Comparing the probability of the expert moves in 1352 unseen expert games for different sizes of training dataset. **Top**: Mean predictive probability for each pattern size. **Bottom**: Mean predictive probability for different phases of the game (each phase is 30 moves).

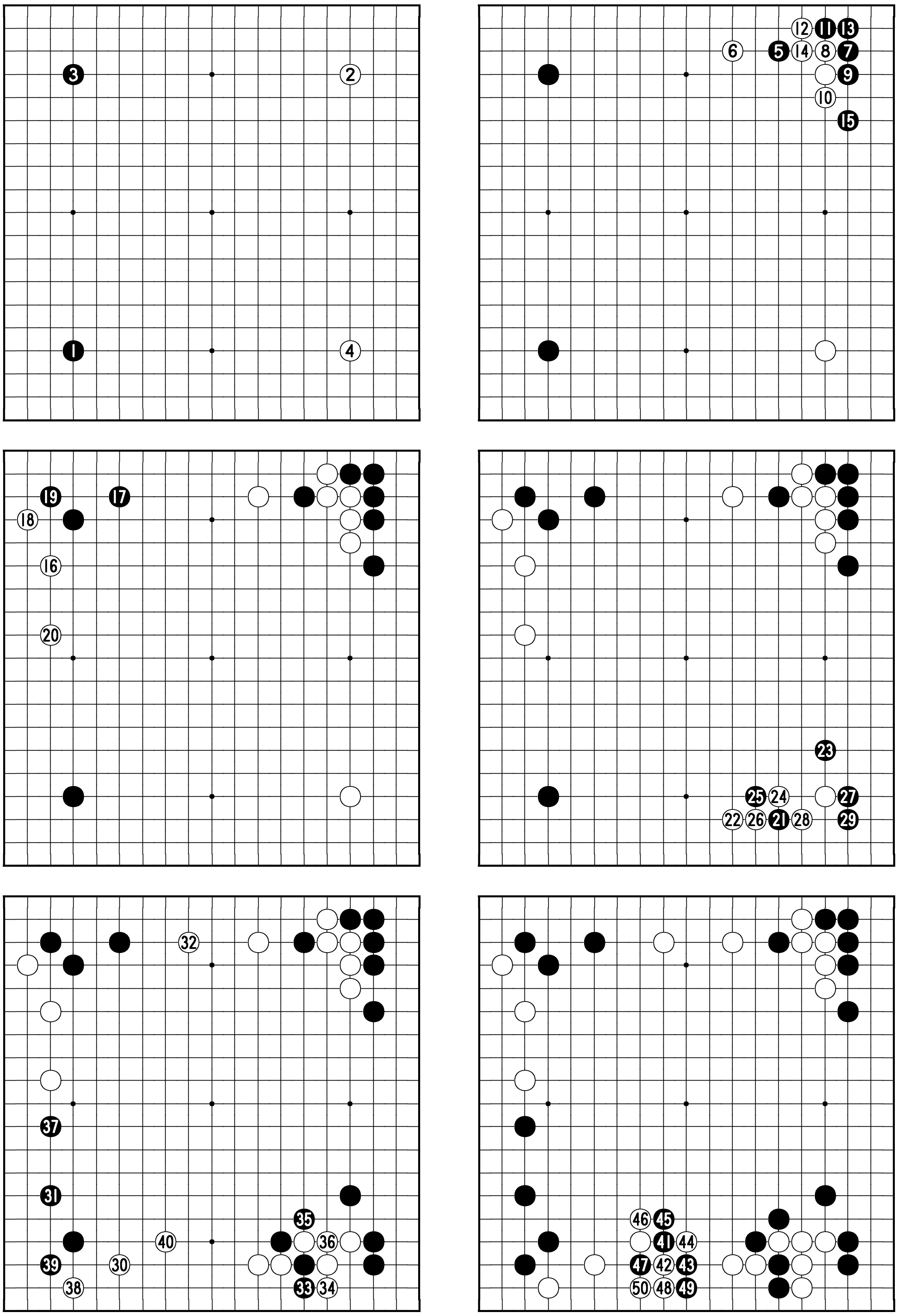


Figure 3.13: Diagram of the the start of a match of the Bayesian pattern ranking system against itself.

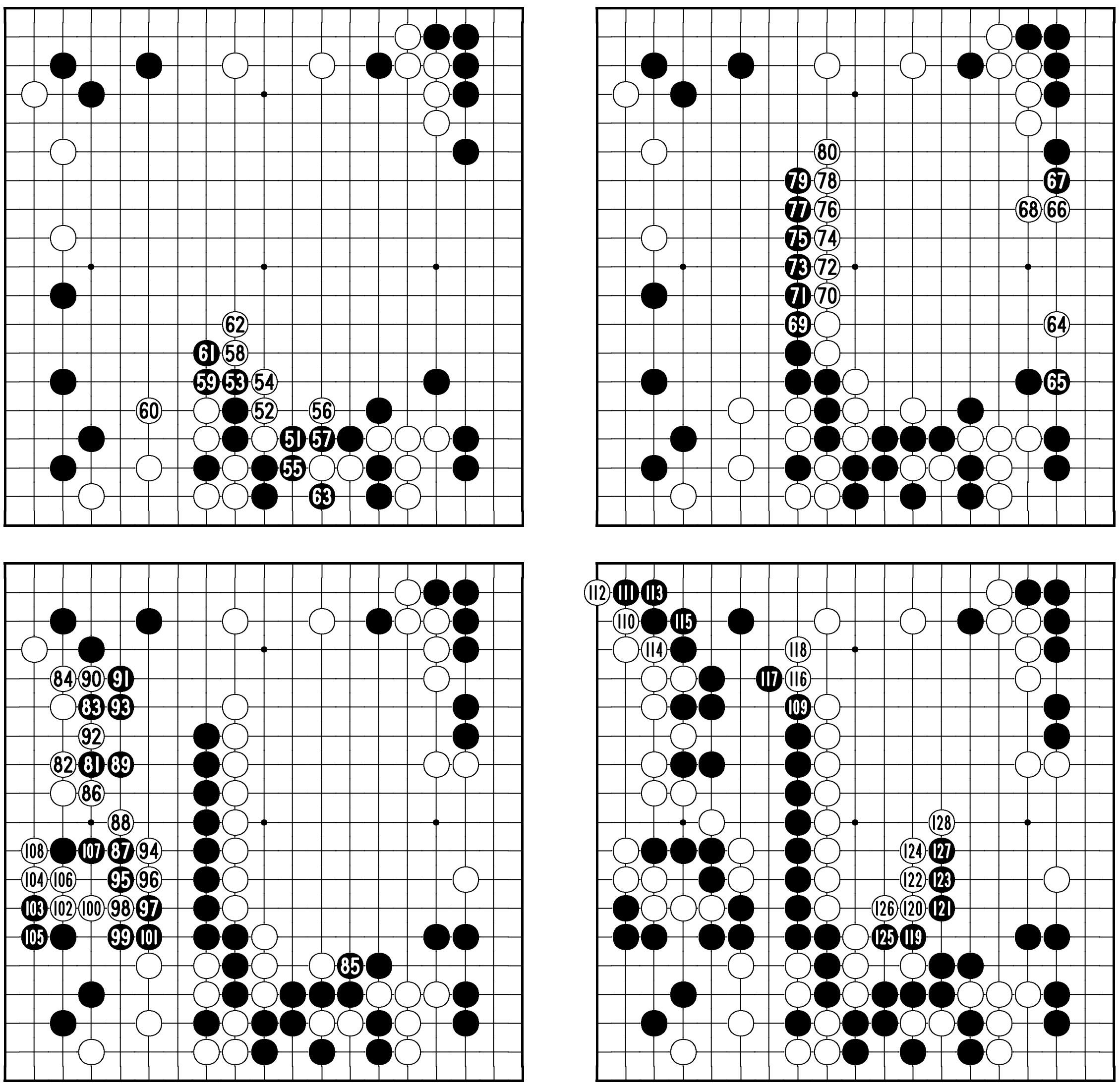


Figure 3.14: Continuation of a match of the Bayesian pattern ranking system against itself.

A game between this opponent and the pattern system is shown in Figures 3.15 and 3.16. The pattern system played as black with 9 handicap stones. The handicap stones should compensate for a difference of 9 levels between the players. A few things are worth pointing out. Firstly, the pattern system does not simply respond passively to the threats made by the opponent but makes moves into fresh areas of the board when no other more urgent move-patterns are available (knowing when to take the initiative in this way is important and is known as *tenuki* by Japanese Go players). See moves 28 and 56 for examples. The system responds correctly to urgent threats as long as the necessary information is contained in the pattern or features present. See move 86 for example. The pattern system crucially does not respond to white move 179 which leads to it losing the large black group in the bottom right of the board (as commented by the opponent). This is because the move does not seem locally urgent if you do not take into account the wider context which shows that the group is running out of eye space so will likely to be captured if no action is taken. After black loses this group we enter the end-game. Since the pattern system cannot recognise the fact that the groups in the bottom left and bottom right are dead it continues to waste moves trying in vain to save these groups. In the end, white (the human) wins by 52 points. The human player wrote that the pattern system 'successfully cut off a couple of my groups'. He also mentioned its weaknesses such as it 'played in dead groups. Failed to handle some of the situations between the borders of [the two player's] groups early enough. It did this quite well in the end game, but did not recognise earlier [...] that these could affect the lives of groups'. When asked about how consistently the pattern system seemed to play they responded that 'it makes some advanced moves in many phases but missed fairly obvious (to a human) things like when its group in the [bottom right] corner was in danger'.

Another player with a rating of 1 kyu (strong amateur level) judged that the pattern system had a strength of 9 kyu and commented that 'the opening was pretty good (although a bit passive for my taste). The AI has a good sense of shape.' When asked to comment on negative aspects of the pattern system's playing style the same player said that 'when the tactics get tricky, the AI makes silly moves (and probably due to its reliance on stronger players, whenever I relied on a ladder the AI would assume that it worked for me, even when that was not the case). In addition, the AI does not seem to have a good sense of which area is most urgent.' In response to a question about the consistency of play the player responded that 'The play was generally reasonably good, but occasionally the AI would ignore a significant threat from one of my moves.'

I believe that the picture that emerges from these (rather subjective) comments is that the system performs poorly at reading life and death (not surprising as local patterns can hardly be expected to be able to discriminate between live and dead groups) and performs poorly at predicting the results of local tactical battles (also not surprising as it seems likely that a search module is necessary to make good assessments of such situations as we will see in the next chapter). Also, the system has no representation of overall strategy and no attempt is made to globally evaluate board positions or predict the final territory outcome of the game which may explain the comment from a strong Go player that 'the AI does not seem to have a good sense of which area is most urgent'. However, the pattern system as Go player does make good 'shape' and plays a strong opening. Also, according to its human opponents, the pattern system makes a number of 'advanced' moves which challenge strong Go players. I believe that if the mistakes due to incorrect assessment of life and death could be eliminated then the playing strength could be improved.

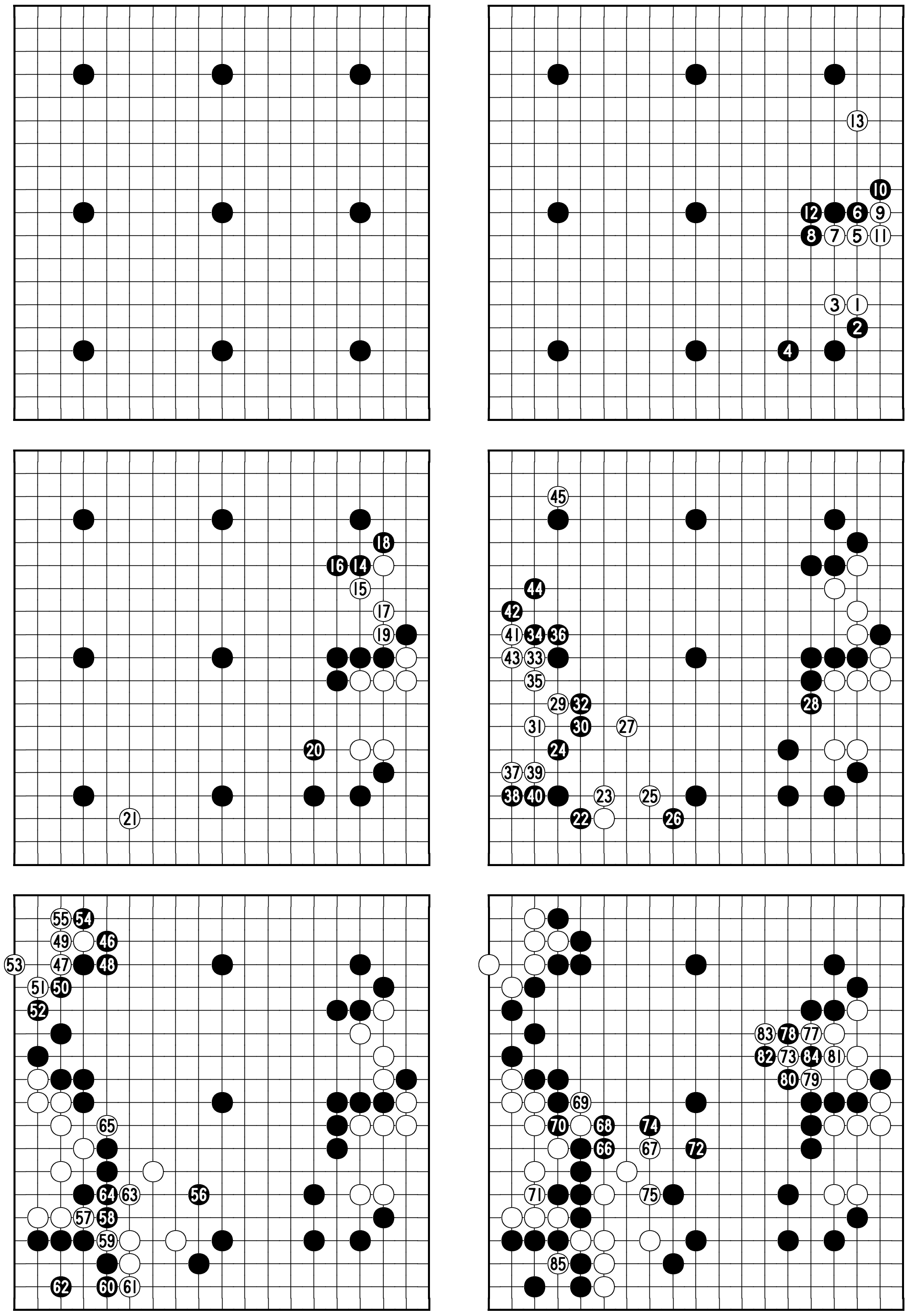


Figure 3.15: Diagram of the start of a match of the pattern system against a 2 *dan* human opponent. The pattern system played as black with 9 handicap stones (which should compensate for a difference of 9 rating levels between the computer and the opponent).

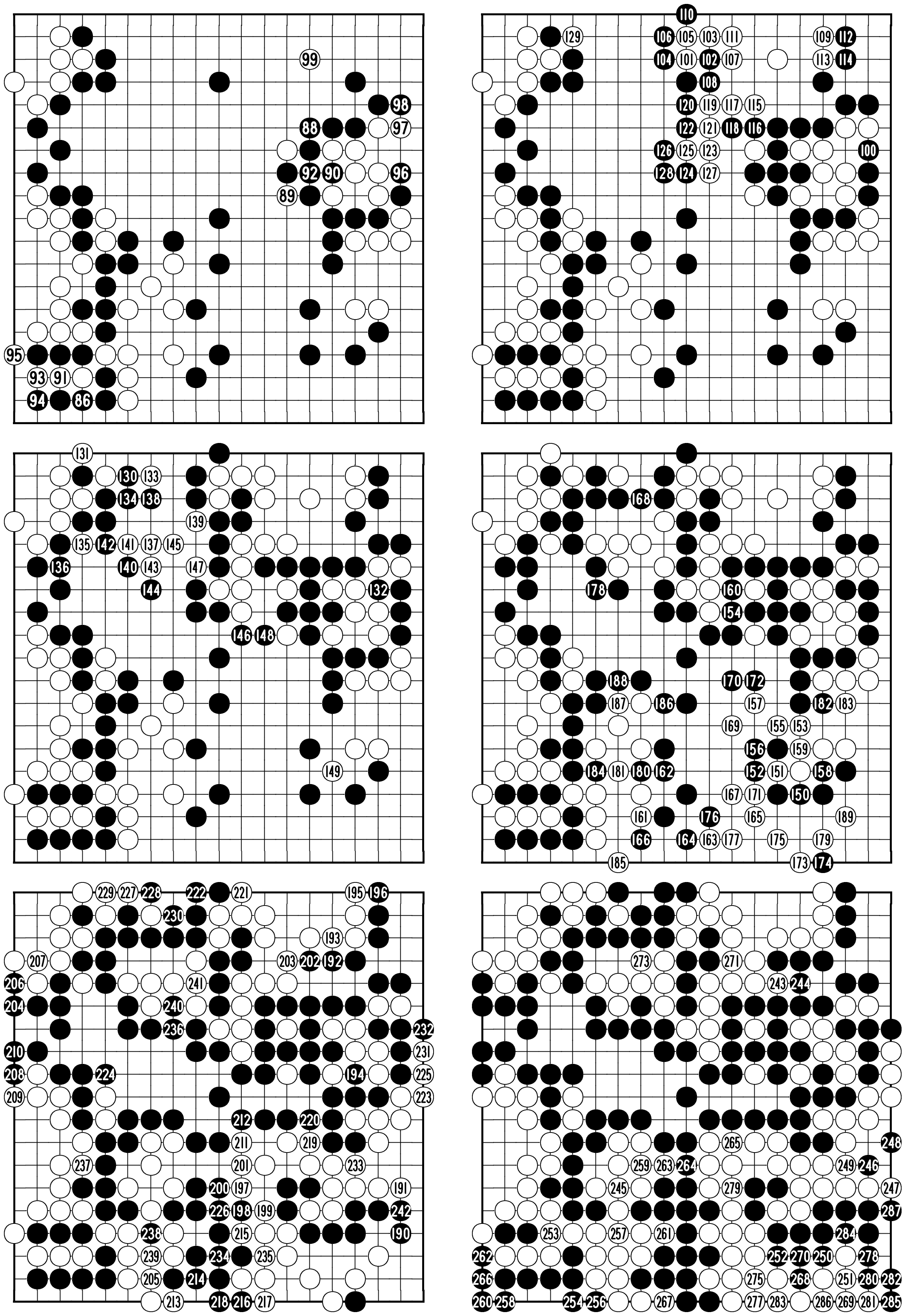


Figure 3.16: The rest of the match of the pattern system against the human opponent. White (the human player) won by 52 points and the two black groups in the bottom left and bottom right are dead.

3.3 Hierarchical Gaussian Pattern Model

A comparison can be made between the pattern system for Go move generation and language modelling in general. The patterns are used to predict Go moves based on a local context. An n-gram language model predicts future words (or letters) based on the previous $n-1$ words (letters). A simple method of dealing with data sparsity in language models is *backing-off* (Katz, 1997) where n is selected so only contexts that have been seen a sufficient number of times in training are used to make predictions. This means larger, more discriminative, contexts are used where available but a smaller context is used where necessary. The pattern system also makes use of this principle because only patterns seen more than a certain number of times in the training data are stored.

These similarities might suggest that we can learn from techniques used in language modelling. However, it should be realized that Bayesian language models (MacKay and Peto, 1994) define a generative model for text whereas the pattern system, on the other hand, does not necessarily lead to a generative model for Go positions. Instead, the patterns are used to associate a *value* with a particular vertex in a position. The role of the pattern system in conjunction with a probabilistic model is to provide a prior on this vertex-value whose definition depends on the application. In general, it could be related to the urgency of a move as in the move prediction task, the probability of a move leading to a certain goal being achieved (see the next chapter) or perhaps act as the prior on the final territorial ownership of the point in question at the end of the game as in Sanner *et al.* (2007).

The patterns form a hierarchy. The single vertex (size 1) 'grandmother' pattern is at the top. The set of all patterns matching the next template up (size 2) form the next level of the hierarchy and so on. Full board positions (centered on the point in question) form the bottom level. The pattern that matches in the same board position as another pattern but only within a template one size smaller is called the 'parent' of the other pattern. The hierarchical structure of the population of patterns suggests that their values should be modelled hierarchically.

Intuitively, the influence a pattern exerts on the vertex-value of the center vertex of the match should depend on the size of the pattern and the number of times it was seen in the training data. The evidence provided by a larger pattern should dominate over the evidence from a smaller pattern if they both match at the same location and they were both seen the same number of times in training. This is because a larger pattern contains all the information of a smaller pattern matching at the same location as well as additional information. However more accurate beliefs can be formed about patterns seen more frequently in training and smaller patterns are more common than large patterns. Therefore the smaller patterns should be allowed to influence the value of the larger patterns in cases where the larger patterns have been seen infrequently. These desiderata motivate a hierarchical model. Note that Bayesian hierarchical language models appear to recover the desired 'smoothing' automatically (MacKay and Peto, 1994).

We now outline our hierarchical model for pattern values. Let each pattern have a latent value. The distribution over this value forms the prior distribution over the possible values of the children of this pattern. The prior distribution of the single vertex pattern is defined to be Gaussian. Each latent pattern value is defined to be distributed as a Gaussian centered on the value of its parent pattern. We also say that the value of a vertex in a board position is distributed normally about the latent value of the largest pattern. In this way we define a generative model of a vertex-value in a given Go position. The values of the patterns are never observed directly but an observation can be made of the

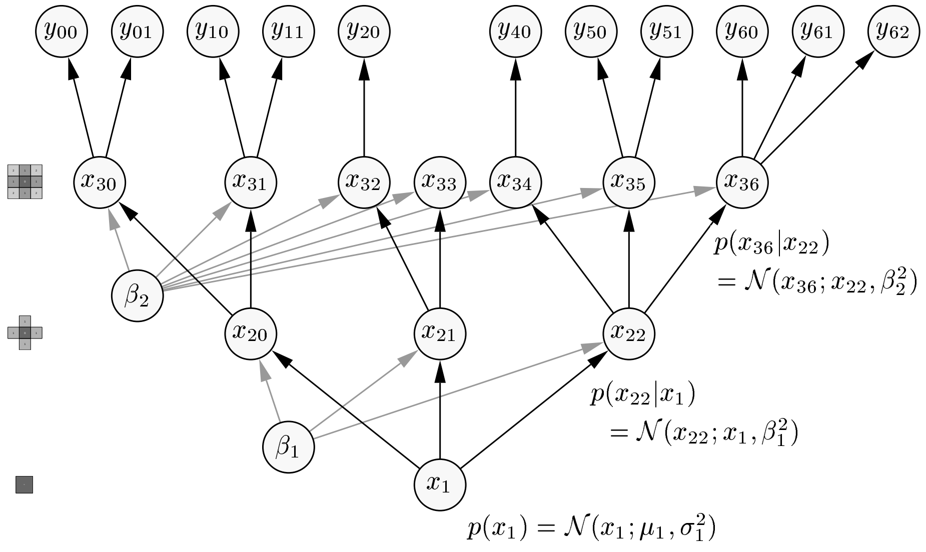


Figure 3.17: Hierarchical Gaussian pattern model. In this example the number of pattern sizes, $n = 3$. There are 11 patterns in total in the hierarchy and 11 observations. A realistic implementation contains millions of patterns and billions of observations.

value only in the context of a particular full board position. The conjugacy properties of the Gaussian distribution lead to efficient inference.

3.3.1 Model Details

The set of all observed point-values is denoted $\mathcal{Y}$. Also let the set of values of all possible patterns of all sizes be $\mathcal{X}$. Each member of $\mathcal{X}$ shall be denoted x_{ij} where x_{ij} is the value of the jth pattern of size i. The index of the smallest pattern size is 1 and the value of the smallest (single vertex) pattern denoted x_1. The largest pattern size is n and only the values of the largest patterns are ever observed directly (patterns are always observed in the context of a whole board position). An observed value, $y_{jk} \in \mathcal{Y}$ is the kth observation of the jth full board position.

The joint distribution $p(\mathcal{Y}, \mathcal{X})$ is defined to be:

$$\begin{aligned} p(\mathcal{Y}, \mathcal{X}) &= p(\mathcal{X}) \cdot p(\mathcal{Y}|\mathcal{X}) \\ &= p(x_1) \prod_{i=2}^{n} \prod_{k} \prod_{j} p(x_{ij}|x_{(i-1)k}) \cdot \prod_{q} \prod_{m} p(y_{qm}|x_{nq}) \end{aligned}$$

where

$$\begin{aligned} p(x_{ij}|x_{(i-1)k}) &= \mathcal{N}(x_{ij}; x_{(i-1)k}, \beta_{i-1}^2) \\ p(y_{rs}|x_{nl}) &= \mathcal{N}(y_{rs}; x_{nl}, \beta^2) \\ p(x_1) &= \mathcal{N}(x_1; \mu_1, \sigma_1^2) \end{aligned}$$

An example Bayesian network corresponding to this model for $n = 3$ is shown in Figure 3.17. The variance parameters, β_i^2, correspond to the variability of the latent value of patterns of sizes $i + 1$ about the value of the size i pattern that also matches. One would expect this value to be larger for small patterns due to the fact that smaller patterns contain less information for discrimination. It might also be expected that the variance β_i^2 depends on the difference between the sizes of templates i and $i + 1$. The predictive distribution over the value, y, of a vertex in a position where the index of the largest (size n) pattern matching at that location is j is found by integrating out all the other variables and observation beliefs:

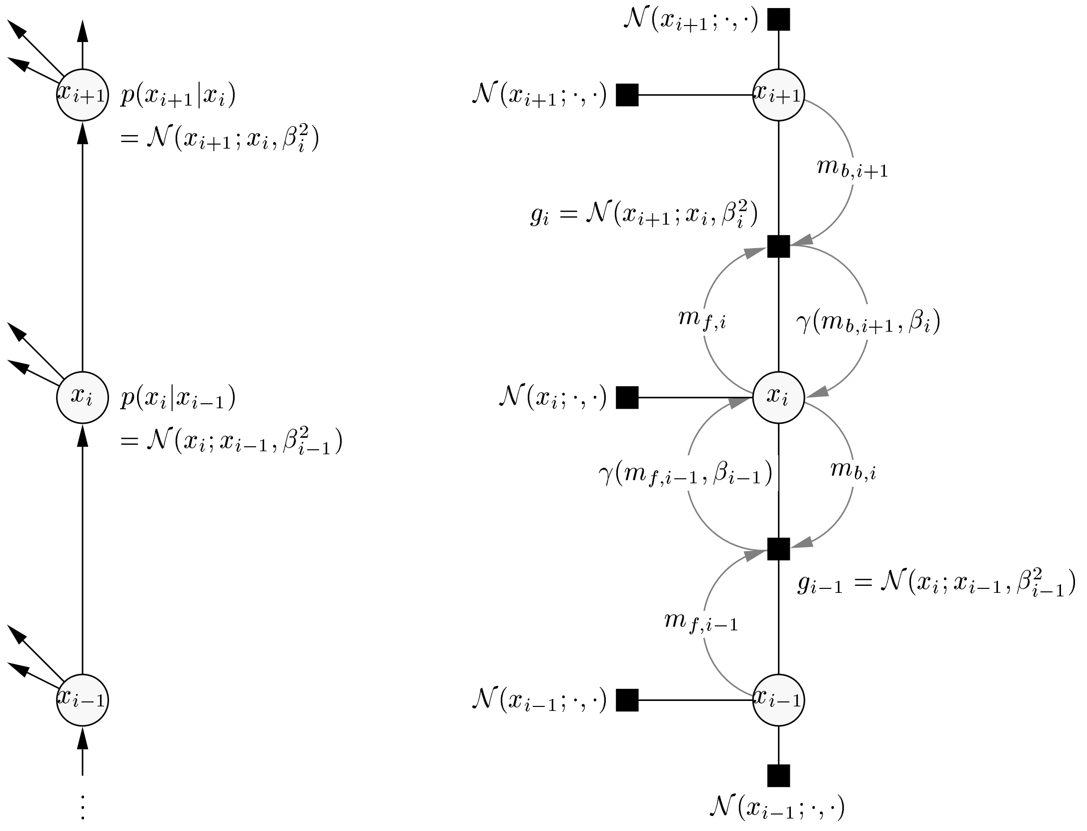
$$p(y) \quad = \quad \int_{\mathcal{Y}\setminus y} \int_{\mathcal{X}} p(y|x_{nj}) \cdot p(\mathcal{Y} \setminus y|\mathcal{X}) p(\mathcal{X}) \mathrm{d}\mathcal{X} \mathrm{d}\mathcal{Y}.$$

This integration can be accomplished by message passing.

3.3.2 Inference

Each node in the graphical model (Figure 3.17) corresponds to the latent value of a pattern in the hash table. Exact inference on this acyclic graph is accomplished using the sum-product algorithm for Gaussian densities (Section 2.3.2). Efficient inference is possible due to the fact that messages passing towards the root of the tree (which we call the 'backward messages') do not depend on the messages from the root of the tree (the 'forward messages').

The pattern system returns the sequence of nested patterns at a particular vertex on the board and we denote their values by $\mathbf{x} = \{x_1, x_2, \cdots, x_{n-1}, x_n\}$ (smallest ... largest). For the following discussion we drop the per template pattern index and just index each pattern in the sequence by its size. The stack of matching patterns, $\mathbf{x}$, corresponds to a single path from the root of the tree (Figure 3.17) to one of the leaves. Figure 3.18 shows a fragment of this path and its corresponding factor graph representation. Each conditional probability distribution is given by a factor g_i.



x_{i+1}
$p(x_{i+1}|x_i)$
$= \mathcal{N}(x_{i+1}; x_i, \beta_i^2)$
x_i
$p(x_i|x_{i-1})$
$= \mathcal{N}(x_i; x_{i-1}, \beta_{i-1}^2)$
x_{i-1}
$\vdots$
$\mathcal{N}(x_{i+1}; \cdot, \cdot)$
$\mathcal{N}(x_{i+1}; \cdot, \cdot)$
x_{i+1}
$m_{b,i+1}$
$g_i = \mathcal{N}(x_{i+1}; x_i, \beta_i^2)$
$m_{f,i}$
$\gamma(m_{b,i+1}, \beta_i)$
$\mathcal{N}(x_i; \cdot, \cdot)$
x_i
$\gamma(m_{f,i-1}, \beta_{i-1})$
$m_{b,i}$
$g_{i-1} = \mathcal{N}(x_i; x_{i-1}, \beta_{i-1}^2)$
$m_{f,i-1}$
$\mathcal{N}(x_{i-1}; \cdot, \cdot)$
x_{i-1}
$\mathcal{N}(x_{i-1}; \cdot, \cdot)$

Figure 3.18: Fragment of hierarchical model with messages labelled.

The forward message from pattern value node x_i is denoted $m_{f,i}$. The backward message from pattern value node x_i is denoted $m_{b,i}$. Let γ denote convolution with a Gaussian density (the addition of Gaussian noise). For example, if the message $m_{b,i+1} = m_{x_{i+1}\to g_i} = \mathcal{N}(x_{i+1}; \mu_m, \sigma_m^2)$ then equation 2.19 yields:

$$\begin{aligned} m_{g_i \to x_i} &= \mathcal{N}(x_i, \mu_m, \sigma_m^2 + \beta_i^2) \\ &= \gamma(m_{x_{i+1}\to g_i}(x_{i+1}), \beta_i). \end{aligned}$$

The forward messages can be stated in terms only of other messages up and down this factor graph fragment via the marginal update equation (2.13) to give:

$$m_{f,i} = m_{x_i \to g_i}(x_i) = \frac{p(x_i)}{m_{g_i \to x_i}} = \frac{\gamma(m_{f,i-1}, \beta_{i-1}) \cdot m_{b,i}}{\gamma(m_{b,i+1}, \beta_i)}. \tag{3.4}$$

Similarly the backward messages are found by:

$$m_{b,i} = \frac{\gamma(m_{b,i+1}, \beta_i) \cdot m_{f,i}}{\gamma(m_{f,i-1}, \beta_{i-1})}. \tag{3.5}$$

As a special case of (3.4) the forward messages from the root node are calculated via:

$$m_{f,1} = \frac{p(x_1)}{\gamma(m_{b,2}, \beta_1)}. \tag{3.6}$$

Each pattern stores one Gaussian density: $p(x_1)$ for the single vertex pattern and the backward messages, $m_{b,i}$, for every other pattern. To determine $p(y_i)$, the predictive probability distribution over the point-value in a board position, the forward messages must be updated for the stack of observed patterns at this point. This is achieved by using equation 3.4 for each pattern (starting at the root and working up). Note that since the backward messages are cached and up to date, this calculation requires a single sweep forward from root to the leaf in question (linear in the number of template sizes). The learning update is achieved by propagating the backward messages according to (3.5), caching the new backward message for each pattern and updating the marginal probability of the root (also linear in the number of template sizes).

When harvesting new patterns, the Gaussian density stored for each pattern (apart from the zero sized pattern) is set to be uniform ($\sigma = \infty$), which means that the backward messages are uniform, corresponding to no prior observations. The Gaussian density for the single vertex pattern (corresponding to $p(x_1)$) is set to $\mathcal{N}(0, 100)$.

3.3.3 Move Prediction Experiments

The performance of the hierarchical model was evaluated for the task of move prediction. For this application we do not observe the numerical value, y, of a move directly. Rather, we let this value be the move value as in the model of Section 3.2.1 so $y = u(\vec{v}, c)$ and the distribution over moves is given by equation 3.1. In other words we use the hierarchical model to provide the Gaussian prior on a move value and the observations are in the form of constraints on the ordering of the values of moves based on the decisions made by the human players. Inference is achieved by message passing on the composite factor graph of the two models. Clearly the full composite graph may contain loops because the smaller patterns may match for multiple available moves but we ignore this issue, much

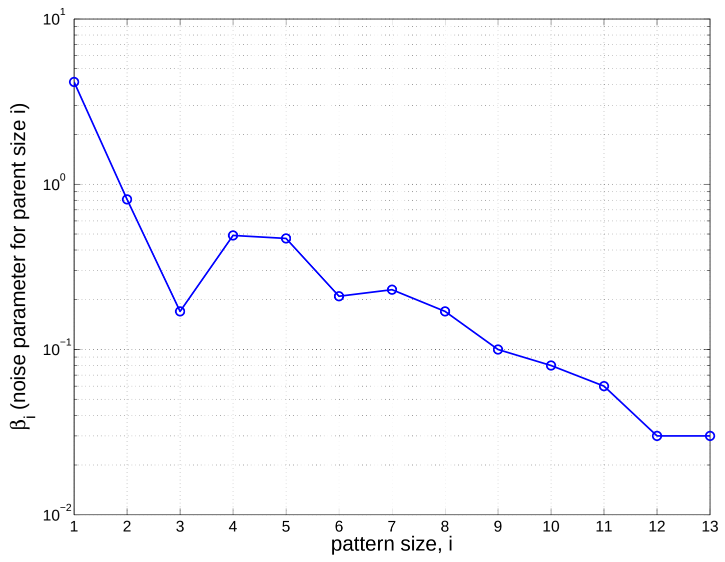


Figure 3.19: Empirically estimated β_i values.

as we implicitly ignored it previously by using ADF. All 14 pattern templates were used so $n = 14$ (see Figure 3.2).

Empirical Determination of Variance Parameters

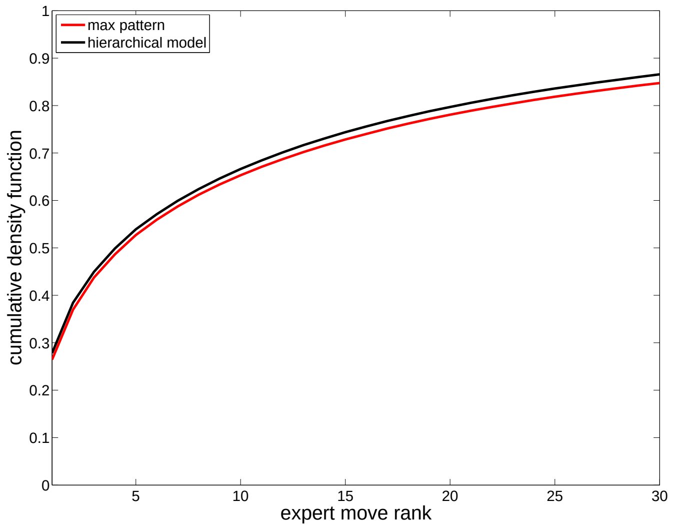
The variance parameters, β_i^2, must be determined. The full Bayesian approach would be to place a prior on them and marginalize them out. For the experiments here we simply estimated the values using

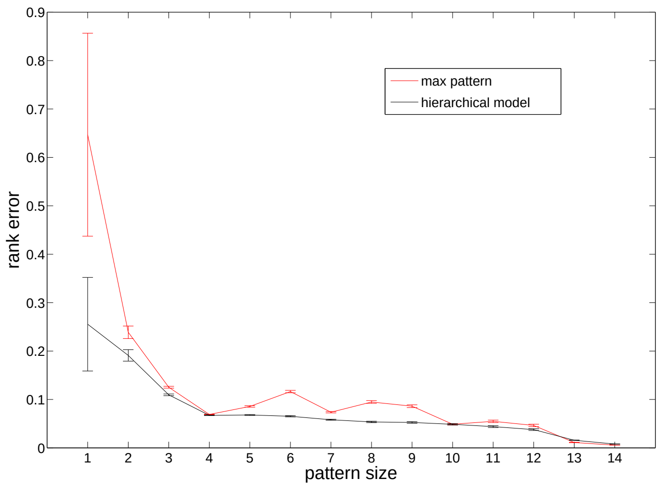
$$\beta_i^2 = \frac{\sum_k [(\mu_{i+1,k} - \mu_{i,\text{pa}(k)})^2]}{\sum_k 1}$$

where $\mu_{i+1,k}$ is the mean value of pattern k of size $i + 1$ and $\mu_{i,\text{pa}(k)}$is the mean value of its parent pattern as learned using the ranking model when matching only the maximum sized pattern (as in Section 3.2.3). The resulting values are shown in Figure 3.19. Note also that the magnitude of β_i depends on the relative difference in size between successive nested templates (figure 3.2). The expectation is taken over patterns harvested and trained on the GoGoD training set (training only the biggest pattern to match for each move as in the experiments in Section 3.2.3). It should also be pointed out that the training set used to tune these parameters does not intersect any test set used here. These β_i settings were used for the experiments in this section but in practice it was found that the performance depended little on this setting (very similar results are achieved if β_i is set to 1).

Results

Figures 3.20 and 3.21 compare the performance of the pattern system using the hierarchical model as the prior on the move value against the results obtained when only the maximum sized pattern is used (as in the previous experiments). The system was trained on 18,000 expert games and tested on 1352 games. Figure 3.20 shows that the move prediction performance of the hierarchical model is better at every stage of the game and for every pattern size, although the increase is rather modest, especially at the start of the game. Figure 3.21 compares the probability of the expert moves according to the two models. This shows that the hierarchical model is a better probabilistic model throughout most





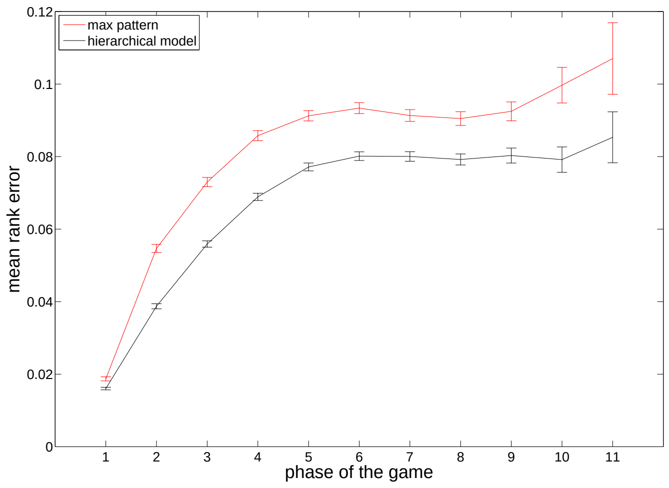


Figure 3.20: The performance of the hierarchical model is compared with the performance of the model using only the largest matching pattern. The models were trained on 18,000 games and tested on 1352 test games. **Top**: Cumulative distribution of the ranks the models assign to the moves played by expert players. **Middle**: Mean rank error for each pattern size. Rank error is $\frac{\text{expert move rank}}{\text{number available moves}}$. **Bottom**: Mean rank error for different phases of the game (each phase is 30 moves).

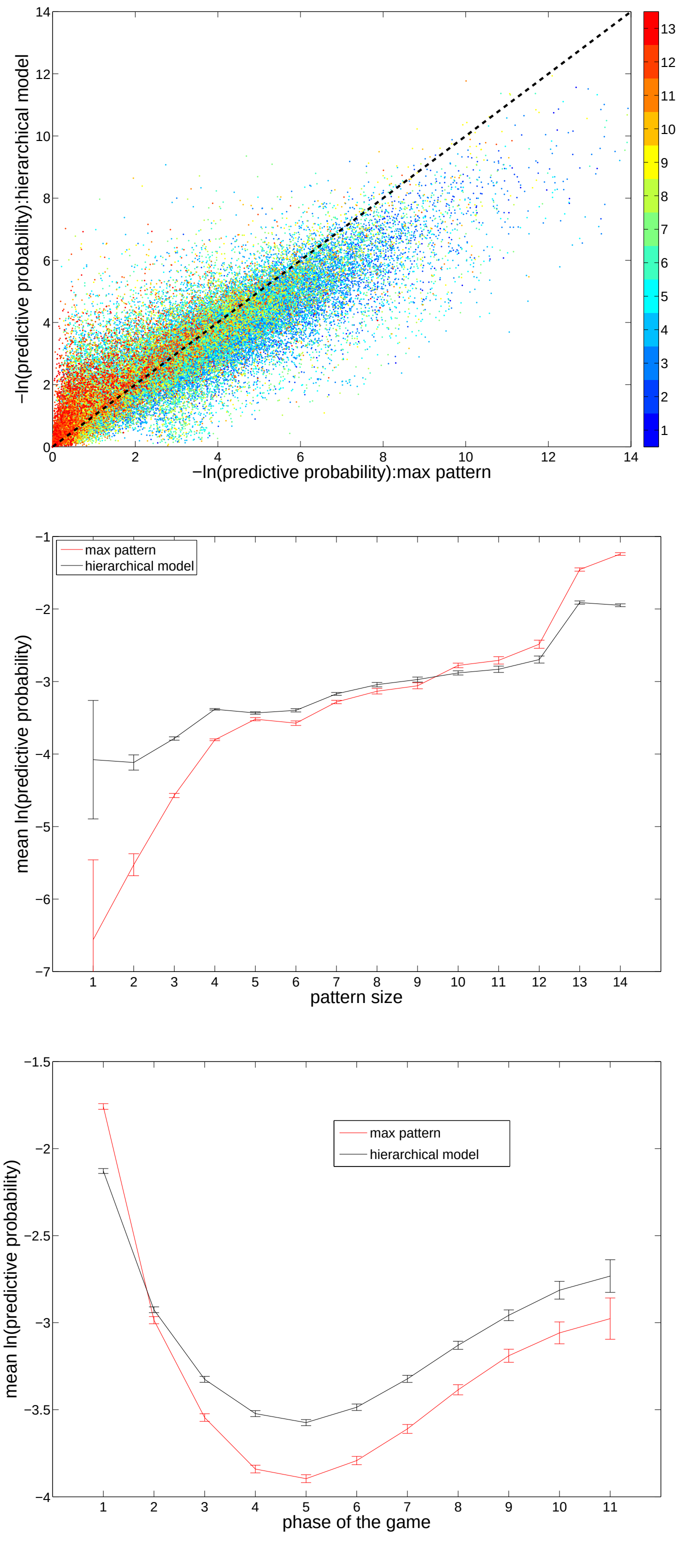
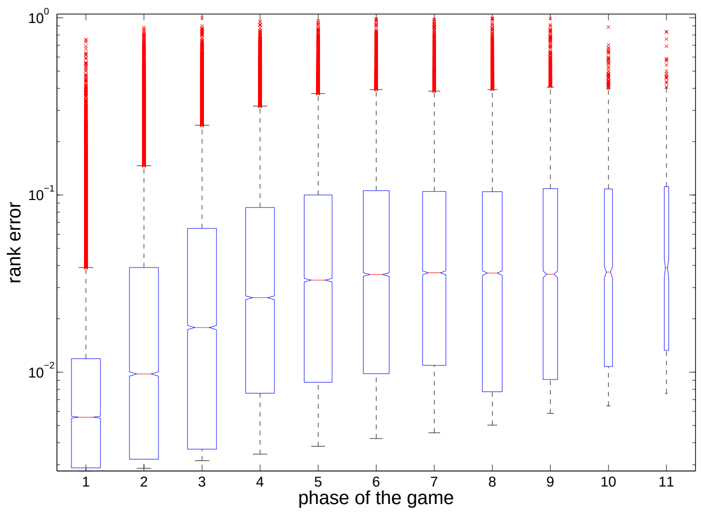


Figure 3.21: The probability of the moves in unseen expert games according to the hierarchical model is compared with the probability of the moves according to the model using the largest pattern. The models were trained on 18,000 games and tested on 1352 games. **Top**: Scatter Plot. Each colour corresponds to a different maximum pattern size (for the model using only the largest pattern). **Middle**: Mean predictive probability for each pattern size. **Bottom**: Mean predictive probability for different phases of the game (each phase is 30 moves).



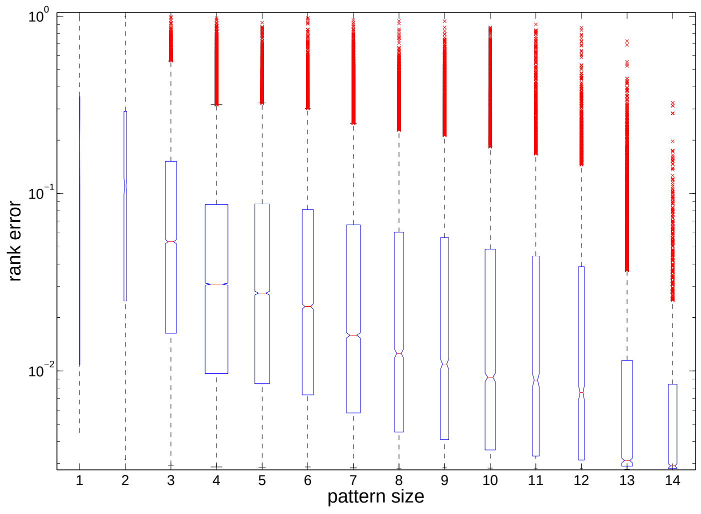


Figure 3.22: Box plots showing test performance of the hierarchical model on 1352 unseen games. The models were trained on 18,000 games. Lower and upper sides of box: quartiles; horizontal line across middle: median; width: number of data points; whiskers (dashed lines): extent of data within $3 \times$ IQR; dots: data points outside $3 \times$ IQR. **Top:** Box plot of the rank error for different phases of the game, each phase corresponding to an interval of 30 moves. Rank error is $\frac{\text{expert move rank}}{\text{number available moves}}$. **Bottom:** Box plot of the rank error for different pattern sizes.

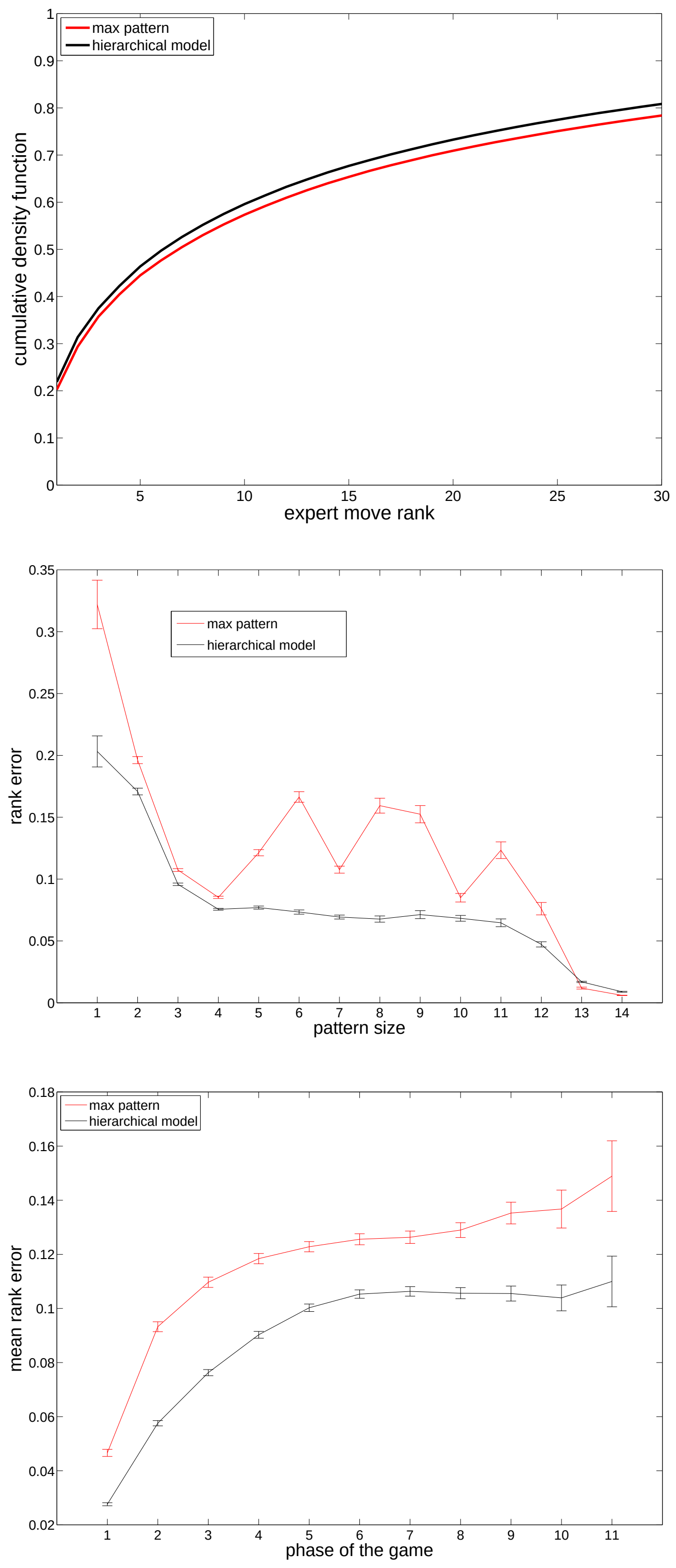
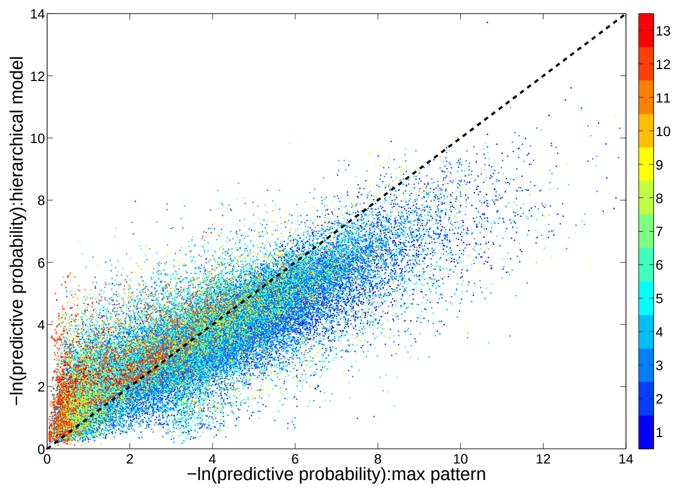
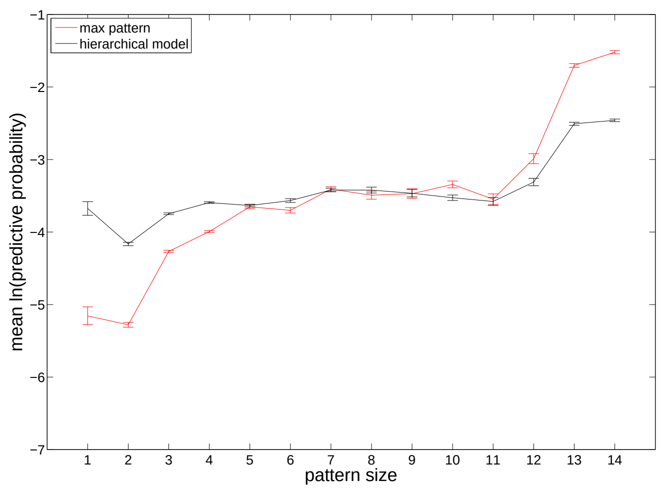


Figure 3.23: The performance of the hierarchical model is compared with the performance of the model using just the largest matching pattern. Here both models are trained on only 684 games. The test set here was 211 unseen games (a subset of the other test set). **Top**: Cumulative distribution of the ranks the models assign to the moves played by expert players. **Middle**: Mean rank error for each pattern size. Rank error is $\frac{\text{expert move rank}}{\text{number available moves}}$. **Bottom**: Mean rank error for different phases of the game (each phase is 30 moves).





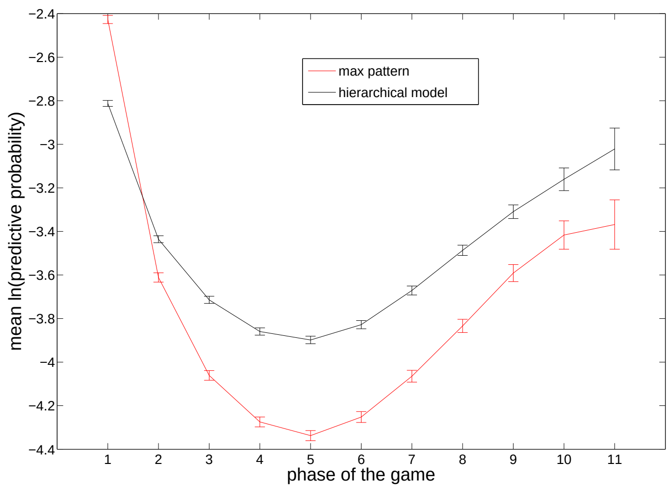
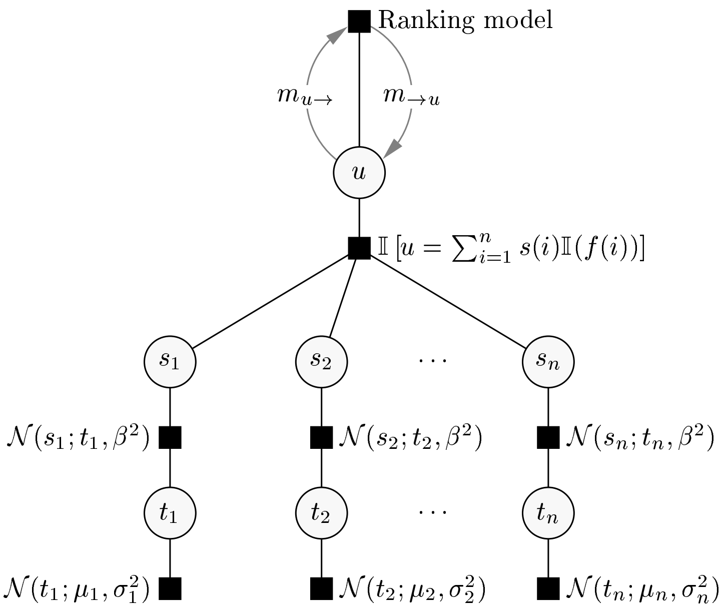


Figure 3.24: The probability of the moves in unseen games according to the hierarchical model is compared with the probability of the moves according to the model using just the largest matching pattern. Both models are trained on only 684 games. The test set here contained 211 games (a subset of the other test set). **Top**: Scatter Plot. Each colour corresponds to a different maximum pattern size (for the model using only the largest matching pattern). **Middle**: Mean predictive probability for each pattern size. **Bottom**: Mean predictive probability for different phases of the game (each phase is 30 moves).



Ranking model
$m_{u\to}$
$m_{\to u}$
u
$\mathbb{I}\left[u = \sum_{i=1}^{n} s(i)\mathbb{I}(f(i))\right]$
s_1
s_2
$\cdots$
s_n
$\mathcal{N}(s_1; t_1, \beta^2)$
$\mathcal{N}(s_2; t_2, \beta^2)$
$\mathcal{N}(s_n; t_n, \beta^2)$
t_1
t_2
$\cdots$
t_n
$\mathcal{N}(t_1; \mu_1, \sigma_1^2)$
$\mathcal{N}(t_2; \mu_2, \sigma_2^2)$
$\mathcal{N}(t_n; \mu_n, \sigma_n^2)$

Figure 3.25: Factor graph fragment showing that the move value is modelled as the sum of the values of a set of features. The factor at the top corresponds to beliefs propagated from the ranking model. Note that the full composite graph of the ranking model contains cycles if the same feature is present in more than one of the legal moves in a position (as is often the case). This issue is ignored for the experiments here.

of the game except during the first 30 moves. The model is worse at the start of the game because the available moves correspond to large patterns and the model predictive probability is worse for the larger patterns as shown by Figure 3.21 (middle).

Figures 3.23 and 3.24 make the same comparisons for the case the system is trained on 684 games only and show that the increase in performance yielded by using the hierarchical model is slightly larger. This may be because large patterns are less frequently matched as the pattern table is smaller so smaller gains are achieved by smoothing over contexts of different sizes by the hierarchical model.

3.4 Linear Feature Model

Since the initial publication of this large scale application of patterns for move prediction in Go (Stern *et al.*, 2006) the work was followed up by Coulom (2007). Instead of the full Bayesian ranking model described in Section 3.2.1 he used a generalisation of the Elo ranking system used internationally to rank chess players (also known as the Bradley-Terry model). His approach allowed him to avoid the exponential number of parameters ($2^{|F|} \times$ #patterns) required by the addition of the pattern features described in section 3.1.2. He achieved strong move prediction performance, probably because he was able to use a much larger set of features. This section demonstrates that by using perhaps the simplest possible approach (a linear model) we can use a large number of features. We assume that the value of a move is the sum of the values of a number of features in the same manner as Coulom (2007) but using the full ranking model of Section 3.2.1.

3.4.1 Model Description

The value of a move, u, is modeled as the sum of the values of a set of binary features. The set of features is given by $F = \{1 \cdots n\}$ and the function $f : F \to \{\text{TRUE}, \text{FALSE}\}$ maps each feature to its logical state. The function $s : F \to \mathbb{R}$ returns the value of each feature in a particular position, $s(i) := s_i$. The distribution over the value, u, of a move is given by:

$$p(u|s_1, s_2, \cdots, s_n) = \mathbb{I}\left(u = \sum_{i \in F} s(i)\mathbb{I}(f(i))\right).$$

The value of the feature in a particular position, s_i, is taken to vary about the underlying value of the feature, t_i, with Gaussian noise having variance β^2 so $p(s_i|t_i, \beta) = \mathcal{N}(s_i; t_i, \beta^2)$ The prior on the underlying value of each feature is taken to be Gaussian, $p(t_i) = \mathcal{N}(t_i; \mu_i, \sigma_i^2)$. This yields a Gaussian predictive distribution on the value of a move,

$$p(u|\boldsymbol{\mu}, \boldsymbol{\sigma}, \beta) = \mathcal{N}\left(u; \sum_{i \in F} \mu_i \mathbb{I}(f(i)), \sum_{i \in F}(\sigma_i^2 + \beta^2)\mathbb{I}(f(i))\right).$$

This is used as the prior on $u(\vec{v}, c)$, the move value as in the ranking model of Section 3.2.1, that is, the probability of a move in a position is given by equation 3.1. Figure 3.25 shows the relevant factor graph fragment. The learning procedure is the same as before except now messages are propagated to each feature variable, t_i, and the mean and variance of each feature are stored. The message updates required for inference in this part of the graph are given by equations 2.20 and 2.19. The full composite factor graph produced by combining the ranking model with the linear feature model is likely to contain multiple cycles due to a particular feature being TRUE for multiple moves. For the experiments in this section we ignore this issue and the update messages are just propagated to the feature variables for each move in a position separately.

3.4.2 Features

For a pattern table containing m patterns, the first m features in F indicate which pattern is the largest to match for the move (only one of m will be TRUE). The other features are explained in Table 3.1 and are similar to those used by Coulom (2007). The Monte Carlo territory feature is calculated by playing 64 random games from the current position. The feature state is determined by the number of times that the empty vertex where the potential move is to be played is owned at the end of the random game by the player to move. Since the simple linear model cannot detect the importance of a conjunction of features we compensate for this lack of representational power by including some features which are in fact a combination of features of the position, such as ‘atari when ko is present’ to characterise a ko threat.

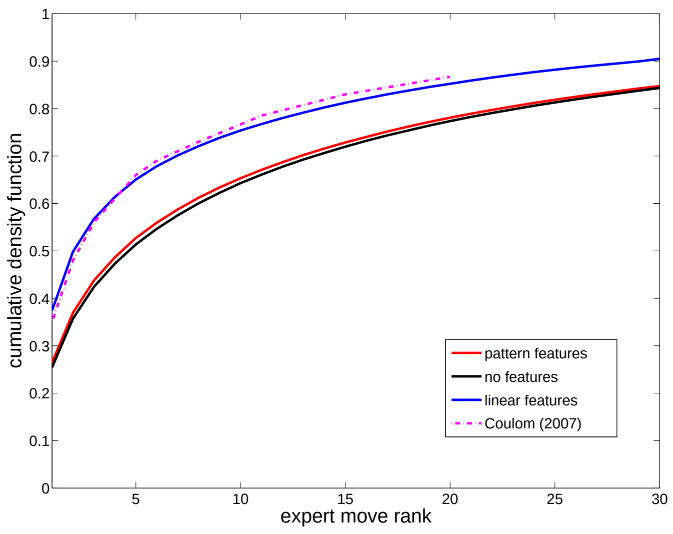
3.4.3 Results

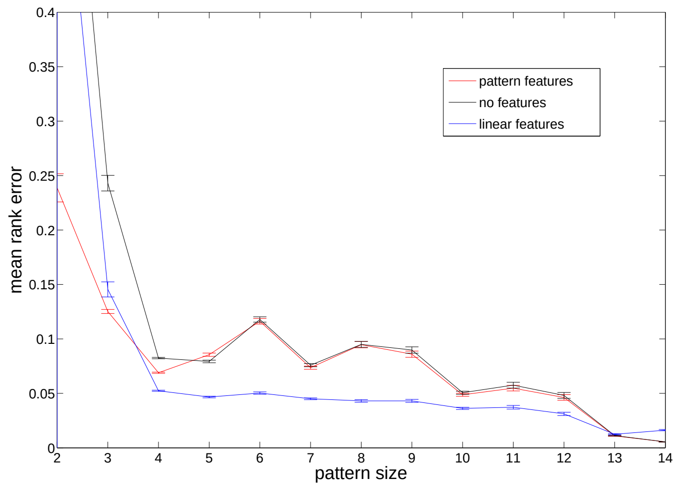
Figures 3.26 and 3.27 show that (trained on 18,000 games) the linear feature model produces better results than our previous method of including features, both in terms of predictive probability and move ranking performance. This is presumably because far more features are included. Only at the start of the game does the inclusion of the extra features make no improvement because all of the information needed to predict a move is already present in the patterns because they are sufficiently large.

Figure 3.26 (top) also includes the corresponding cumulative density function curve from Coulom (2007). We perform slightly better at the start of the curve (we predict the expert move correctly

Feature Category	State	μ_i	Details
Capture	1	2.11	Capture.
	2	-0.18	Capture but stone in ladder anyway.
Escape Atari	1	1.85	Escape atari.
	2	0.42	Escape atari but still in ladder.
Atari	1	0.87	Atari in ladder.
	2	1.55	Atari when ko present.
	3	1.03	Other Atari.
Liberties of New Chain	1	-0.75	1 liberty means self-atari.
	2	-0.06	
	3	0.18	
	4	0.19	
	> 4	0.22	
Distance to Edge	0	-0.16	
	1	0.45	
	2	0.48	
	3	0.22	
	4	0.04	
	5	0.02	
Monte Carlo Territory	1	-1.56	0–7
	2	-0.67	8–15
	3	0.13	16–23
	4	0.38	24–31
	5	0.29	32–39
	6	-0.23	40–47
	7	-1.24	48–55
	8	-2.61	56–63
Liberties of Closest Opponent	1	-0.17	Liberties of closest opponent chain
	2	0.38	(if distance less than 6.)
	3	0.06	
Distance To Previous Move	1	2.12	
	2	1.46	
	3	1.10	
	...		
	16	-0.14	
	≥ 17	-0.19	
Distance to Move Before Previous	1	1.13	
	2	0.83	
	3	0.61	
	...		
	16	0.24	
	≥ 17	0.25	

Table 3.1: Features used by the linear model. There are 9 feature categories which can each take on a number of states. This leads to 63 binary features in addition to the largest matching pattern which together are used to characterise a move. The table also shows the mean, μ_i, of the posterior distribution over the value, $t(i)$, of each feature after training.





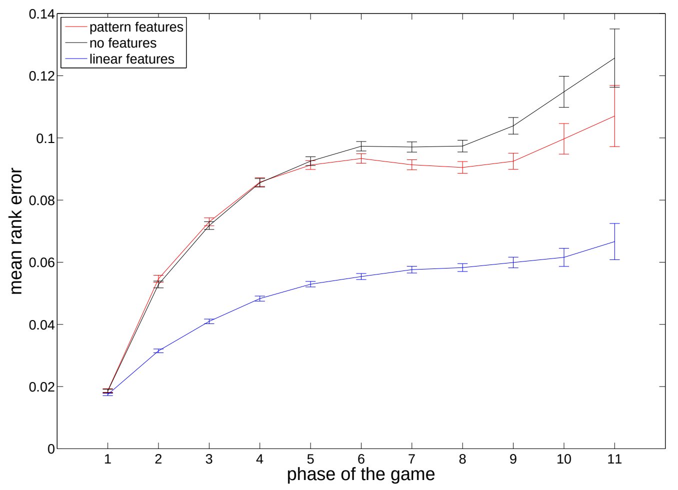
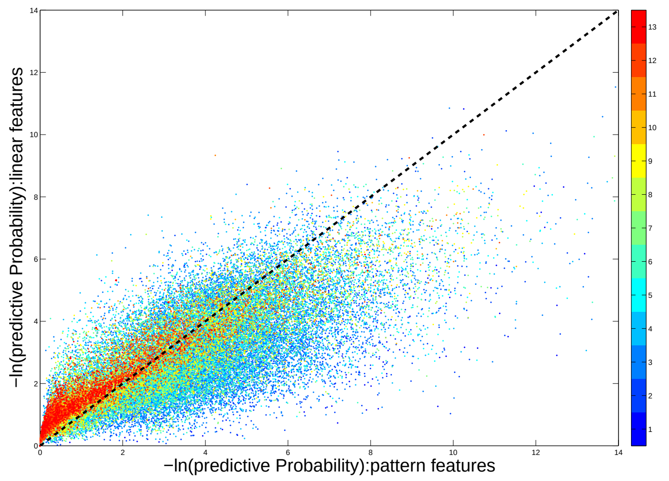
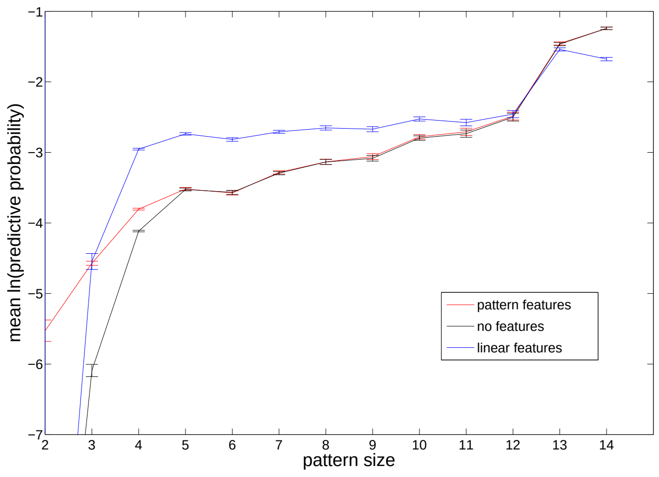


Figure 3.26: Comparing the performance of 3 methods of feature inclusion: no features (besides the patterns), binary pattern features as in Section 3.1.2 and the linear feature model. The models were trained on 18,000 games. **Top**: Cumulative distribution of the ranks the models assign to the moves played by expert players. **Middle**: Mean rank error for each pattern size. Rank error is $\frac{\text{expert move rank}}{\text{number available moves}}$. **Bottom**: Mean rank error for different phases of the game (each phase is 30 moves).





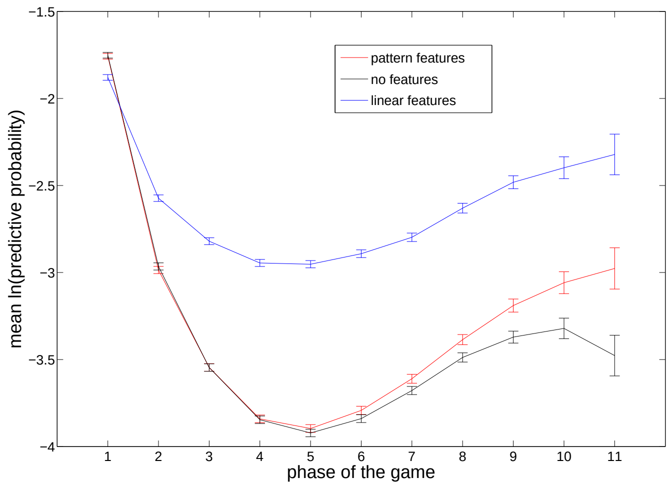
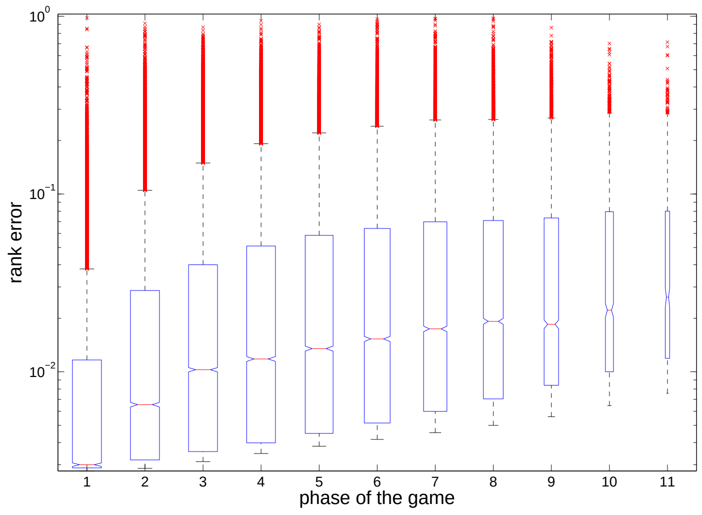


Figure 3.27: Comparing the probability of the expert moves in unseen expert games for 3 methods of feature inclusion: no features (besides the patterns), binary pattern features as in Section 3.1.2 and the linear feature model. The models were trained on 18,000 games. **Top**: Scatter Plot comparing the predictive probability of the linear feature model with the pattern features. Each colour corresponds to a different maximum pattern size. **Middle**: Mean predictive probability for each pattern size. **Bottom**: Mean predictive probability for different phases of the game (each phase is 30 moves).



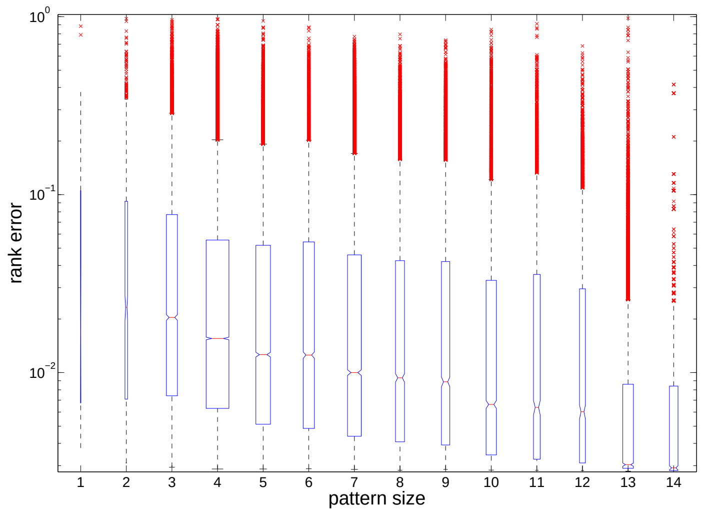


Figure 3.28: Box plots showing test performance of the linear feature model on 1352 unseen games. The models were trained on 18,000 games. Lower and upper sides of box: quartiles; horizontal line across middle: median; width: number of data points; whiskers (dashed lines): extent of data within $3 \times$ IQR; dots: data points outside $3 \times$ IQR. **Top:** Box plot of the rank error for different phases of the game, each phase corresponding to an interval of 30 moves. Rank error is $\frac{\text{expert move rank}}{\text{number available moves}}$. **Bottom:** Box plot of the rank error for different pattern sizes.

37% of the time) and slightly worse at the tail of the curve. The linear feature model is trained on 18,000 games yet we obtain similar performance as the previous experiment where the system was trained on 181,000 games and used only the 8 pattern features (Figure 3.6). Perhaps the move prediction performance of the linear model could be increased further by training on more data. The computational cost is now somewhat higher than before because of computationally expensive features such as the ladder features and the Monte Carlo feature. The system can predict moves at a rate of about 10 positions per second on a standard 2GHz PC.

Feature Values

It is interesting to examine the values learned for the features during training (Table 3.1). The ordering of the values of the different features is similar to that found by Coulom (2007). Ataris and captures are frequently played when they are available (as long as the stones would not be captured in a ladder anyway). Ko threats are also learned.

The value of a move seems to strongly depend on the predicted territory at the point in question (the Monte Carlo feature). It seems that moves tend not to be made into areas that are either black or white territory but are more likely to be made into regions of the board where ownership has not yet been decided. This corresponds with standard Go-playing wisdom. Interestingly, moves into one's own sure territory are penalised even more than moves into the opponent's sure territory. Possibly this is because a move into one's own territory is almost certainly a waste whereas a move into the opponent's territory could represent a threat, which costs nothing if it requires a response from the opponent.

As van der Werf *et al.* (2002) discovered, the distance of a move to the previous move is a valuable feature for the task of move prediction and we find that here too. Players presumably don't play moves merely because they are close to the previous move, but because they are good moves! Logically, if you're playing an optimal opponent, the optimal move in a given position has no dependence on the previous move. If experts were optimal and we were to produce a model that captured all important aspects of the board configuration then presumably the value of the move would be fully explained away by this model and the distance to the previous move would become an unimportant feature.

3.5 Discussion

In the introduction we referred to a distinction between static and dynamic approaches to position evaluation. Pattern matching falls into the category of static evaluation methods. By restricting the system to matching only exact, local, symmetrical, non-overlapping templates we achieve surprisingly accurate move prediction performance at little computational cost. There are many possible extensions which might improve the performance of the model. For example we could use templates with variable shape which could take into account the structure inherent in the arrangement of the stones on the board (perhaps using a common fate type representation (Graepel *et al.*, 2001)).

A strength of the pattern system is that it is fast enough to work within the inner loop of a larger Go playing system. This suggests that these computationally expensive extensions may not be worthwhile. I think the most worthwhile direction forward would be to move towards using a more dynamic approach to position evaluation which uses the pattern system to incorporate static knowledge in such a way as to exploit the speed of the pattern system. This is because although it is possible for a pure static method to be accurate in principle (a very large look up table would do), I believe that static evaluation alone seems an inefficient means of evaluating a move. A dynamic evaluation can adapt and learn about the position at hand by exploring the local part of the state space, rather than having to know in advance how to play in every possible position, as a static method must. The recent success of Monte Carlo planning for Go adds weight to this argument.

The pattern system as it stands can predict expert moves at a rate of hundreds per second but by restricting the size of the maximum pattern matched and by reducing the set of other features being checked the speed can be increased to a rate of thousands of moves per second. In that case the speed of the pattern matcher is sufficient that it can be used to guide search and this application is investigated in the next chapter. Another application of the pattern matching system is to guide adaptive Monte Carlo planning (as discussed in Chapter 6).

Chapter 4

Learning To Solve Game Trees[1]

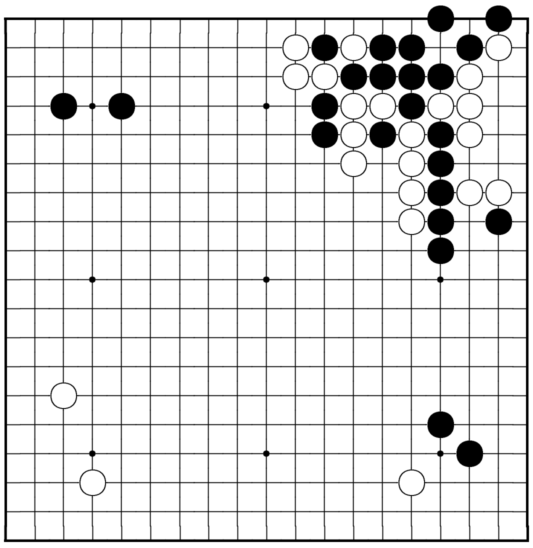


Figure 4.1: A battle emerges in the top right of the board.

In this chapter, we consider the task of proving whether goals can be achieved by a player in an adversarial game. The task is to prove that a player (the 'attacker') can achieve the goal, whatever the actions of the opponent (the 'defender'). Such problems are solved by searching the state space (the game tree) (Pearl, 1984; Russell and Norvig, 1995). We saw in Section 1.3 that it is intractable to use minimax search to select moves in the game of Go because the evaluation function is too complex. However, search can be applied to sub-problems in Go such as the task of determining whether a particular stone on the board can be captured. The evaluation function for that task is trivial: return 'TRUE' if the stone is captured and 'FALSE' otherwise. Therefore, the ideas developed in this chapter are applied to the task of solving capture problems in Go. A system that can solve capture problems is an important sub-component of most Go programs and the results returned by this system could be useful for position evaluation or for building a structured board representation.

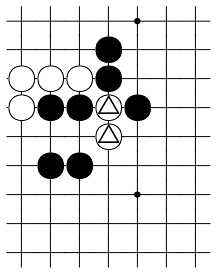
[1]The content of this chapter builds on work carried out in collaboration with Thore Graepel and Ralf Herbrich and presented at the International Conference of Machine Learning 2007 (Stern *et al.*, 2007).

We assume every node in a game tree has an underlying 'Delphic' value: the Boolean value that would be returned by an oracle with perfect knowledge (Palay, 1985). This value is TRUE for a node if the goal can be provably achieved in the corresponding position and FALSE if the goal cannot be achieved. After fully exploring the tree we can determine by logical deduction the value of the root node. In this case the tree is 'solved' and search terminates. During a search the values of some nodes are not yet determined and we can quantify the uncertainty about these values using probabilities. We assign a probability of 1 to every TRUE node and probability 0 to every FALSE node. All other nodes have some probability between 0 and 1 which represents a degree of belief about whether the node is TRUE (see Section 4.3). The final outcome of search does not depend on these probabilities. Rather, they are used to guide search to find the solution more rapidly.

By placing prior distributions on the values of the leaf nodes of the game tree we can incorporate knowledge into search. These distributions are surrogates for the unexplored parts of the tree. As searches are performed, nodes become proved as TRUE or FALSE and these proofs can be used to update the surrogate distributions so future searches are more efficient. As more is learned and the surrogate distributions become more accurate we are essentially compressing the game tree.

Many individual nodes must be solved in order to solve a game tree and each node represents a search problem in its own right so a problem decomposes recursively into a number of simpler building blocks. This means that a complex problem potentially provides a rich source of information about problem solving in general. Learning a distribution over plausible values of these building blocks is analogous to the process of *chunking* described in the psychology literature (Miller, 1956). Human short-term memory can only hold a limited number of discrete units (so-called *chunks*) so the more information each chunk holds, the more information can be contained in short-term memory. There is some evidence that the 'better' chunks we form, the better we are at solving problems (Simon, 1974). A Go position in conjunction with the rules of the game contains all the information necessary for perfect play but if we have limited computational resources we must take care to extract only relevant information. The game tree gives the structure required to extract this knowledge and the more we learn from experience about the plausible values of subtrees in the game tree the more efficiently we can search in the future.

This type of supervised learning is unusual because it is the agent itself generating the observations (active learning). The search algorithm in conjunction with the probabilistic model used to guide search gives a generative model of game trees. Since the search algorithm uses observations made in previous searches to learn how to guide the next search (and hence learn to search more efficiently) one might be concerned that we are introducing a systematic bias into the observations. However, as long as we make our inferences based entirely on our generative model (the search algorithm in conjunction with the model) and on which nodes are observed to be TRUE or FALSE we receive the protection of the *likelihood principle* (MacKay, 2003): Given a generative model of data which depends on some parameters, inference about these parameters depends only on the likelihood of the observed data, not on other data which could have been observed but was not. Therefore a systematic bias cannot be introduced by this type of active learning. Another way of thinking about it is that in learning a model of x given an input y, it is allowed to look at the value of y for a datum and choose whether to include the datum in the data set - the inference about $p(x|y)$ will not be biased (MacKay, 1992). The critical thing is that that exactly the same search algorithm is used in training as in testing. If this was not the case then we would be using a different generative model in testing and this way we would indeed potentially introduce a bias.


The Knight's-Move Tesuji

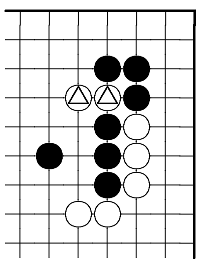

The Nose Tesuji

Figure 4.2: Example tesuji capture problems (Davies, 1975). The goal is to capture the stones marked with a triangle.

4.1 Go Tesuji Problems

We apply the ideas developed in this chapter to the task of solving *tesuji* capture problems where the goal in each case is to capture a particular stone on the board (Davies, 1975) (See Figure 4.2 for two examples). The Japanese word *tesuji* means the best play in a certain local Go position. Tesuji moves often have names such as the 'net', the 'cranes nest' and the 'throw in'. Tesuji problems are used for teaching a player these standard plays.

4.2 Search in Games

Let the set of all possible positions in a game be $\mathcal{S}$. Each position $n \in \mathcal{S}$ has a set of legal moves available, $\mathcal{L}(n)$, each of which generates a *successor* position. Two players, *attacker* and *defender*, take it in turns to move. A problem is defined by its *goal*, $g : \mathcal{S} \to \{\text{TRUE, FALSE, UNKNOWN}\}$. We are concerned with proving whether, starting in a particular position, n, the attacker can reach a state in which the goal is TRUE taking into account all possible actions of the defender. The (Boolean) result of this proof is the Delphic value of the node and is denoted $d(n)$ where $d : \mathcal{S} \to \{\text{TRUE}, \text{FALSE}\}$.

Starting at a root position $r \in \mathcal{S}$ we *develop* it by generating each legal successor position (its *children*). In this way we begin to generate a *search tree*, $\mathcal{T} := \{\mathcal{S}, \mathcal{E}\}$, which represents possible (directed) paths through state space. Each edge $e \in \mathcal{E}$ corresponds to a transition between states (a move). We stick to the conventional term 'tree' here although this is slightly misleading because the possible paths through state space actually correspond to a directed acyclic graph because the same state can be reached via multiple paths. We refer to the set of legal successor positions of a node n as $\text{ch}(n)$, the children of node n. The parent of node c is denoted $\text{pa}(c)$. Once a position is developed it is called an *internal* node otherwise it is a *leaf* node. Leaf nodes, l, where $g(l)$ is TRUE or FALSE are called *terminal* nodes. We iterate the process of developing non-terminal leaf positions to expand the search tree.

4.2.1 Minimax Search

Game trees of this type are usually solved by minimax search. The tree is fully expanded to a fixed number of moves in the future (depth-first search) and the following function is applied recursively to the nodes in the tree (Russell and Norvig, 1995):

$$\text{minmax}(n) = \begin{cases} \max_{c \in \text{ch}(n)} \text{minmax}(c) & \text{if attacker to move and } n \text{ internal} \\ \min_{c \in \text{ch}(n)} \text{minmax}(c) & \text{if defender to move and } n \text{ internal} \\ 1 & \text{if } g(n) = \text{TRUE} \\ -1 & \text{if } g(n) = \text{FALSE} \\ u(n) & \text{if } n \text{ is a leaf node and } g(n) = \text{UNKNOWN} \end{cases}$$

which returns 1 if the goal can be achieved. This is known as a *back-up* routine as information is 'backed up' the tree towards the root. The *evaluation function*, $u(n) \in [0, 1]$, is a game specific heuristic. In practice the alpha-beta pruning algorithm is usually used to ensure only nodes which can actually change the choice of move are developed (Knuth and Moore, 1975). The effectiveness of alpha-beta pruning depends on the ordering of the moves generated. With a perfect move ordering the cost is $O(\sqrt{t})$ where t is the number of nodes in the full minimax tree.

4.2.2 Best-First Search

Using depth as the criterion for terminating search may result in a great deal of wasted computational effort by not concentrating on important lines of play. In this work we focus on best-first search. At each step in a best-first search the most promising node (according to some criterion) is developed. In practice, depth-first search has proved much more successful in game playing applications because choosing which node to expand is difficult. Both depth-first and best-first methods suffer from the *horizon effect*: important lines of play may be terminated before they are played out leading to a poor estimation of the value of the root (Berliner, 1979).

4.2.3 AND / OR Trees

The values of previously explored paths through state space are represented as an AND/OR tree (Nilsson, 1971) which is a graphical representation of a logical function of a set of variables (the leaves of the tree). Each node $n \in \mathcal{S}$ in the AND/OR tree is labeled with a logical value $v(n) \in$ {TRUE, FALSE, UNKNOWN}. If $v(n)$ is TRUE or FALSE then node n is 'solved' and $v(n) = d(n)$. An AND/OR tree has two types of nodes: OR nodes and AND nodes. For a given tree with values assigned to the leaf nodes we determine the values of the internal nodes by:

$$\begin{aligned} \text{AND node: } v(n) &= \bigwedge_{c \in \text{ch}(n)} v(c) \\ \text{OR node: } v(n) &= \bigvee_{c \in \text{ch}(n)} v(c). \end{aligned}$$

The AND operator ($\wedge$) is defined such that if any child of a node is FALSE then the node is FALSE, otherwise if any child is UNKNOWN then the node is UNKNOWN, otherwise it is TRUE. The OR operator ($\vee$) is defined such that if any child of a node is TRUE then the node is TRUE, otherwise if any child is UNKNOWN the node is UNKNOWN, otherwise it is FALSE; see Figure 4.3 for an example tree.

Each AND node corresponds to a position in which it is the defender's turn to move (because every defender response must be considered to prove that the goal can be achieved). Each OR node

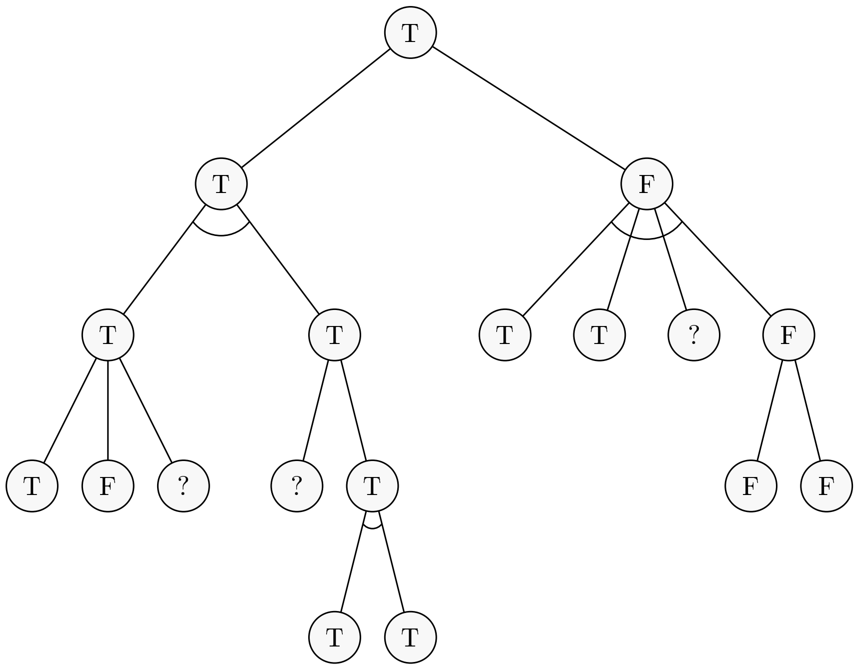


Figure 4.3: And / Or Tree with truth values of nodes labelled. The arcs underneath some of the nodes indicate that they are AND nodes. The other nodes are OR nodes.

corresponds to a position in which it is the attacker's turn to move (because only one working attacker move must be found in each position along the path to the solution). The AND / OR rules correspond to the minimax back-up procedure with a binary valued evaluation function.

If the root has value TRUE or FALSE then the tree is 'solved'. The value of the tree is the value of its root. If a tree has value TRUE it is 'proved', if it has value FALSE it is 'disproved'. If no children can be added to a leaf node l (because no legal moves are available) and the evaluation $g(l)$ = UNKNOWN then it has value FALSE if it is an OR node and TRUE if it is an AND node.

4.2.4 Proof Number Search

We compare our techniques to Proof Number Search (Allis, 1994), a state-of-the-art best-first search algorithm for finding solutions to problems represented as AND/OR trees. Proof Number Search is related to Conspiracy Number search (McAllester, 1988), a best-first search for searching game trees when we have a continuous evaluation function.

In Proof Number Search, two numbers are assigned to each node, n: the proof number, N_n, and the disproof number, D_n. The proof number is defined as the minimum number of nodes that must be developed in order to prove that node TRUE. Therefore, the proof number of an AND node is the sum of the proof numbers of its children (as all of its children must be proved to prove the node). The proof number of an OR node is the minimum of the proof numbers of its children as only one of the children needs to be proved to prove the node. The disproof number is defined as the minimum number of nodes that must be developed in order to disprove the node (prove it FALSE). By symmetry, the rules for propagating disproof numbers are the same as the rules for proof numbers if we exchange OR for

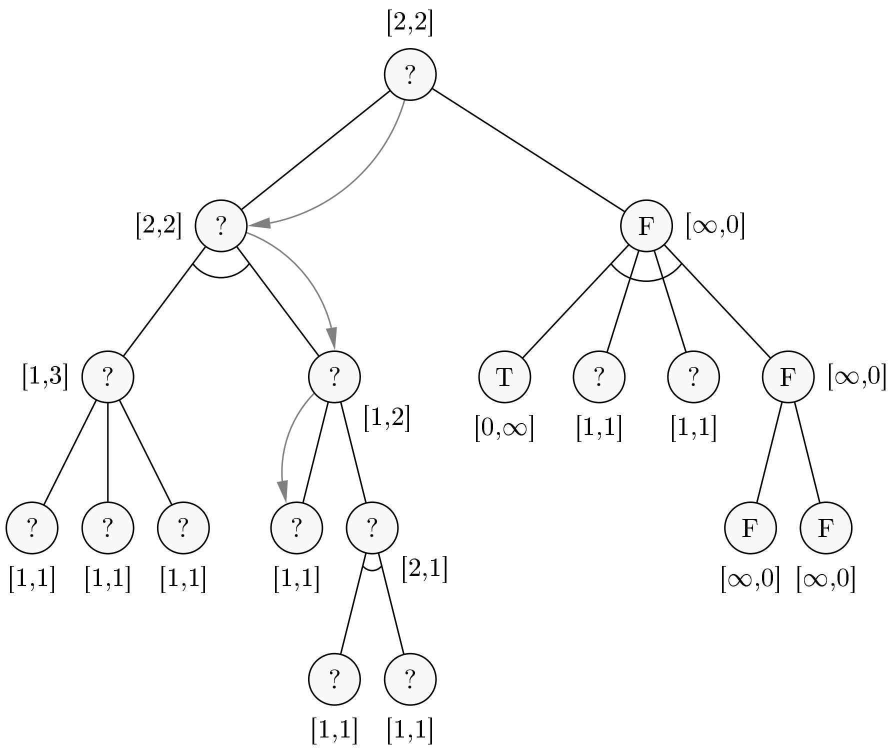


Figure 4.4: And / Or Tree with proof and disproof numbers labelled as $[N_n, D_n]$. The path to the most proving leaf is shown.

AND and TRUE for FALSE. The back-up propagation rules are:

$$N_n = \begin{cases} \sum_{c \in \mathrm{ch}(n)} N_c & \text{if internal AND node,} \\ \min_{c \in \mathrm{ch}(n)} N_c & \text{if internal OR node,} \\ 0 & \text{if } v(n) = \text{TRUE,} \\ \infty & \text{if } v(n) = \text{FALSE.} \\ 1 & \text{if UNKNOWN leaf node} \end{cases}$$

$$D_n = \begin{cases} \min_{c \in \mathrm{ch}(n)} D_c & \text{if internal AND node,} \\ \sum_{c \in \mathrm{ch}(n)} D_c & \text{if internal OR node,} \\ \infty & \text{if } v(n) = \text{TRUE,} \\ 0 & \text{if } v(n) = \text{FALSE.} \\ 1 & \text{if UNKNOWN leaf node} \end{cases}$$

Figure 4.4 shows an AND/OR tree with proof and disproof numbers labeled. Given a search tree with proof numbers and disproof numbers assigned according to the above rules the next node to develop is determined by working down the tree from the root. At each OR node the child node with the lowest proof number is selected and at each AND node the child node with the lowest disproof number is selected. Once a leaf is reached it is developed and then the proof numbers and disproof numbers are propagated up to the root. This process is repeated until the tree is solved.

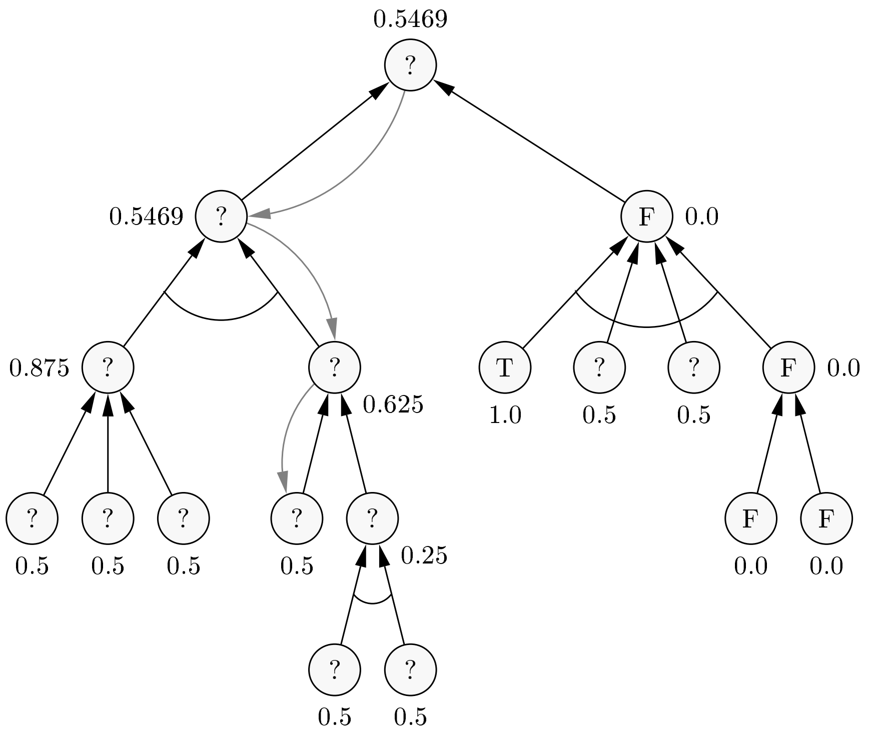


Figure 4.5: Search tree as Bayesian network. Each node is labelled with its probability of being TRUE. The estimated path of best play is also labelled - notice it is the same as the path followed by Proof Number Search (Figure 4.4).

4.3 Search and Inference

In this chapter we use *probabilities* to guide search. For each node, n, we store the probability of it being TRUE: $P_n := P\left(d(n) = \text{TRUE}\right), n \in \mathcal{S}$. If a node has value FALSE the probability $P_n = 0$, if it has value TRUE then $P_n = 1$. If the node is UNKNOWN then the probability represents our degree of belief about the value of the node being TRUE. Under the assumption of independence between the values of the children of a node inference is achieved by simple propagation rules (Pearl, 1984; Chi and Nau, 1988) which we call 'Probability Propagation':

$$\text{AND} : P_n = P\left(\bigwedge_{c \in \text{ch}(n)} d(c)\right) = \prod_{c \in \text{ch}(n)} P_c \tag{4.1}$$

$$\begin{aligned} \text{OR} : P_n &= P\left(\bigvee_{c \in \text{ch}(n)} d(c)\right) = P\left(\neg \bigwedge_{c \in \text{ch}(n)} \neg d(c)\right) \\ &= 1 - \prod_{c \in \text{ch}(n)} (1 - P_c). \end{aligned} \tag{4.2}$$

In other words we treat the game tree as a Bayesian network as shown in Figure 4.5. The joint distribution of the Delphic values of all nodes in the game tree is:

$$p\left(\{d(t)\}_{t \in \mathcal{S}}\right) = \prod_{n \in \mathcal{S} \setminus \mathcal{F}} p\left(d(n) | \{d(c)\}_{c \in \text{ch}(n)}\right) \prod_{l \in \mathcal{F}} p(d(l)) \tag{4.3}$$

where $\mathcal{F}$ is the set of leaves (the search *frontier*). For an AND node, $p(d(n)|\{d(c)\}_{c\in\mathrm{ch}(n)}) = \mathbb{I}(d(n) = \bigwedge_{c\in\mathrm{ch}(n)} d(c))$ and for an OR node $p(d(n)|\{d(c)\}_{c\in\mathrm{ch}(n)}) = \mathbb{I}(d(n) = \bigvee_{c\in\mathrm{ch}(n)} d(c))$. For leaf nodes, l, with $g(l) = \mathrm{TRUE}$ or $g(l) = \mathrm{FALSE}$ the priors on the leaf values, $P(d(l) = \mathrm{TRUE})$ are set to 1 or 0 respectively (the evaluation function). For UNKNOWN leaves the priors represent our prior belief about whether the node is TRUE or FALSE (set to 0.5 in initial experiments).

4.3.1 The Search Tree Factor Graph

A more general perspective can be obtained by representing the AND / OR tree as a factor graph as shown in Figure 4.6. Each node in the AND/OR tree corresponds to a variable node (circle) in the factor graph. The subscript for each factor indicates the parent node of that factor (the node directly above it in the search tree). AND nodes are connected to their children via the AND factor:

$$f_{\wedge,n}\left(d(n), \{d(c)\}_{c\in\mathrm{ch}(n)}\right) = \mathbb{I}\left(d(n) = \bigwedge_{c_i\in\mathrm{ch}(n)} d(c_i)\right),$$

and OR nodes are connected to their children by the OR factor:

$$f_{\vee,n}\left(d(n), \{d(c)\}_{c\in\mathrm{ch}(n)}\right) = \mathbb{I}\left(d(n) = \bigvee_{c_i\in\mathrm{ch}(n)} d(c_i)\right).$$

The factors enforce the AND / OR constraints. The logical value of each node, $d(n)$, is represented as a binary number so TRUE:=1 and FALSE:=0. Bernoulli factors $f_l(l) = \mathrm{Ber}(d(l); q_l)$ are attached to each of the leaves. For UNKNOWN leaf nodes the probability q_l encodes our prior belief about whether the leaf is TRUE or FALSE. If a node, l, is evaluated and found to be FALSE (that is, $g(l) = \mathrm{FALSE}$) then $q_l = 0$, if it evaluates to TRUE then $q_l = 1$. A Bernoulli factor $f_{\mathrm{root}} = \mathrm{Ber}(d(r); q_r)$ can also be attached to the root and this can be used to incorporate an exogenous belief about the probability of the tree being eventually proved true or false. If q_r is set to 0.5 then we recover the same distribution as that represented by the Bayesian network of figure 4.5.

The messages propagated on this factor graph are represented as Bernoulli distributions: $m_{f\to n}(d(n)) = \mathrm{Ber}(d(n); q(n))$. The parameters of the messages to and from node n are denoted $q^{\mathrm{u}}(n)$ for an upward message and $q^{\mathrm{d}}(n)$ for a downward message. Using (2.11) and (2.12) we can calculate the messages up and down the search tree factor graph. The upward messages are calculated as:

$$m_{f_{\wedge,n}\to n}(d(n)) = \mathrm{Ber}\left(d(n); q^{\mathrm{u}}_{\wedge}(n)\right) = \mathrm{Ber}\left(d(n); \prod_{c\in\mathrm{ch}(n)} m_{c\to f_{\wedge,n}}(d(c)=1)\right) \quad (4.4)$$

$$m_{f_{\vee,n}\to n}(d(n)) = \mathrm{Ber}\left(d(n); q^{\mathrm{u}}_{\vee}(n)\right) = \mathrm{Ber}\left(d(n); 1 - \prod_{c\in\mathrm{ch}(n)} \left(1 - m_{c\to f_{\vee,n}}(d(c)=1)\right)\right) \quad (4.5)$$

so

$$q^{u}_{\wedge}(n) = \prod_{c\in\mathrm{ch}(n)} q^{\mathrm{u}}(c) \quad (4.6)$$

$$q^{u}_{\vee}(n) = 1 - \prod_{c\in\mathrm{ch}(n)} (1 - q^{\mathrm{u}}(c)) \quad (4.7)$$

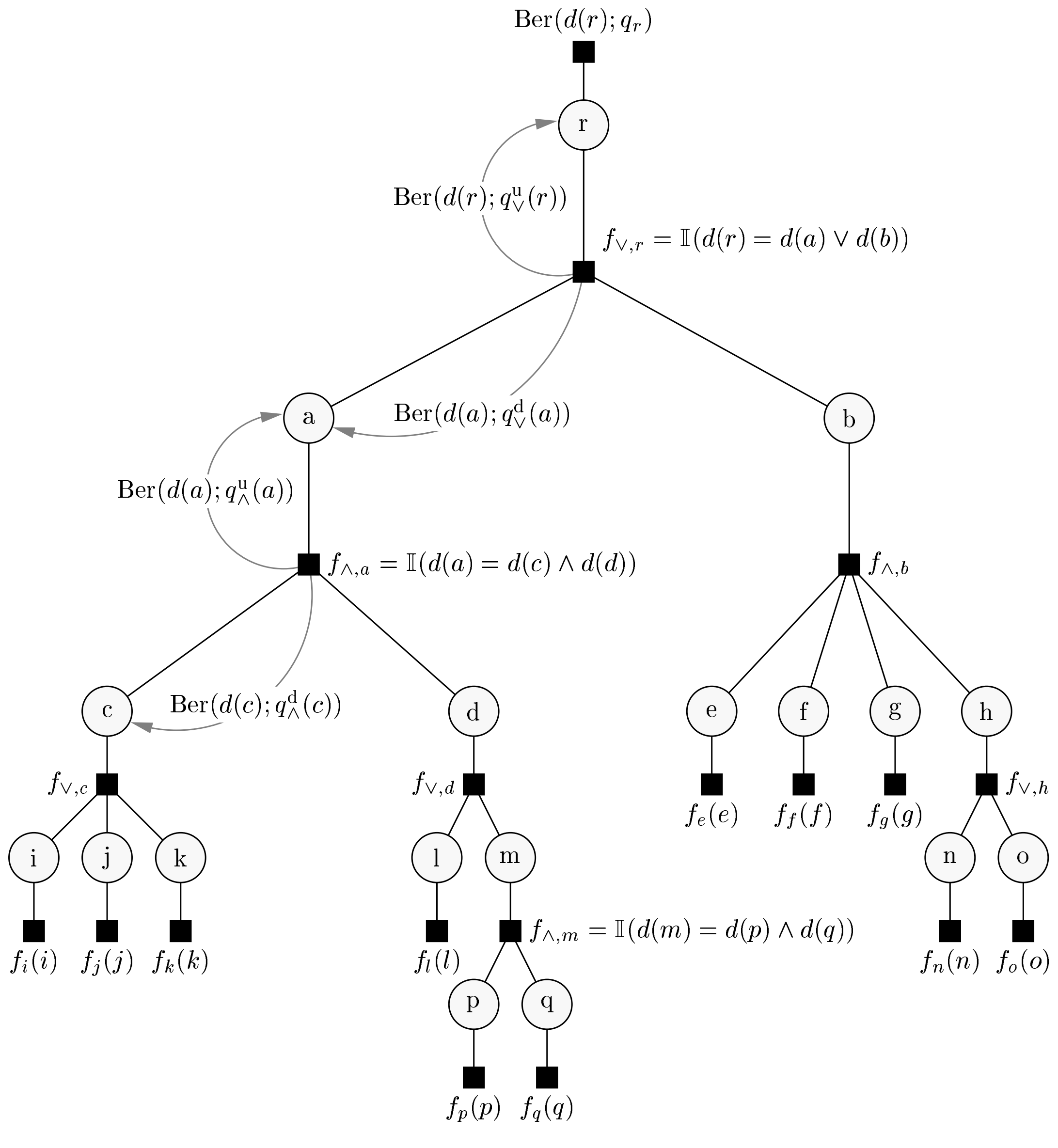

$\mathrm{Ber}(d(r); q_r)$
r
$\mathrm{Ber}(d(r); q^{\mathrm{u}}_{\vee}(r))$
$f_{\vee,r} = \mathbb{I}(d(r) = d(a) \vee d(b))$
a
$\mathrm{Ber}(d(a); q^{\mathrm{d}}_{\vee}(a))$
b
$\mathrm{Ber}(d(a); q^{\mathrm{u}}_{\wedge}(a))$
$f_{\wedge,a} = \mathbb{I}(d(a) = d(c) \wedge d(d))$
$f_{\wedge,b}$
c
$\mathrm{Ber}(d(c); q^{\mathrm{d}}_{\wedge}(c))$
d
e
f
g
h
$f_{\vee,c}$
$f_{\vee,d}$
$f_e(e)$
$f_f(f)$
$f_g(g)$
$f_{\vee,h}$
i
j
k
l
m
n
o
$f_i(i)$
$f_j(j)$
$f_k(k)$
$f_l(l)$
$f_{\wedge,m} = \mathbb{I}(d(m) = d(p) \wedge d(q))$
$f_n(n)$
$f_o(o)$
p
q
$f_p(p)$
$f_q(q)$

Figure 4.6: Search tree as a factor graph. Each variable node corresponds to a node in the search tree. Every internal variable node, x, is connected to a factor: $f_{\vee,x}$ for an OR node and $f_{\wedge,x}$ for an AND node. The factors enforce the AND / OR constraints on the logical values of the nodes.

(compare to equations 4.1 and 4.2). The downward messages are:

$$m_{f_{\wedge,n}\to c}(d(c)) = \text{Ber}\left(d(c); q^{\text{d}}_{\wedge}(c)\right) \tag{4.8}$$

$$m_{f_{\vee,n}\to c}(d(c)) = \text{Ber}\left(d(c); q^{\text{d}}_{\vee}(c)\right) \tag{4.9}$$

where

$$q^{\text{d}}_{\wedge}(c) = \frac{q^{\text{d}}(n)\prod_{c_j\in\text{ch}(n)\backslash c} q^{\text{u}}(c_j) + (1-q^{\text{d}}(n))\left(1-\prod_{c_j\in\text{ch}(n)\backslash c} q^{\text{u}}(c_j)\right)}{q^{\text{d}}(n)\prod_{c_j\in\text{ch}(n)\backslash c} q^{\text{u}}(c_j) + (1-q^{\text{d}}(n))\left(2-\prod_{c_j\in\text{ch}(n)\backslash c} q^{\text{u}}(c_j)\right)} \tag{4.10}$$

and

$$q^{\text{d}}_{\vee}(c) = \frac{q^{\text{d}}(n)}{2q^{\text{d}}(n) - (1-2q^{\text{d}}(n))\prod_{c_j\in\text{ch}(n)\backslash c}(1-q^{\text{u}}(c_j))} \tag{4.11}$$

for $c \in \text{ch}(n)$. If we set q_r (the root bias) to 0.5 then (4.8) and (4.9) are both equal to 0.5 and do not effect the marginal distributions of the variables in the graph so we do not need to calculate them. In this case the propagation rules reduce to (4.4) and (4.5) (c.f (4.1) and (4.2)).

4.3.2 The Algorithm

We now have a system for inferring the distribution over possible final values for the tree given beliefs propagated up from the leaves. A best-first search algorithm also needs a method of selecting the best node to expand next. We adopt the following simple strategy: as in Proof Number Search we start at the top of the tree and work downwards. At each OR node we select the child with the highest marginal probability of being TRUE and at each AND node we select the child with the highest probability of being FALSE. When we reach a leaf node this node is selected as the next node to expand.

The method we use for node selection is to simply follow the path of best play from root to leaf and expand the leaf we reach at the end of this path. This is motivated by Russell and Wefald (1991). They ask the question: why do we search? The answer is: we search in order to improve the quality of our actions, in other words, to 'Do The Right Thing'. As noticed by Palay (1985), there are two ways in which new observations can change the planned sequence of actions: by reducing the value of the current plan or by increasing the value other actions so as to make them preferable to current choices. By following the path of best play we explore the first of these possibilities and try to determine if further exploration of the consequences of our actions will change our current plan by reducing its value. We also attempted the sensitivity analysis approach (in the manner of Rivest (1988)) as it is straightforward to differentiate the probability of the root node with respect to the probability of each leaf, $\frac{dP_r}{dP_l}$, for our model and use this as the criterion for node expansion. This method performed poorly as it penalised depth too much, leading to a breadth-first search.

Fortunately, the upward messages on a factor tree (where 'up' points towards the root of the tree) do not depend on any of the downward messages. Therefore inference can be performed efficiently on the fly using a dynamic programing approach (see algorithms) with cost linear in the depth of the tree. If the parameter q_r (the root bias) is equal to 0.5 (as is the case for the experiments here) then the downward messages are all equal to 0.5 and we do not need to calculate them, thus simplifying the algorithm.

Experiments (See Section 4.5.1 and Figure 4.9) show that Probability Propagation and Proof Number Search must expand roughly the same number of nodes to solve Go problems. Figures 4.4 and 4.5 show an example where Probability Propagation and Proof Number Search both select the same leaf node for expansion. Both methods avoid exploring branches of the tree leading to AND nodes with many children because a proof of such a branch would involve proving more nodes in total. In this way the algorithms strive for quick solutions to problems. Probability Propagation differs from Proof Number Search in that it has an affinity for developing OR nodes with many children because each child of an OR node represents an independent additional chance of finding a proof of the parent. In other words, Probability Propagation tends to explore parts of the search tree where *attacker* has many moves available and *defender* has fewer moves available. Thus Probability Propagation seems to recover a heuristic often used by programmers of traditional game tree search: mobility.

Algorithm 1 Search

```
while $p(\text{root}) < 1.0$ do
  $n = \text{FindBestNode}(\text{root})$
  Develop($n$)
  UpdateBeliefs($n$)
end while
```

Algorithm 2 FindBestNode(n)

```
if $n$ is leaf then
  return $n$
end if
for all $c_i \in \text{ch}(n)$ do
  if $n$ is AND node then
    Calculate $m_{f_n \to c_i}$ via (4.8)
  else
    Calculate $m_{f_n \to c_i}$ via (4.9)
  end if
  Calculate $p(d(c_i)) = \frac{1}{z} \cdot m_{f_n \to c_i} \cdot m_{f_{c_i} \to c_i}$
end for
if $n$ is AND node then
  return FindBestNode($\text{argmin}_{c_i \in \text{ch}(n)} \{p(d(c_i))\}$)
else
  return FindBestNode($\text{argmax}_{c_i \in \text{ch}(n)} \{p(d(c_i))\}$)
end if
```

Algorithm 3 Develop(n)

```
Generate each child, $c \in \text{ch}(n)$, according to $\mathcal{L}(n)$
for all $c \in \text{ch}(n)$ do
  if $g(c)$ TRUE then
    $m_{f_c(c) \to c} = \mathbb{I}(d(c) = 1)$
  else if $g(c)$ FALSE then
    $m_{f_c(c) \to c} = \mathbb{I}(d(c) = 0)$
  else
    $m_{f_c(c) \to c} = \text{Ber}(d(c); q_c)$ (surrogate prior)
  end if
end for
```

Algorithm 4 UpdateBeliefs(n)

```
if $n$ is AND node then
    Calculate $m_{f_n \to n}$ via (4.4)
else
    Calculate $m_{f_n \to n}$ via (4.5)
end if
UpdateBeliefs(pa($n$))
```

4.3.3 Log-Odds Message Representation

To implement the propagation rules in such a way as to avoid difficulties because of limited numerical precision we represent the probabilities as log-odds ratios, $\text{logodds}(p) := L(p) := \ln(\frac{p}{1-p})$. This uses the full floating point range to represent $L(p)$, with high precision at both ends of the range (corresponding to probabilities close to 1 or 0). The log-probability domain is not suitable as it has poor accuracy for probabilities close to 1 which are readily generated by the OR rule in large problems.

Since the AND and OR operations are associative it is possible to perform these calculations efficiently (see Appendix B). If the probability of a node, c, changes during the UpdateBeliefs back-up phase (see Algorithm 4) then the message to the parent, $m_{f_{\text{pa}(c)} \to \text{pa}(c)}$, is calculated by removing the old message using the ANTI-AND or ANTI-OR updates (equations B.14 and B.16) for an AND or OR node respectively and then adding the new message using the AND or OR incremental updates (equations B.13 and B.15). The downward messages can also be incrementally updated in a similar manner using equations B.17 and B.18.

4.3.4 Discussion

Related Work on Best-First Search

A number of best-first search algorithms have been suggested that take into account the shape of the tree for move selection in the case where a continuous evaluation function is used. McAllester (1988) noticed that there are sets of leaves that must all change their value for the value of the root to change. The larger this set of nodes (called a 'conspiracy'), the more nodes will need to be expanded to make the expansion of any one of the member nodes worthwhile. Therefore it is better to expand nodes from smaller conspiracies. The resulting algorithm is a precursor to Proof Number Search. A second method by Rivest (1988) is to use a propagation scheme that approximates minimax by using continuous operators instead of max and min. This means that a sensitivity analysis (simply taking the derivative of the value of the root with respect to each leaf node) can be used to select the best node to expand.

The first practical application of probability distributions to guide best-first search was by Palay (1985). Palay's PB* algorithm extends the B* algorithm of Berliner (1979). In B* the evaluation function is replaced by upper and lower bounds on the value of a state. The algorithm is concerned with proving that one move is better than all the others by ensuring that the lower bound on the value of the resulting position is greater than the upper bounds on all alternatives. This can be achieved by either raising the lower bound on the value of the current best action (called the 'Prove Best' strategy) or by lowering the upper bounds on other available actions (called the 'Disprove Rest' strategy). Palay's PB* algorithm assigns a probability distribution to leaf nodes in the search tree which is based on these bounds. The simplest case would be to let the prior distribution on the value of a node be uniform between the upper and lower bounds on its value as used by B*. A probability distribution over the value of all internal nodes is calculated by the probabilistic form of minimax (i.e. the value, $x(n)$, of a MAX node is distributed according to $x(n) \sim p(\max_c x(c) = x(n)), c \in \mathrm{ch}(n)$.)

Russell and Wefald (1991) consider the principled decision theoretic utility of node expansion and take into account the cost of time which yields a principled criterion for search termination. However, their approach had a high computational cost and made use of an unrealistic assumption that the value of a node is Gaussian distributed. The approach of Baum and Smith (1997) took their work forward and instead stored for each node a discrete distribution over possible values for that node and used a clever, exact propagation scheme.

All of the above approaches assume that a node has some continuous underlying value and the probabilistic algorithms of Palay, Baum and Smith as well as the approach of Russell and Wefald all store probability distributions over this value. We, on the other hand, assume that a node has only a binary underlying 'delphic' value: does the corresponding position lead to the goal being achieved or not?

Probability Propagation

Minimax search has historically relied on the availability of a heuristic evaluation function which returns some quantitative indication of how valuable a game position is and the precise semantics of this value are not usually discussed. The Probability Propagation rules (4.1 and 4.2) were originally proposed by Pearl (1984) for the analysis of minimax search. These rules are the correct backup scheme under a different and somewhat stronger set of assumptions to minimax. The first assumption is that when a move is made both players will gain complete knowledge of the game outcome. This means that the meaning of the evaluation of a position is well defined as the probability of winning under optimal play from that position to the game end. Pearl suggests in the postscript to his book that 'translating' the usual heuristic evaluation function into an estimate of the probability of winning (in [0,1]) and then propagating these values using probability propagation might be a good alternative to minimax for backing up the values in a depth-first search. Chi and Nau (1988) take up this suggestion and compare the performance of minimax with Probability Propagation and they show that results depend on the game. Here, we avoid the use of a real-valued evaluation function but incorporate knowledge into search by the assignment of prior beliefs about the binary TRUE / FALSE value of each leaf. These prior models are surrogate for the as-yet unexplored part of the tree and are learned from data to model the probability that the goal can be achieved from the corresponding position so no ad-hoc 'translation' is needed.

The second assumption implicit in the Probability Propagation rules is that there is no dependence between the outcomes of the leaves. As mentioned by Frank (1994) this assumption can lead to pathological behavior with the probabilities often taking on wildly fluctuating unrealistically high and low values as they are propagated up the tree. We also found this to be the case, hence the need for a representation of the probabilities in the log-odds domain to avoid numerical overflow. The independence assumption seems too strong in the case of 'global' game tree search, where the purpose of search is to find the move which will win the entire game. The notions of being 'ahead' or 'behind' in a game depend on the fact that the values of moves in a given position are correlated: if we have already found several moves leading to victory in a position then we are more likely to find others. However, for local search, this may be less of an issue. If we are interested in proving whether a local goal can be achieved we really must consider each available move as representing an additional

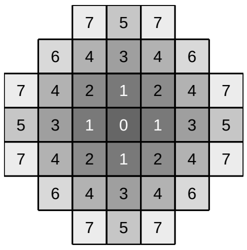


Figure 4.7: The sequence of nested pattern templates T_i with $i \in \{0, \ldots, 7\}$.

independent chance for the player to achieve the goal (or prevent the goal being achieved). At any rate, for the task of solving Go capture problems, Probability Propagation seems empirically to perform similarly to proof-number search (see Section 4.5.1).

4.4 Search and Knowledge

Search is a process of observation. Starting with some prior knowledge, encoded in prior beliefs about whether the available actions will lead to the goal being achieved, the search algorithm explores the state space and updates these beliefs in the light of new evidence. There is a trade-off between search and knowledge: the more accurate the prior beliefs, the less search should be needed to obtain accurate posterior beliefs and so determine the correct action.

Knowledge is combined with the search algorithm described above by the assignment of prior beliefs on the values of the leaf nodes, $p(d(l)) = \mathrm{Ber}(d(l); q_l)$. These prior distributions are surrogate for the as-yet unexplored parts of the tree. When search terminates the parameters of each surrogate prior distribution can be updated according to evidence propagated from the search tree. If L is the set of all of the nodes in all of the searches then the joint likelihood is given by $p(\{d(l)\}_{l \in L} | \mathbf{q}) = \prod_{l \in L} p(d(l)|\mathbf{q})$. The parameters $\mathbf{q}$ are shared across all nodes and all problems so generalisation across different game states and search tasks is possible. Which particular parameter is used for each state depends on domain knowledge and we use the pattern system described in the previous chapter to determine this.

The AND / OR tree recursively breaks down a problem into a set of simpler sub-problems which must be solved first. A complex problem provides a rich source of knowledge about problem solving in general because of all the sub-problems it generates. Indeed, even if the search algorithm were incapable of solving a complex problem in a reasonable time it can still learn from progress made in solving sub-problems encountered on the way.

The search algorithm we have described up to this point is domain independent and could be used for theorem-proving tasks in unspecified applications. For the rest of this chapter we are concerned with Go-specific methods for using domain knowledge to guide search.

4.4.1 Pattern Matching

We saw in the previous chapter that exact local pattern matching gives a rapid and surprisingly accurate Go move predictor. Now we apply this technique to represent knowledge in search. Here we use only 8 pattern template sizes as shown in figure 4.7 in order to improve the speed of pattern matching and it seems unlikely that larger patterns would occur frequently in local problems. In contrast with the pattern definition in Section 3.1.1, in this chapter each pattern vertex has five states (attacker stone, defender stone, empty vertex, off-board, goal stone) and the patterns are not

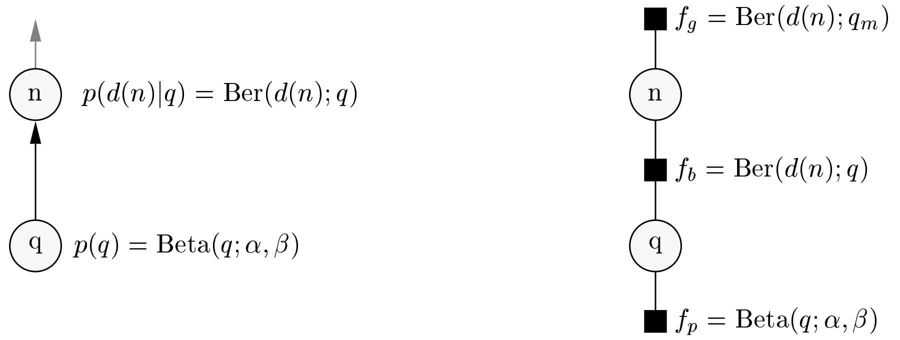

n
$p(d(n)|q) = \mathrm{Ber}(d(n); q)$
q
$p(q) = \mathrm{Beta}(q; \alpha, \beta)$
$f_g = \mathrm{Ber}(d(n); q_m)$
n
$f_b = \mathrm{Ber}(d(n); q)$
q
$f_p = \mathrm{Beta}(q; \alpha, \beta)$

Figure 4.8: Beta Surrogate Tree Model represented as belief network (left) and factor graph (right).

invariant to colour reversal. Nodes in the game tree (positions) are mapped to patterns via the move that generated the node. Let the stack of all patterns that match node n be denoted by $\boldsymbol{\pi}(n)$.

Partial-Transposition Table

Each pattern defines a many-to-one mapping from search tree nodes into a look-up table via the hash key. This table can be viewed as a 'partial-transposition table'. A 'transposition table' is a tool used in most practical game search implementations which contains the values of all board positions that previously appeared in the search so if a position is encountered again the information already gathered about it can be exploited (Plaat *et al.*, 1986). In this work we do not map from full board positions to transposition table entries but instead from *patterns* (partial positions) to table entries. Thus here we effectively use a partial-transposition table. This partial matching allows generalisation across different search tasks but this generalisation is bought at the cost of uncertainty - hence the entries in the partial-transposition table are probability distributions.

We use two pattern tables: one for the defender moves and one for the attacker moves. This means that each pattern is either for an attacker move or a defender moves and learning which moves are good for the defender is considered a separate problem to learning which moves are good for the attacker.

Harvesting

When a node, n, is developed and its children $c \in \mathrm{ch}(n)$ are added to the tree, the patterns matching for the pattern templates of all sizes, $T \in \mathcal{T}$, centered on each move, $m_i \in \mathcal{L}(n)$, which generated each new node, $c \in \mathrm{ch}(n)$, are all *harvested*, that is, added to the pattern table if not already present.

4.4.2 Surrogate Tree Models

Two different models were considered for producing the prior distribution over the truth value of a leaf node. The parameters of these models were mapped to Go positions via the patterns which matched for the move generating the node.

Beta Distribution With Back-Off

The prior distribution on the value of a leaf node, l, is $p(d(l)) = \mathrm{Ber}(d(l); q_l)$. Let $\hat{\pi}(l)$ be the largest pattern (which has been seen at least once before) that matches for leaf node l. This pattern is used to map to the parameter of the prior belief on the value of this node ($q_l := q_{\hat{\pi}(l)}$) so the prior distribution is $p(d(l)) = \text{Ber}(d(l); q_{\hat{\pi}(l)})$. We place a conjugate Beta prior on the parameter of this distribution: $p(q_{\hat{\pi}(l)}) = \text{Beta}(q_{\hat{\pi}(l)}; \alpha_{\hat{\pi}(l)}, \beta_{\hat{\pi}(l)})$. This gives the predictive distribution $p(d(l)|\alpha_{\hat{\pi}(l)}, \beta_{\hat{\pi}(l)}) = \frac{\alpha_{\hat{\pi}(l)}}{\alpha_{\hat{\pi}(l)}+\beta_{\hat{\pi}(l)}}$ and the parameters $\alpha_{\hat{\pi}(l)}$ and $\beta_{\hat{\pi}(l)}$ are *pseudo-counts* corresponding to the number of observed proofs and disproofs respectively of all nodes where $\hat{\pi}(l)$ is found to match. This simple model is represented by the belief network of Figure 4.8 (left). The gray arrow at the top of the directed graph represents the connection to the rest of the search tree model (corresponding to the top factor, f_g, of the equivalent factor graph.

To implement this model, each entry in the pattern table contains the parameters of a Beta distribution. When a pattern is first harvested these parameters are set to $\alpha = \alpha_0$ and $\beta = \alpha_0$. When the node is attached to the tree and its prior distribution is assigned we determine its parameters from the largest matching pattern which has been seen at least once before. If no pattern is found then the prior $\sim \text{Beta}(\alpha_0, \beta_0)$ is returned.

To learn means to calculate the posterior distribution over the parameter: $p(q_n|d(n))$. If we observe a value for $d(n)$ (that is, if the node n becomes proved or disproved) then the posterior is a Beta distribution with the pseudo-counts updated appropriately: that is $p(q_l|d(l))$ =Beta$(q_l; \alpha'_{\hat{\pi}(l)}, \beta'_{\hat{\pi}(l)})$ with $\alpha' = \alpha + 1$ if $d(l) = \text{TRUE}$ and $\beta' = \beta + 1$ if $d(l) = \text{FALSE}$ (see Section 2.3.1).

It is also possible to update the distribution over q after search termination for all nodes in the tree, even if the corresponding node is still UNKNOWN after search. This is because beliefs can still be propagated from the search tree based on the current state of affairs when search was terminated. The beta surrogate prior model can be represented as a factor graph (figure 4.8 right). The messages on this graph are calculated as follows:

$$\begin{aligned}
\text{Message from prior: } m_{f_p \to q}(q) &= \text{Beta}(q; \alpha, \beta) \\
\text{Message up to search graph: } m_{f_b \to n}(d(n)) &= \int \text{Ber}(d(n); q)\text{Beta}(q; \alpha, \beta)\text{d}q \\
&= \text{Ber}\left(d(n); \frac{\alpha}{\alpha + \beta}\right) \\
\text{Message from likelihood (search tree): } m_{f_b \to q}(q) &= \sum_{d(n)} m_{n \to f_b}(d(n)) \cdot f_b(d(n), q) \\
&= \sum_{d(n)} \text{Ber}(d(n); q_m) \cdot \text{Ber}(d(n); q) \\
&= q_m \cdot q + (1 - q_m) \cdot (1 - q)
\end{aligned}$$

To update we need the posterior marginal distribution $p(q)$ which is calculated using (2.13):

$$\begin{aligned}
p(q) &= \frac{1}{Z(\alpha, \beta, q_m)} \{q_m \cdot q + (1 - q_m) \cdot (1 - q)\} \cdot \text{Beta}(q; \alpha, \beta) \\
&\propto \left\{q_m \cdot q^{\alpha+1} \cdot (1 - q)^{\beta} + (1 - q_m) \cdot q^{\alpha} \cdot (1 - q)^{\beta+1}\right\}
\end{aligned}$$

where

$$Z(\alpha, \beta, q_m) = \frac{\alpha}{\alpha + \beta} \cdot (2q_m - 1) + 1 - q_m$$

If $q_m = 0$ or $q_m = 1$ (i.e $d(n)$ is observed as FALSE or TRUE) then this corresponds to the standard posterior pseudo-count update for the Beta distribution (see Section 2.3.1). Otherwise, $p(q)$ can be very closely approximated by finding the beta distribution closest to $p(q)$ in terms of KL divergence:

$$p'(q) = \text{Beta}(q; \alpha', \beta') \tag{4.12}$$

where

$$\alpha', \beta' = \text{argmin}_{a,b} \text{KL}\left(\text{Beta}(a, b) || p(q)\right) \tag{4.13}$$

This is achieved by moment matching and the update equations are derived and presented in Section A.2. Thus we have an ADF online learning scheme.

When the learning update is applied for a move (node), the pattern payloads for the matching patterns for all of the pattern templates T_i are updated according to the above scheme, not just the parameters that were actually used in the search (corresponding to the biggest pattern). If only the largest matching pattern were updated it is unlikely this pattern would be seen in the future so the new knowledge would be lost.

When a new pattern is harvested the prior pseudo-counts are set to the same values as the parent (one size smaller) pattern. The child pattern contains all of the information of the bigger pattern so it makes sense to initialise it to the same distribution. If the parameters were not initialised in this way then harvesting new patterns on the fly could worsen performance as the information already learned about a smaller pattern would be masked by the bigger pattern.

Hierarchical Gaussian Model

We also applied the hierarchical Gaussian pattern prior on the value of a move as described in Section 3.3. Under this model the value y of a move has a Gaussian distribution: $p(y) = \mathcal{N}(y; \mu, \sigma^2)$. In order to obtain the desired distribution over the binary logical value, $d(l)$, of a game tree node we use the conditional distribution to link the two models:

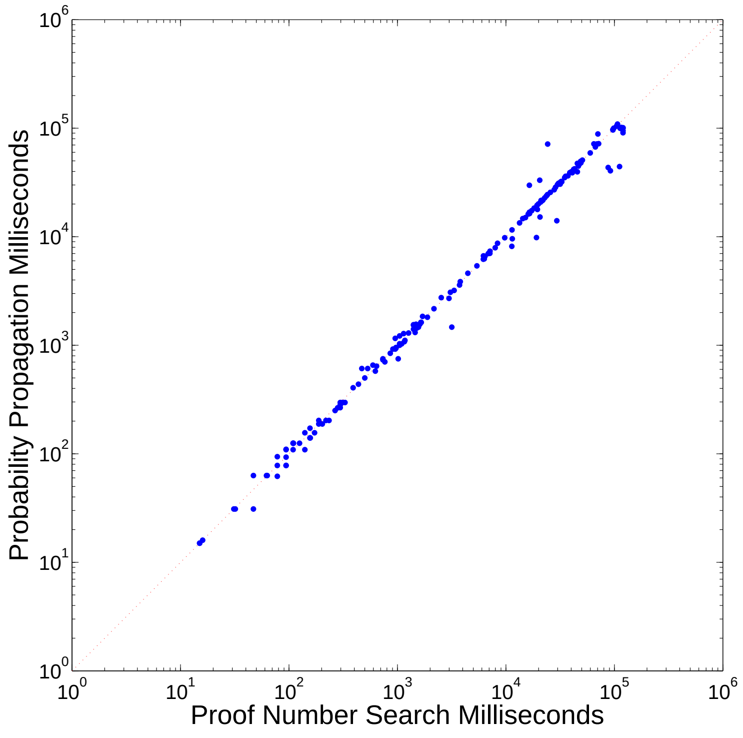
$$p(d(l)|y_l) = \mathbb{I}((y_l > 0) \wedge d(l)) + \mathbb{I}((y_l < 0) \wedge \neg d(l)). \tag{4.14}$$

That is, we observe the constraint that the value of a move generating a TRUE node is greater than zero and the value of a move generating a FALSE node is less than zero. Inference is performed as described in section 2.4.2. Again it is possible to propagate beliefs from the search tree after search termination whether or not the corresponding node was actually solved.

4.5 Experiments

Proof-Number Search and Probability Propagation were applied to the task of solving a set of Go tesuji capture problems *which we know in advance all have a TRUE solution.* The problems we used for these experiments are available at `http://t-t.dk/madlab` thanks to Thomas Thomsen. A disproof criterion was defined based on the liberty count of the goal stone. The goal function was set as

$$g(n) = \begin{cases} \text{TRUE} & \text{if goal vertex empty (the target stone has been captured),} \\ \text{TRUE} & \text{if the goal vertex can be captured in a ladder,} \\ \text{FALSE} & \text{if liberties of goal } > L, \\ \text{UNKNOWN} & \text{otherwise} \end{cases}$$



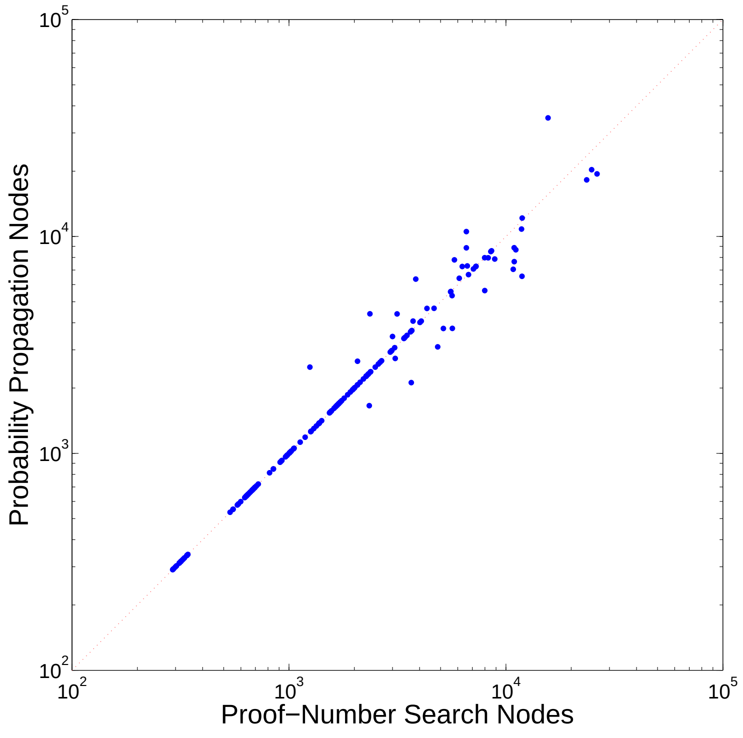


Figure 4.9: Comparing the problem solving speed of Proof-Number Search with Probability Propagation on a set of 190 tesuji Go problems. **Top**: Time taken to solve each problem. **Bottom**: Number of nodes developed in order to solve each problem.

The liberty disproof cut-off, L, was initially set to 1. If search terminates with value FALSE then L was incremented and the search repeated. This process is repeated until the search terminated with value TRUE. Note that we included a ladder solver in the evaluation function which meant the search algorithm did not have to explore the ladders. This greatly improved performance.

4.5.1 No Knowledge

Firstly the performances of Proof-Number Search and Probability Propagation with an ignorant prior ($p(d(l)) = 0.5$, $l \in \mathcal{F}$) were compared on a set of 190 Go capture problems (see Figure 4.9). In most problems the two algorithms perform similarly as discussed in Section 4.3.2. These 190 problems were a subset of our total set of 434 problems. Of the rest of the problems either one or both of the algorithms failed to solve them before time out. Probability propagation could solve 4 problems that Proof Number Search could not solve and Proof Number Search could solve 9 problems that Probability Propagation could not solve.

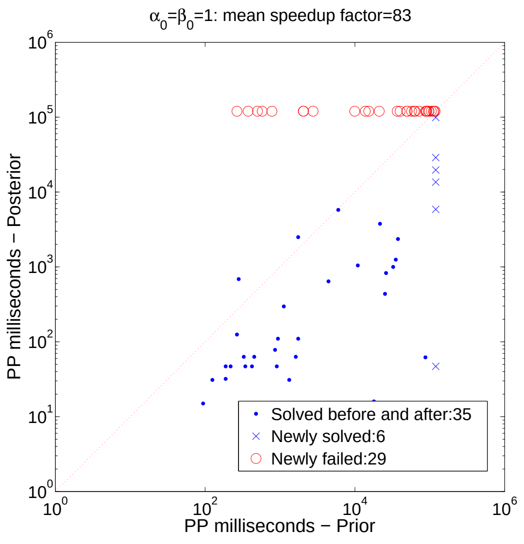
4.5.2 Batch Learning

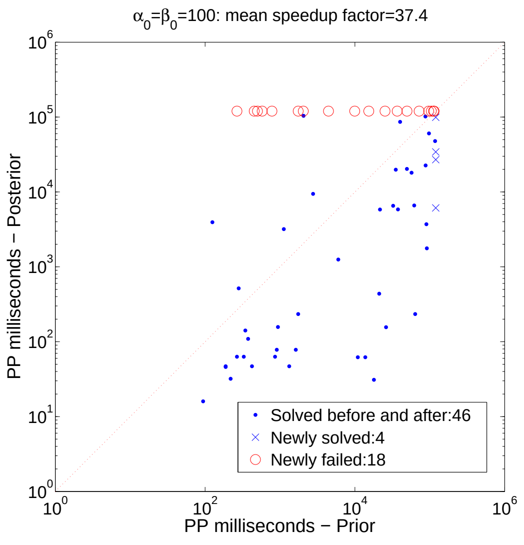
In a second set of experiments we included additional problems that could not be solved by either algorithm before time out. The problems were randomly divided into a training set (289 problems) and a test set (145 problems). The pattern table was initially empty. During learning the searcher harvested patterns continuously into two separate tables: one for attacker moves and one for defender moves. The prior distribution for each new leaf node was assigned using one of the surrogate models described in Section 4.4.2, using the patterns to determine the parameters of the models. Learning was carried out by updating the posteriors over the parameters of the surrogate models after search termination by propagating beliefs to the nodes.

A time-out of 120s was set for each problem. The searcher iterated over the training set re-attempting problems that previously failed until as many problems as possible were solved. The result plots are generated by timing the solver on the test set using the parameters learned by training and separately timing the solver for the prior settings of the parameters. Time is used for comparison so additional cost associated with pattern matching and inference is taken into account. Problems that are solved after learning but could not be solved before learning are labeled as crosses on the plots. Problems that could be solved before learning but not after learning are labeled as circles on the plots. The dots correspond to problems that could be solved both before and after learning. Problems that could be solved neither before or after learning are omitted.

Beta Prior

The Beta model was tested (Figure 4.10) with three different initial settings for the prior parameters (α_0 and β_0). In these experiments the surrogate models were updated after search termination for nodes which were found to be TRUE and FALSE only, not for nodes which remained UNKNOWN. Learning improves the speed of problem solving, often by orders of magnitude. However, there are a number of problems which fail after learning but were able to be solved without learning. This is a cause for concern: the learning actually seems to worsen performance on unseen problems in many cases. This suggests that the Beta-Pattern model as it stands is a poor model of the value of moves in tactical search trees.





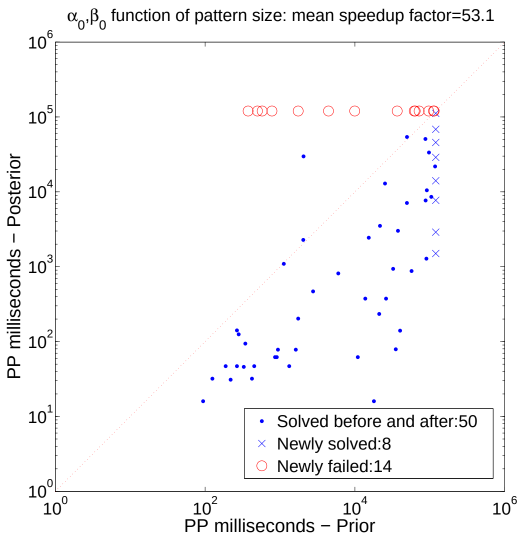


Figure 4.10: Time taken to solve Go problems - Probability Propagation before and after learning. Beta model. **Top**: Prior $\alpha_0 = 1, \beta_0 = 1$ **Middle**: Prior $\alpha_0 = 100, \beta_0 = 100$. **Bottom**: Prior function of pattern size (so as to reduce learning rate of small patterns).

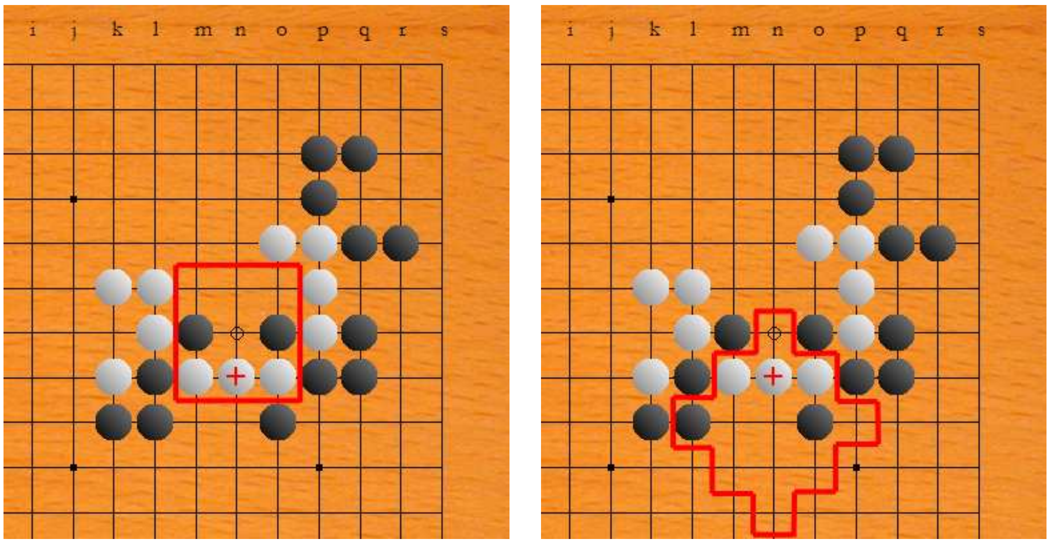


Figure 4.11: Two example patterns matching for a position in a search tree generated by one of the problems in the test set. The goal is to capture the marked chain. **Left**: The correct move, the pattern parameters (learned from problems in the training set) are $\alpha = 1$ and $\beta = 39$. **Right**: An incorrect move, with $\alpha = 15$ and $\beta = 8$ (leading to an incorrect prediction that this is a good move). Note that in these cases important information is ignored because of the small size of the pattern templates.

The reason for this state of affairs is that the patterns that match are frequently small and do not contain enough information to correctly distinguish between good and bad moves because important stones are beyond the pattern template. As an example Figure 4.11 shows two screenshots of a position from one of the search trees (the goal is to capture the chain of three white stones marked with a red cross). The position on the left is labeled with the pattern template corresponding to the largest matching pattern for the correct move to solve this problem. For this pattern, $\alpha = 1$ and $\beta = 39$ so this move is ranked low and therefore not explored until the searcher has wasted a great deal of time on other parts of the search tree. Clearly this decision is based on too little context which does not take into account the surrounding white chains. The right position shows a template corresponding to the largest pattern matching for a wrong move. This pattern has $\alpha = 15$ and $\beta = 8$ and leads the search in this direction. This pattern is also too small to make good predictions. Perhaps it is important to take account of the uncertainty due to the fact that our predictions are based on only small local contexts if this system is to generalise well.

One ad-hoc scheme which we used to attempt to address this issue was to initialise the pseudo-counts of the beta distribution (α_0 and β_0) to higher values which corresponds to setting the prior distribution to be strongly peaked around $p(d(l) = \text{TRUE}) = 0.5$. This slows down learning and makes the probabilities returned by the model close to 0.5. Since we know that Probability Propagation performs similarly to Proof Number Search on tactical problems with a prior setting of $p(d(l)) = 0.5$ this should mean that there are fewer problems which fail after learning. Figure 4.10 (middle) shows the performance of the model for $\alpha_0 = \beta_0 = 100$. Figure 4.10 (bottom) shows the results of using the following setting (where i is the pattern size):

$$\alpha_0 = \beta_0 = \begin{cases} 100,000 \times 10^{-i} & \text{for } i \leq 5 \\ 1 & \text{otherwise.} \end{cases}$$

This has the effect of pinning the small patterns to give the prior $p(d(l)) = 0.5$ but allowing the large patterns to change their values more. This improved the results. However, more principled model improvements were required - hence the adoption of the hierarchical Gaussian prior.

Hierarchical Gaussian Model

The hierarchical Gaussian model performs somewhat better (Figure 4.12). The top plot corresponds to an initial setting of zero previous observations (the backward messages are initialised to be uniform distributions (see Section 3.3). Recall that the hierarchical Gaussian model produces a Gaussian distribution over the continuous value of a node which is then used to produce a distribution over its binary value by observing a sign constraint (equation 4.14). The bottom plot shows the result of an experiment where each time a new leaf node was added to the game tree (and its prior assigned using the hierarchical model) an initial soft pseudo-observation of the value of y_l is added by performing a single update as described in Section 3.3 for a Gaussian belief $p(y_l) = \mathcal{N}(y_l, 0, 0.5)$. This was intended to produce the same effect as initialising the pseudo-counts of the Beta distribution and ensures that if few observations have been made of the large patterns then $p(d(l))$ is close to 0.5.

Learning from Uncertain Nodes

The next set of results (Figure 4.13) compares the effect of three different update schemes (using the Beta Model with $\alpha_0 = \beta_0 = 1$):

1. Learning only from nodes proved TRUE (top).
2. Learning from nodes proved TRUE or FALSE (middle).
3. Learning from all nodes when each search terminates. The approximate posterior is found by (4.13) for the beta model. This corresponds to Assumed-Density Filtering (Minka, 2001) (bottom).

The performance seems to be better the more information we extract from the search trees during training. If only proof information is extracted (the top plot) then the resulting system generally performs extremely poorly, perhaps because the knowledge learned about each pattern is biased towards proof rather than disproof. If disproof information is taken into account then performance is much better (middle plot). The performance is improved slightly more by updating the surrogate models based on evidence propagated to all nodes, even those that are not actually proved or disproved (bottom plot).

4.5.3 Online Learning

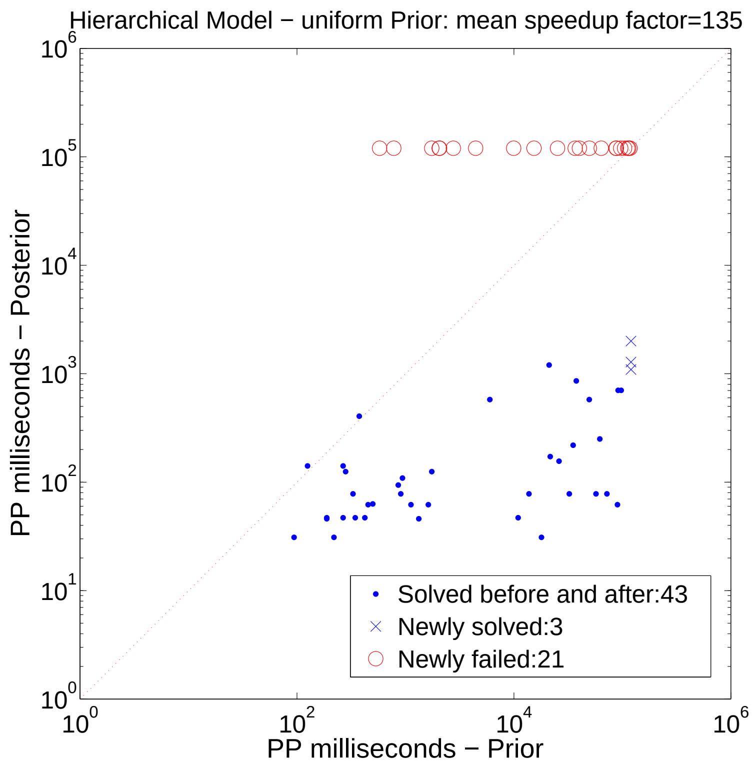
A natural application of this work is an on-line learning setting: the searcher solves (or tries to solve) a set of problems while simultaneously learning. The experiment was set up as follows. 400 Go problems were (roughly) ordered by their difficulty, with more difficult problems at the end and easier problems at the start. The searcher tried to solve each problem. Again the time-out was set at two minutes. When each search terminated (due to a solution being found or due to time-out), beliefs were propagated to the surrogate models and the parameters were updated. The hypothesis is that observations made in earlier (simpler) problems should help in solving later (more difficult) problems and the system should be able to ‘bootstrap’ as learning progresses.

Online learning experiments were carried out for the Beta model and the hierarchical Gaussian model for the best prior settings found in the Batch experiments. The results are shown in Figure 4.14. There is evidence that the searcher can become significantly faster by learning as it solves the problems. However there are a number of problems from the training set that could not be solved despite the shaping effect of starting with simpler problems. Also, as we saw with batch learning, some of the problems could be solved before learning but not after learning.

4.6 Conclusions

Probability Propagation explores a similar number of nodes as Proof-Number Search in order to solve Go tesuji problems. Domain knowledge can be used to improve the speed of search by adding prior distributions on the logical values of the leaves of the search tree. The models which provide these distributions are surrogates for the unexplored parts of the tree. We performed experiments where exact pattern matching (as described in the previous chapter) was used to determine which parameters are shared between these surrogate models depending on the board position local to the move corresponding to the leaf nodes in question. Two of the models introduced in the previous chapter were used: the Beta distribution and the hierarchical Gaussian model. The hierarchical Gaussian model performed better as it is capable of properly taking into account the uncertainty in the value of infrequently seen patterns.

Unfortunately, there were a number of Go problems in the test set which took longer to solve with the learned values for the parameters than with the prior values, even with the hierarchical Gaussian model. This suggests that model improvements are necessary. Perhaps the situation could be improved by the addition of extra pattern features, specific to the search task. A second, and perhaps more serious, cause for concern is the assumption of independence between child values made by the Probability Propagation model. This leads to large, unrealistic, values being taken on for the probabilities of internal game tree nodes and this forced us to adopt a log-odds representation to prevent numerical overflow. In Chapter 6 we will discuss models which turn the problem of game tree search on its head. Rather than considering a deductive model where we wish to infer the probability of the root node being TRUE or FALSE depending on evidence from the leafs, we instead consider models that assume that the values of nodes are determined by an inductive process. That is, the distribution over the value of a node acts as the prior on the values of its children. This alternative view means that correlations between the values of children of nodes can easily be modelled.



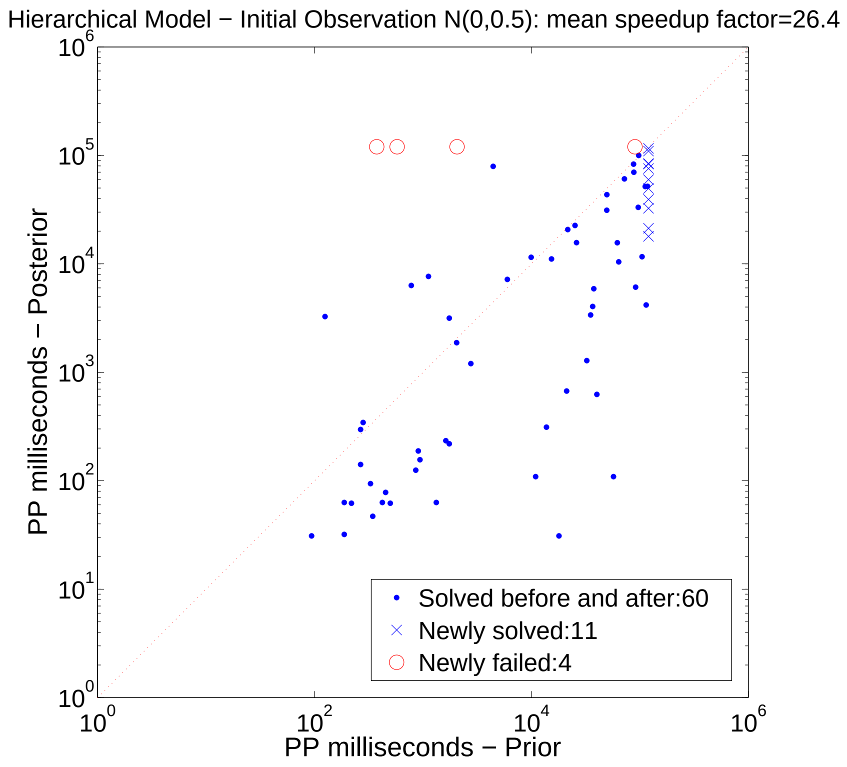
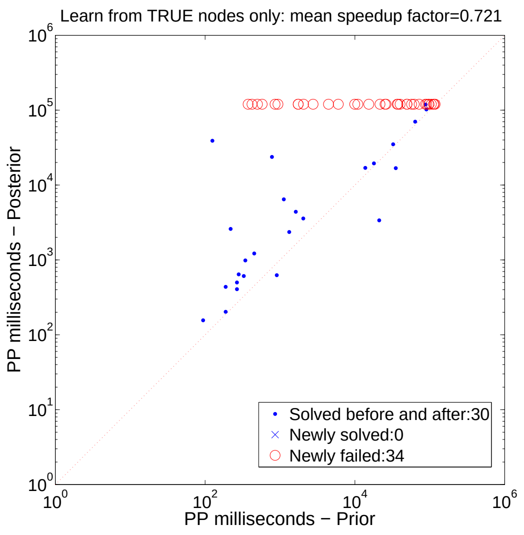
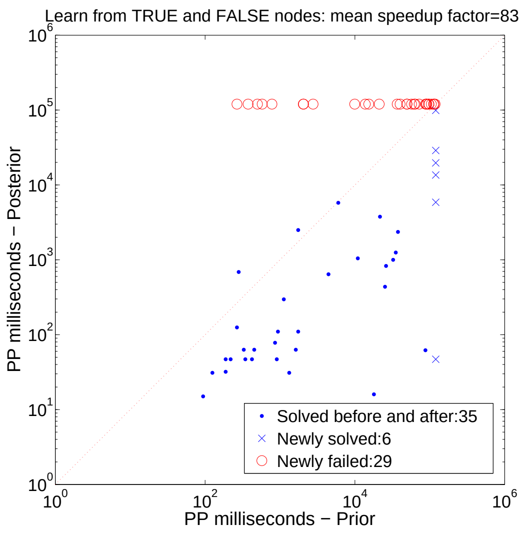


Figure 4.12: Time taken to solve Go problems - Probability Propagation before and after learning. Hierarchical model. **Top**: Uniform prior. **Middle**: Initial pseudo-observation $\sim \mathcal{N}(0, 0.5)$.





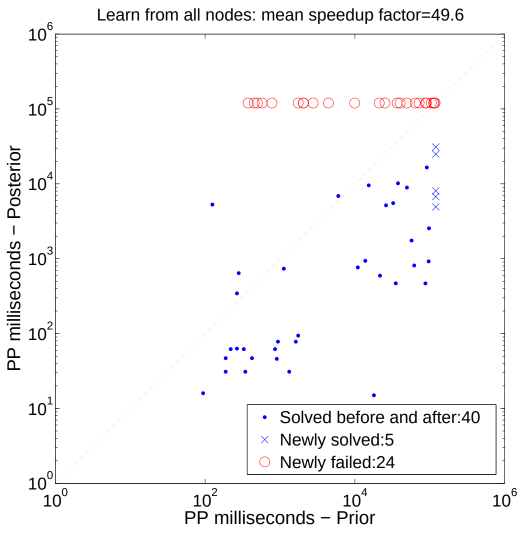
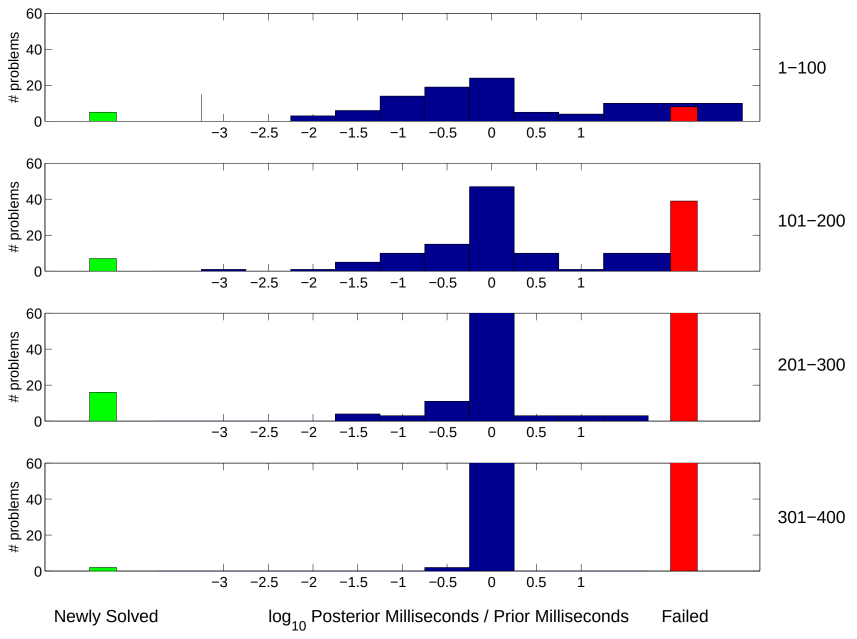


Figure 4.13: Time taken to solve Go problems - Probability Propagation before and after learning. Beta model. prior $\alpha_0 = 1, \beta_0 = 1$ **Top:** Learning from Proved nodes only. **Middle**: Learning from Proved and Disproved nodes. **Bottom**: Learning from all nodes at end of search including UNKNOWN internal nodes.



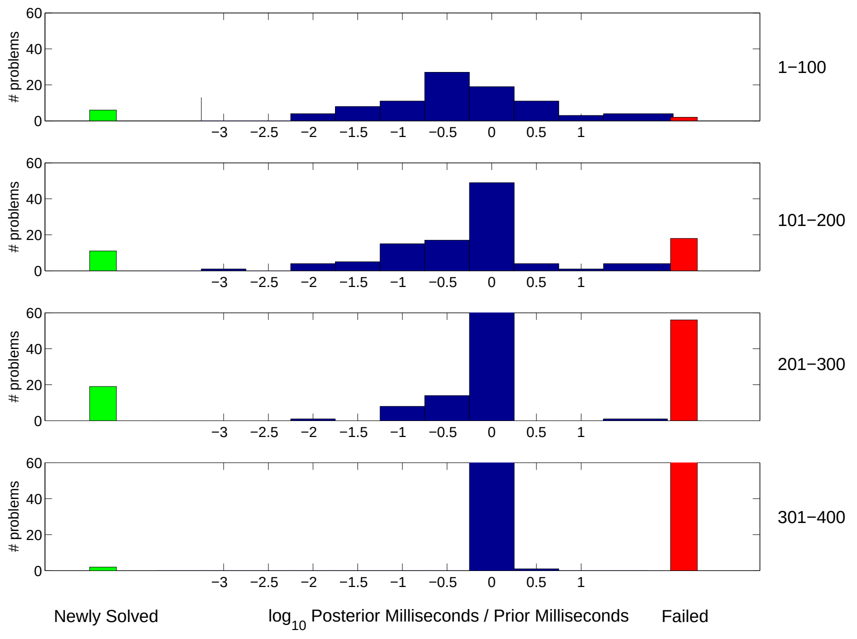


Figure 4.14: Online learning performance. The numbers on the right indicate the subset of the problems each bar graph corresponds to. The problems are grouped in sets of 100 each with each set increasing in difficulty from the previous set. The green bar on the left indicates the number of problems in that subset that could be solved after learning but not before. The red bar on the right indicates the number of problems in that subset that could not be solved, even after learning. The rest of the bars indicate $\log_{10}\frac{\text{posterior time}}{\text{prior time}}$where 'posterior time' is the time needed to solve the problem with the current parameter values (learned from all the problems seen up to this point) and 'prior time' is the time needed with the parameters set to their prior values. For example a value of -2 indicates that the searcher was faster by two orders of magnitude after learning. **Top**: Beta model, prior pseudo-counts function of pattern size (Figure 4.10 (bottom) shows the performance of this model in the batch learning setting). **Bottom**: Hierarchical model with initial pseudo-observation $\sim \mathcal{N}(0, 0.5)$.

Chapter 5

Boltzmann Machines for Territory Prediction[1]

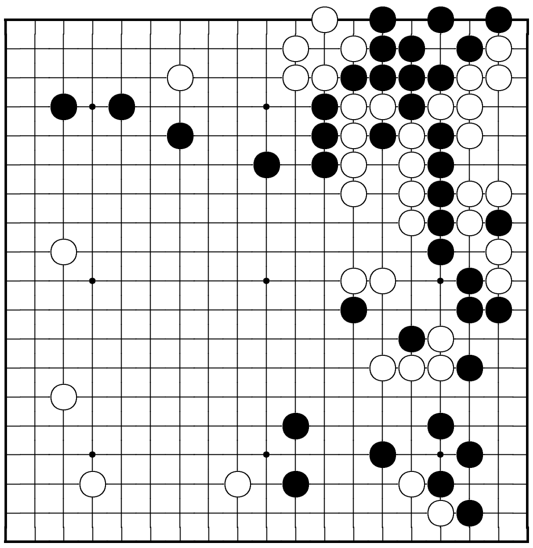


Figure 5.1: Board position from a game at move 80. We wish to model the probability distribution over final territory outcomes given a position such as this one.

We have seen that it is possible to learn from game records how to predict the moves of expert Go players. Game records contain another source of information that we have not yet exploited: the final outcomes of the games. When a game is finished, each point on the board is labeled as belonging to white or black so the game can be scored. This point-wise territory outcome yields a great deal of information about each of the game positions encountered during the game. Firstly, it tells us the life and death status of all stones on the board. If a stone ended up as part of the opponent's territory then that stone must have been captured by the end of the game so must be dead. If a stone ends up

[1]This chapter builds on work done in collaboration with Thore Graepel and David MacKay and was presented at the Neural Information Processing Systems conference (Stern *et al.*, 2004). Iain Murray provided the generalised version of the Swendsen-Wang algorithm.

as part of territory of its own colour then the stone must be alive (that is, not captured). Secondly, the territory outcome provides an observation of the strength of a chain of stones on the board. Go players use the word *thickness* to refer to the extent of the influence exerted by a group because of the combination of its strength and location. If an isolated chain of stones ends up part of a large region of territory then we can say that it is plausible it had a lot of *thickness*. The Japanese word *aji* refers to the influence of a stone because of the uncertainty about its effect in the rest of the game. If a group that appears weak and surrounded by opponent chains goes on to become part of a region of territory of its own colour then it can be said to have had aji. Thirdly, the final territory outcome partitions the board position into a set of *groups*. In Section 1.1.2 we defined a group as a set of chains of stones connected to common eyes. Using the notion of the final territory outcome we can generalise this concept to earlier stages of the game when the stones are not actually yet connected: we define a group as the set of stones within a contiguous region of a single colour in the final territory outcome.

In traditional computer Go programs, the final territory outcome of a game is estimated by first predicting which chains of stones are alive or dead and then determining how much territory each living chain represents (Muller, 2002). The second task may be achieved by an influence function (Zobrist, 1970) which is an iterative method that first assigns positive values to a player's stones and negative values to the opponent's stones and then propagates influence. At each iteration the new value of each point on the board is calculated as its current value plus the number of neighbors of the same sign minus the number of neighbors of the opposite sign. Mathematical morphology has also been applied to the task of estimating territory (Bouzy, 2003). These approaches have been improved on by training a neural network to predict the final territory outcome at each point of board positions taken from expert Go games (van der Werf *et al.*, 2004). This method treats the territory outcome of each board vertex as an independent event which seems simplistic but the system benefits from being able to be automatically tuned from records of expert games.

In contrast to these earlier techniques we wish to obtain a full probability distribution over the territory outcome at each point on the board and we train the parameters of this distribution from positions in game records in conjunction with the true final territory outcomes of those games. This distribution contains a great deal more information than that provided by an influence function because it captures the correlations (both positive and negative) between the outcomes at different points on the board and by properly quantifying the uncertainty about the final territory outcome we open up the possibility of using this system as a component within a larger probabilistic model.

5.1 Territory Prediction

We denote a board configuration by the vector $\mathbf{c} \in \{\text{black,white,empty}\}^{N\times N}$ where each element, $c_{\vec{v}} = c(\vec{v})$, corresponds to the colour of the board at a particular vertex. At the end of a game the point wise territory outcome is determined (as described in Section 1.1.2) and this is denoted by the score vector $\mathbf{s} \in \{+1, -1\}^{N\times N}$ where $s_{\vec{v}} = s(\vec{v})$ so, again, each element corresponds to a board vertex. We score from the point of view of black so elements of $\mathbf{s}$ that have the value $+1$ represent black territory and elements with value -1 represent white territory. Neutral territory is ignored. Here, as discussed in Section 1.1.2, we are using the Chinese method of scoring where the stones of each player as well as the surrounded empty regions count towards their territory. The distribution we model is

$$p(\mathbf{s}|\mathbf{c}),$$

that is, the distribution over final territory outcomes given a particular board position from any stage of the game. This model would have a number of applications as a component within a Go playing program.

5.1.1 Evaluation Function

With a distribution over plausible territory outcomes given the current position we can produce an evaluation function, the simplest being derived from score at the end of the game which, using Chinese scoring from the point of view of black, is given by $S_{\text{black}} = \sum_{\vec{v} \in \mathcal{N}} s_{\vec{v}}$. One possible evaluation function would be the expected score:

$$u_{\text{score}}(\mathbf{c}) = \langle S_{\text{black}} \rangle_{p(\mathbf{s}|\mathbf{c})} = \left\langle \sum_{\vec{v} \in \mathcal{N}} s_{\vec{v}} \right\rangle_{p(\mathbf{s}|\mathbf{c})} .$$

Another evaluation function that is probably better (if we assume, as strong human players do, a win by 1 point is just as good as a win by 10 points) is given by the probability of winning from the current position:

$$u_{\text{win}}(\mathbf{c}) = p\left(\left(\sum_{\vec{v} \in \mathcal{N}} s_{\vec{v}} \right) > 0 \right).$$

5.1.2 Connectivity

As explained in Section 1.1.2 the fate of stones depends on the other stones they are connected to and ultimately it is determined by whether a group of connected stones can become connected to two eyes. The connectivity, $C(\vec{v}_1, \vec{v}_2, \mathbf{c})$, between two locations on the board could be estimated by the correlation between the territory outcomes at the points in question:

$$C(\vec{v}_1, \vec{v}_2, \mathbf{c}) = \frac{\langle (s_{\vec{v}_1} - \langle s_{\vec{v}_1} \rangle)(s_{\vec{v}_2} - \langle s_{\vec{v}_2} \rangle) \rangle}{\sqrt{\left\langle (s_{\vec{v}_1} - \langle s_{\vec{v}_1} \rangle)^2 \right\rangle \left\langle (s_{\vec{v}_2} - \langle s_{\vec{v}_2} \rangle)^2 \right\rangle}}$$

where the expectations are taken under the distribution $p(\mathbf{s}|\mathbf{c})$. Determining which stones are connected would be useful for building a structured representation of the board and for solving life and death problems.

It would also be useful to determine for which pairs of vertices the function $C(\vec{v}_1, \vec{v}_2, \mathbf{c})$ takes on a negative value. The Japanese refer to this as *miai* meaning situations where a player has two mutually exclusive options and if one player takes one option the other player will take the other.

5.1.3 Group Safety

The safety of a group of stones on the board can be modeled using the distribution over territory outcomes. If we denote a group of interest by $\hat{\mathcal{N}} \subset \mathcal{N}$ and let the vector $\hat{\mathbf{s}}$ represent the territory outcome for each vertex in $\hat{\mathcal{N}}$ then a model of the safety of each element of this group for black is given by the marginal distribution

$$p(\hat{\mathbf{s}}|\mathbf{c}) = \sum_{\{s_{\vec{v}} : \vec{v} \in \mathcal{N} \setminus \hat{\mathcal{N}}\}} p(\mathbf{s}|\mathbf{c}).$$

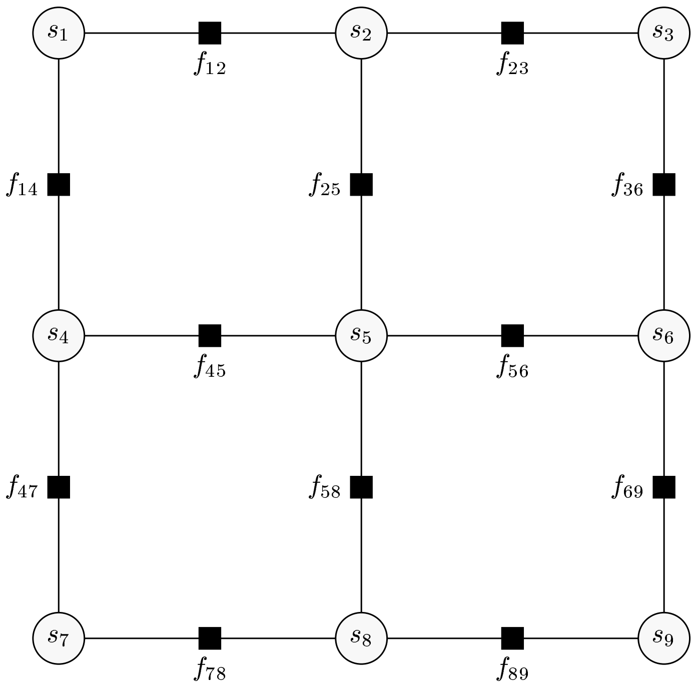


Figure 5.2: Factor graph for the Boltzmann territory prediction model for a 3×3 Go board. This picture does not show the current board position $\mathbf{c}$.

5.2 Model

5.2.1 Framework

The model we propose for territory prediction is a conditional Markov random field (Lafferty *et al.*, 2001) with the grid topology of the Go board. In other words the model factorises over factors f_{ij} each of which connects two adjacent vertices of the board. The distribution is given by

$$p(\mathbf{s}|\mathbf{c}) = \frac{1}{Z(\mathbf{c},\theta)} \prod_{(i,j)\in\mathcal{E}} f_{ij}(s_i, s_j, \mathbf{c}, \theta),$$

where $\mathcal{E}$ is the set of edges in the graph corresponding to the Go board (as defined in Section 1.1.2). The factor graph corresponding to this model is shown in Figure 5.2. This model is equivalent to an Ising model where each spin corresponds to the territory outcome at a board vertex and whose couplings and biases are dependent on the board position.

The structure of this model is inspired by Go knowledge. In many situations, the fate of Go stones appears to obey the Markov property along the topology of the board as assumed by the model: stone A is connected to stone C because stone A is connected to B which is connected to C. Edge effects are incorporated to some degree by the fact that points on the edge of the board have less connections than those in the center. Also, if we ignore the effect of the board edge, properties of Go stones are invariant with respect to translation across the board. We make this assumption so the same parameters θ are shared by all factors. If we wish to model the effect of board location we could make θ a function of the board location. In addition this model cannot model non-linear effects that would be taken account of by a model with factors that covered larger regions of the board.

5.2.2 Boltzmann5

The simplest version of the model has factors that decompose into 'coupling' and 'bias' terms as follows:

$$f_{ij}(c_i, c_j, s_i, s_j, \phi) = \exp(w(c_i, c_j)s_i s_j + h(c_i)s_i + h(c_j)s_j).$$

The coupling $w(c_i, c_j)$ between territory outcome nodes, s_i and s_j, depends on the state of the board at the two vertices. The bias for each vertex $h(c_i)$ depends only on the state of the board at the point in question. We assume that Go positions and their territory outcomes are symmetric with respect to colour reversal so $f_{ij}(c_i, c_j, s_i, s_j, \phi) = f_{ij}(-c_i, -c_j, -s_i, -s_j, \phi)$ and also to direction reversal so $f_{ij}(c_i, c_j, s_i, s_j, \phi) = f_{ij}(c_j, c_i, s_j, s_i, \phi)$. If these symmetries are taken into account the model has 5 free parameters:

- $w_{\text{chains}} = w(\text{black}, \text{black}) = w(\text{white}, \text{white})$,
- $w_{\text{inter-chain}} = w(\text{black}, \text{white}) = w(\text{white}, \text{black})$,
- $w_{\text{chain-empty}} = w(\text{empty}, \text{white}) = w(\text{empty}, \text{black}) = w(\text{white}, \text{empty}) = w(\text{black}, \text{empty})$,
- $w_{\text{empty}} = w(\text{empty}, \text{empty})$,
- $h_{\text{stones}} = h(\text{black}) = -h(\text{white})$,

and $h(\text{empty})$ is set to zero by symmetry. We will refer to this model as *Boltzmann5*. This simple model is interesting because all these parameters can be interpreted. For example we would expect w_{chains} to take on a large positive value to give us the 'common fate' property of chains, that is the stones in a chain are all captured together or not at all.

5.2.3 Boltzmann12

A more general model has factors of the form

$$f_{ij}(c_i, c_j, s_i, s_j, \phi) = \exp(w(c_i, c_j, s_i, s_j)).$$

If we take account of the symmetries mentioned above we are left with 12 free parameters so this model is referred to as *Boltzmann12*.

5.2.4 BoltzmannLiberties

As stated by Lafferty *et al.* (2001) adopting a discriminative approach (by conditioning on the board position rather than modelling it fully) allows us to relax the conditional independence assumptions of the Markov random field without increasing the complexity of inference. One way this is possible for the task at hand is by making each factor a function of features whose values are determined by non-local aspects of the board position. As we have seen, the number of liberties of a chain is a useful heuristic for predicting whether it will live or die, so the next version of the model we will consider takes into account the liberty counts of the chains present at the two locations that each factor links. The factors are

$$f_{ij}(s_i, s_j, \mathbf{c}, \phi) = \exp(w(c_i, c_j, s_i, s_j, l_i, l_j))$$

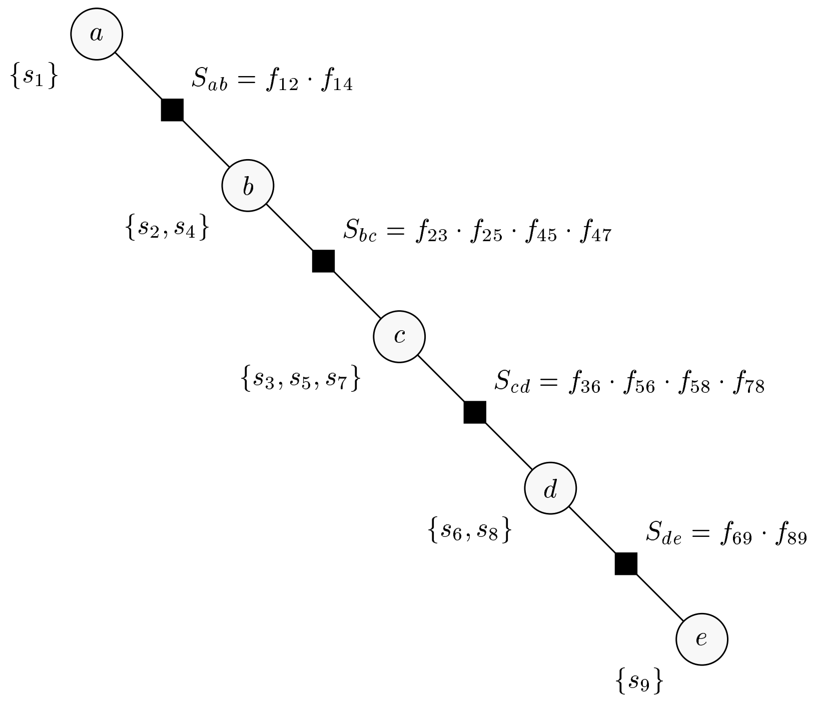

a
$\{s_1\}$
$S_{ab} = f_{12} \cdot f_{14}$
b
$\{s_2, s_4\}$
$S_{bc} = f_{23} \cdot f_{25} \cdot f_{45} \cdot f_{47}$
c
$\{s_3, s_5, s_7\}$
$S_{cd} = f_{36} \cdot f_{56} \cdot f_{58} \cdot f_{78}$
d
$\{s_6, s_8\}$
$S_{de} = f_{69} \cdot f_{89}$
e
$\{s_9\}$

Figure 5.3: Diagonal cluster graph for the 3 by 3 Go board example.

where $l_i \in \{1, 2, 3, \geq 4\}$ is the number of liberties of the chain at vertex i. Again we can take account of the symetries of Go positions which leaves this model with 78 free parameters. This model will be referred to as *BoltzmannLiberties*.

5.3 Inference

We use inference to calculate the marginal distribution over the territory outcome at a single board vertex, $p(s_i|\mathbf{c})$, or a pair of vertices, $p(s_i, s_j|\mathbf{c})$. This can be achieved by a number of methods.

5.3.1 Exact Inference

The marginal distributions over the territory outcome at each vertex, $p(s_i)$, can be calculated by variable elimination (see Section 2.1.1). The cost of this operation strongly depends on the order in which variables are eliminated. For a grid the best order in which to sum out the variables is diagonal by diagonal. This can be illustrated by redrawing the factor graph for the model where we combine some of the variables together into 'clusters' (see Figure 5.3). The composite factors are known as 'separators' because they separate the variables into two sets. This 'cluster graph' has no cycles so we can cast the variable elimination task in terms of the sum-product algorithm. In that case the most expensive operation is calculating the messages from the central cluster such as (for our 3 by 3 Go board example):

$$m_{S_{cd} \to d}(s_6, s_8) = \sum_{s_3} \sum_{s_5} \sum_{s_7} f_{36} f_{56} f_{58} f_{78} \cdot m_{c \to S_{cd}}(s_3, s_5, s_7).$$

Fortunately, the diagonal elimination order means there is only one cluster of size 3 corresponding to the central diagonal and all of the other clusters are smaller. Therefore the expensive message needs to be calculated only twice (once for the downward pass and once for the upward pass). Once the marginal distribution is calculated for each cluster, the marginal distributions for the original variables can be calculated by summing out the other variables in the cluster. For the full size Go board the largest cluster is the full diagonal containing 19 variables.

An alternative method for determining the clusters and performing the variable elimination for a given elimination order is provided by Cozman (2000). The algorithm proceeds by

1. Placing all factors in the graph into a pool.
2. For each variable, v_i, according to the elimination order
   (a) Create a 'bucket', B_i.
   (b) Remove all densities containing v_i from the pool and place them in B_i.
   (c) Multiply all the densities in B_i to create the 'cluster' C_i and store this in the bucket.
   (d) Sum out v_i from C_i to create the 'separator' S_i and add this separator to the pool of densities.

After this process the pool will contain one separator density which when renormalised is the marginal density of the last variable to be eliminated. A reverse update pass through the buckets can then be applied which corrects the densities of each cluster so they can be used to query every marginal in the graph without repeating the above calculation (Cozman, 2000). To calculate the partition function the sum calculated by this algorithm for the elimination order $s_1,s_2,s_4,s_3,s_5,s_7,s_6,s_8,s_9$ for our 3 by 3 Go board example is

$$Z(\mathbf{c},\theta) \propto \sum_{s_9}\sum_{s_8} f_{89} \sum_{s_6} f_{69} \sum_{s_7} f_{78} \sum_{s_5} f_{56}f_{58} \sum_{s_3} f_{36} \sum_{s_4} f_{45}f_{47} \sum_{s_2} f_{25}f_{23} \sum_{s_1} f_{12}f_{14}$$

whereas the sum calculated by the sum-product algorithm on the cluster graph of figure 5.3 is given by

$$Z(\mathbf{c},\theta) \propto \sum_{s_9}\sum_{s_8}\sum_{s_6} f_{89}f_{69} \sum_{s_7}\sum_{s_5}\sum_{s_3} f_{78}f_{56}f_{58}f_{36} \sum_{s_4}\sum_{s_2} f_{45}f_{47}f_{25}f_{23} \sum_{s_1} f_{12}f_{14}$$

which is slightly less efficient (if implemented naively). We used the Cozman approach for the experiments here and all the marginals for a 19 by 19 Go board can be calculated in 10 minutes on a 2GHz computer using the diagonal elimination order. This is too slow for use in a playing Go program but fast enough to use as a benchmark to compare our other inference methods to.

5.3.2 Loopy Belief Propagation

An alternative to performing exact inference is to run the sum-product algorithm directly on the loopy graph (Figure 5.2) with the messages being repeatedly passed around the loops in the graph. The incoming messages to each node are stored and these are updated in an asynchronous fashion by repeatedly sweeping over the graph. The update is 'damped' by updating the messages according to

$$m^{\text{new}}_{f\to n}(n=a) = (1-\lambda)\cdot m^{\text{full}}_{f\to n}(n=a) + \lambda\cdot m^{\text{old}}_{f\to n}(n=a)$$

where $m^{\text{full}}_{f\to n}$ is the message calculated by the sum-product algorithm (equations 2.12 and 2.11). This damping is included to prevent oscillations (Heskes, 2003) and for our experiments a setting of $\lambda = 0.5$ seemed to work well. The approximate marginals for all the nodes in a 19 by 19 Go board can be calculated in about 0.5 seconds using loopy belief propagation on a 2GHz computer.

5.3.3 Gibbs Sampling

The standard method for performing inference on Boltzmann machines is Gibbs sampling. See MacKay (2003) for a tutorial. We generate a Markov chain of samples, each one generated by changing the state of a single territory node, s_i, according to the probability

$$p(s_i = 1|\,\mathbf{s}\setminus s_i\,, \mathbf{c}) = \frac{\prod_{f_{ik}\in\mathcal{F}_{s_i}} f_{ik}(s_i = 1, s_k, \mathbf{c})}{\prod_{f_{ik}\in\mathcal{F}_{s_i}} f_{ik}(s_i = 1, s_k, \mathbf{c}) + \prod_{f_{ik}\in\mathcal{F}_{s_i}} f_{ik}(s_i = 0, s_k, \mathbf{c})}, \tag{5.1}$$

where $\mathbf{s}$ is the state of all the territory nodes in the previous sample. At each stage the next node to be flipped is selected uniformly at random. The first 1,000,000 such samples are disregarded as they are assumed to be influenced by the initial setting of the states of the variables (burn in). Expectations of single variables under the distribution $p(\mathbf{s}|\mathbf{c})$ can then be estimated by

$$\langle s_i \rangle_{p(\mathbf{s}|\mathbf{c})} = \frac{1}{n}\sum_{t=1}^{n}\left\{p\left(s_i^t = 1|\mathbf{s}^t \setminus s_i^t, \mathbf{c}\right) - p\left(s_i^t = -1|\mathbf{s}^t \setminus s_i^t, \mathbf{c}\right)\right\}$$

where t is an index over samples and n is the total number of samples (5,000,000). Note that using the conditional probabilities in the expectation calculation greatly reduces the number of samples needed to obtain an accurate estimate when compared to just taking an average over samples.

5.3.4 Swendsen-Wang Sampling[2]

We will see that the coupling between territory outcome nodes is very strong if the corresponding board locations are part of the same chain of stones because of the common fate property of chains. This means that the probability of changing the state of a single node in a chain conditional on the other nodes according to (5.1) can be very small so it may take a long time before the chain of samples correctly represents the distribution $p(\mathbf{s}|\mathbf{c})$. In other words, Gibbs sampling mixes very poorly.

The situation can be greatly improved by increasing the complexity of the model in such a way as to not change the marginal distribution of the nodes we are interested in while allowing much more efficient sampling. Following Edwards and Sokal (1988) who devised this generalisation of the original process of Swendsen and Wang (1987), we attach an extra binary 'bond' variable, h_{ij}, to each factor as shown in Figure 5.4. The new factors g_{ij} are such that if the bond h_{ij} is set then the variables s_i and s_j must be in the same state if the coupling between them is positive or in the opposite state if the coupling between them is negative. In this way the bonds have the effect of grouping the variables into ferromagnetic (positive coupling) and anti-ferromagnetic (negative coupling) *clusters*. Let the vector of bond variables be $\mathbf{h}$. Gibbs sampling proceeds by alternately sampling from $p(\mathbf{h}|\mathbf{s}, \mathbf{c})$ and $p(\mathbf{s}|\mathbf{h}, \mathbf{c})$.

[2]Special thanks is due to Iain Murray for suggesting the generalised version of the Swendsen-Wang algorithm.

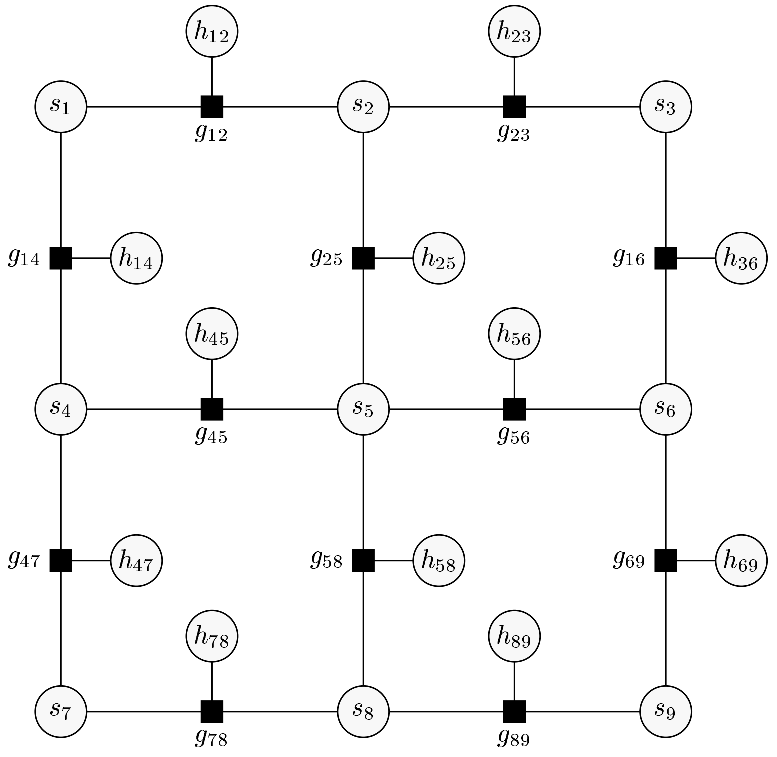


Figure 5.4: Factor graph for the territory prediction model with Swendsen-Wang bond variables, h_{ij}.

The Factors

In order to produce the correct distribution we must choose the factors g_{ij} such that if we marginalise out the bond variables we obtain the original distribution:

$$\sum_{\mathbf{h}} \frac{1}{Z'(\mathbf{c},\theta)} \prod_{(i,j)\in\mathcal{E}} g_{ij}(s_i, s_j, h_{ij}, \mathbf{c}, \theta) = \frac{1}{Z(\mathbf{c},\theta)} \prod_{(i,j)\in\mathcal{E}} f_{ij}(s_i, s_j, \mathbf{c}, \theta). \tag{5.2}$$

Define the 'coupling', $K_{ij} = f_{ij}(1,1) \times f_{ij}(-1,-1) - f_{ij}(-1,1) \times f_{ij}(1,-1)$ for the original factors $f_{ij}(s_i, s_j)$. If the coupling $K_{ij} > 0$ then the factors g_{ij} are defined as

$$g_{ij} = \left\{ \begin{array}{ccccc} & \multicolumn{2}{c}{[h_{ij} = \text{'coupled'}]} & \multicolumn{2}{c}{[h_{ij} = \text{'not coupled'}]} \\ & [s_i = -1] & [s_i = 1] & [s_i = -1] & [s_i = 1] \\ [s_j = -1] & (1 - x_{ij})a_{ij}b_{ij} & 0 & x_{ij}a_{ij}b_{ij} & x_{ij}A_{ij}b_{ij} \\ [s_j = 1] & 0 & (1 - x_{ij})A_{ij}B_{ij} & x_{ij}a_{ij}B_{ij} & x_{ij}A_{ij}B_{ij} \end{array} \right.$$

and if the coupling is negative, that is, $K_{ij} < 0$ then

$$g_{ij} = \left\{ \begin{array}{ccccc} & \multicolumn{2}{c}{[h_{ij} = \text{'coupled'}]} & \multicolumn{2}{c}{[h_{ij} = \text{'not coupled'}]} \\ & [s_i = -1] & [s_i = 1] & [s_i = -1] & [s_i = 1] \\ [s_j = -1] & 0 & (1 - y_{ij})C_{ij}d_{ij} & y_{ij}c_{ij}d_{ij} & y_{ij}C_{ij}d_{ij} \\ [s_j = 1] & (1 - y_{ij})c_{ij}D_{ij} & 0 & y_{ij}c_{ij}D_{ij} & y_{ij}C_{ij}D_{ij}. \end{array} \right.$$

To satisfy equation 5.2 we have

$$g_{ij}(s_i, s_j, h_{ij} = \text{'coupled'}) + g_{ij}(s_i, s_j, h_{ij} = \text{'not coupled'}) \propto f_{ij}(s_i, s_j)$$

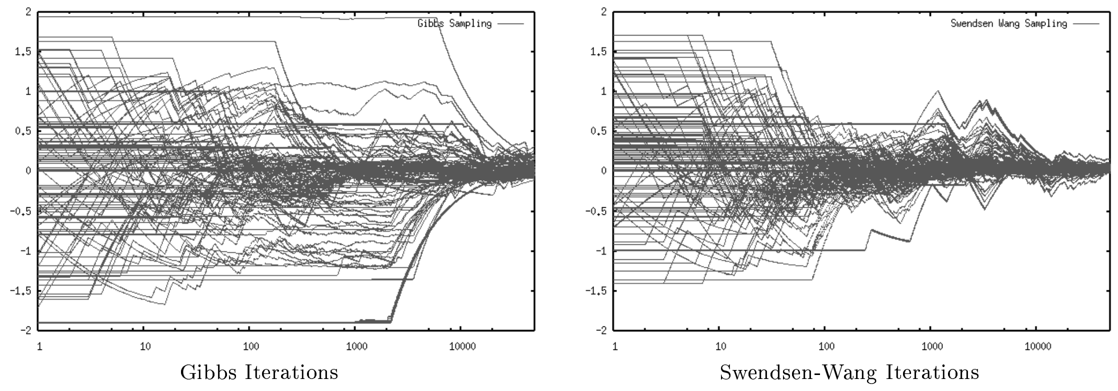


Figure 5.5: Comparing ordinary Gibbs with Swendsen Wang sampling for *Boltzmann5.* The plots show the differences between the running averages and the exact marginals for each of the 361 nodes plotted as a function of the number of whole board samples. Note the log scale. **Left**: Gibbs. **Right**: Swendsen Wang

which gives us

$$\begin{aligned} a_{ij}b_{ij} &\propto f_{ij}(-1,-1), \\ A_{ij}B_{ij} &\propto f_{ij}(1,1), \\ x_{ij}A_{ij}b_{ij} &\propto f_{ij}(1,-1), \\ x_{ij}a_{ij}B_{ij} &\propto f_{ij}(-1,1). \end{aligned}$$

Solving for the unknowns yields

$$\begin{aligned} x_{ij} &= \frac{1}{y_{ij}} = \sqrt{\frac{f_{ij}(1,-1)\cdot f_{ij}(-1,1)}{f_{ij}(1,1)\cdot f_{ij}(-1,-1)}}, \\ \frac{a_{ij}}{A_{ij}} &= \frac{c_{ij}}{C_{ij}} = \sqrt{\frac{f_{ij}(-1,-1)\cdot f_{ij}(-1,1)}{f_{ij}(1,1)\cdot f_{ij}(1,-1)}}, \\ \frac{b_{ij}}{B_{ij}} &= \frac{d_{ij}}{D_{ij}} = \sqrt{\frac{f_{ij}(-1,-1)\cdot f_{ij}(1,-1)}{f_{ij}(1,1)\cdot f_{ij}(-1,1)}}. \end{aligned} \tag{5.3}$$

Sampling Update

Sampling from $p(\mathbf{h}|\mathbf{s},\mathbf{c})$ is straightforward as, conditional on $\mathbf{s}$, each bond is independent. We have

$$p(h_{ij} = \text{`coupled'}|\mathbf{s},\mathbf{h}\setminus h_{ij},\mathbf{c}) = \begin{cases} 1-x_{ij} & \text{if } K_{ij} > 0 \text{ and } s_i = s_j \\ 0 & \text{if } K_{ij} > 0 \text{ and } s_i \neq s_j \\ 1-y_{ij} & \text{if } K_{ij} < 0 \text{ and } s_i \neq s_j \\ 0 & \text{if } K_{ij} < 0 \text{ and } s_i = s_j \end{cases}$$

so x gives the bonding probability for each pair of variables which, as can be seen from equation 5.3, depends on the strength of the coupling between the variables. To sample from $p(\mathbf{s}|\mathbf{h},\mathbf{c})$ we note that there are only two possible global states for each cluster of bonded variables (all nodes the same in the ferromagnetic, positively coupled, case and all adjacent nodes opposite in the anti-ferromagnetic,

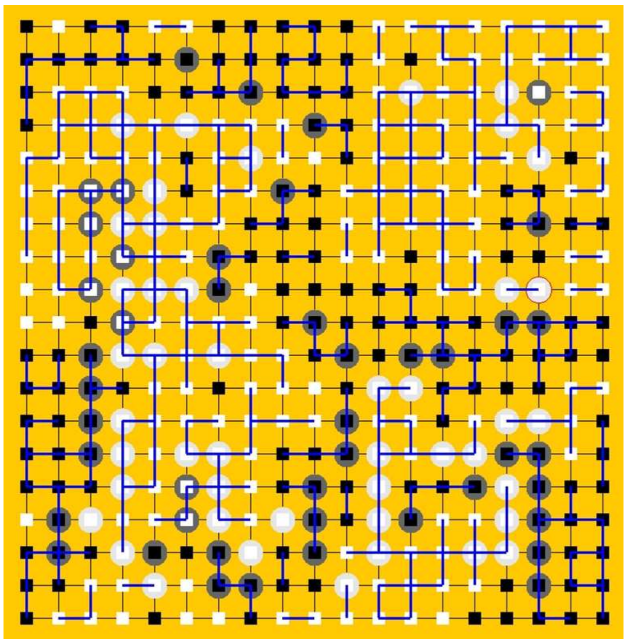


Figure 5.6: Single Swendsen-Wang sample for Botzmann5 with bonds shown as blue lines. This sample was for an expert board position at move 90. For each vertex, $\vec{v}$, a white circle represents a white stone ($c_{\vec{v}}$ = white), a black circle represents a black stone ($c_{\vec{v}}$ = black), a black square represents black territory ($s_{\vec{v}} = +1$), a white square represents white territory ($s_{\vec{v}} = -1$).

negatively coupled, case). We denote the vector of variables involved in a cluster by $\mathbf{s}_b$ and the two legal states of each cluster are given by $\hat{\mathbf{s_b}}$ and $\check{\mathbf{s_b}}$. We flip each cluster in turn based on the probability

$$p(\mathbf{s_b} = \hat{\mathbf{s_b}} | \mathbf{s} \setminus \mathbf{s_b}, \mathbf{h}, \mathbf{c}) = \frac{\Phi(\hat{\mathbf{s_b}})}{\Phi(\hat{\mathbf{s_b}}) + \Phi(\check{\mathbf{s_b}})}$$

where Φ is the cluster potential which for the positively coupled case is given by:

$$\begin{aligned}\Phi(\mathbf{s_b}) \quad = \prod_{g_{ij}(s_i,s_j,h_{ij}) \in R} & [a_{ij}\mathbb{I}(s_i = -1) + A_{ij}\mathbb{I}(s_i = 1)] \cdot [b_{ij}\mathbb{I}(s_j = -1) + B_{ij}\mathbb{I}(s_j = 1)] \cdots \\ & \cdot [x \cdot \mathbb{I}(h_{ij} = \text{‘coupled’}) + (1-x) \cdot \mathbb{I}(h_{ij} = \text{‘not coupled’})]\end{aligned}$$

where R is the set of factors involved in the cluster. The negatively coupled case is the same if x is replaced by y.

Since groups of strongly correlated variables (such as those corresponding to stones in the same chain) will tend to get grouped into a single cluster (according to equation 5.3) which is flipped as a single step during sampling the Swendsen-Wang process mixes much faster for our model than flipping a single variable at a time as in the usual Gibbs sampling update schedule. Figure 5.6 shows a position from an expert game with a single Swendsen-Wang sample overlaid and the Swendsen-Wang bonds shown. The bonded clusters do tend to roughly correspond to regions of common fate. Figure 5.5 compares the empirical convergence of the estimated marginals using Swendsen-Wang sampling with Gibbs sampling.

Using Swendsen-Wang sampling we can find accurate marginals for every vertex of a 19 by 19 Go board in about 5 seconds. A burn-in of 1000 full board samples was used.

5.4 Training

Each training example in the dataset, $\mathcal{D}$, consists of a mid-game position, $\hat{\mathbf{c}}$, from an expert Go game and the final territory outcome of that game, $\hat{\mathbf{s}}$. The training games are a subset of the GoGoD database and the final territory outcomes were determined automatically using GnuGo[3]. Not all the games in GoGoD could be scored by GnuGo because the games were incomplete so we only used games that could be automatically scored for our training and test data.

Bayes rule gives us the posterior distribution over the parameters given the observed territory outcome, assuming each data point is independent:

$$p(\theta|\hat{\mathbf{s}},\hat{\mathbf{c}}) \propto \left\{ \prod_{(\hat{\mathbf{s}},\hat{\mathbf{c}})\in\mathcal{D}} p(\hat{\mathbf{s}},\hat{\mathbf{c}}|\theta) \right\} p(\theta).$$

This requires us to calculate the partition function of $p(\mathbf{s},\mathbf{c}|\theta)$ for all settings of the parameters θ under the distribution $p(\theta)$ leading to a so called 'doubly intractable' problem (Murray and Ghahramani, 2004). Therefore, instead of calculating the full posterior distribution we use gradient descent to determine point estimates for the parameters that maximise one of two alternative objective functions: *maximum a posteriori* (MAP) and *contrastive divergence* (CD).

5.4.1 Maximum Posterior

The goal of MAP learning is to determine

$$\begin{aligned}\operatorname*{argmax}_{\theta} \left(\left\{ \prod_{(\hat{\mathbf{s}},\hat{\mathbf{c}})\in\mathcal{D}} p(\hat{\mathbf{s}},\hat{\mathbf{c}}|\theta) \right\} p(\theta) \right) &= \operatorname*{argmax}_{\theta} \left(\sum_{(\hat{\mathbf{s}},\hat{\mathbf{c}})\in\mathcal{D}} \log\left(p(\hat{\mathbf{s}},\hat{\mathbf{c}}|\theta)\right) + \log\left(p(\theta)\right) \right) && (5.4)\\ &= \operatorname*{argmax}_{\theta} \left(\sum_{(\hat{\mathbf{s}},\hat{\mathbf{c}})\in\mathcal{D}} \mathcal{L}\left(\hat{\mathbf{s}},\hat{\mathbf{c}},\theta\right) + \log\left(p(\theta)\right) \right), && (5.5)\end{aligned}$$

where $\mathcal{L}$ is the log likelihood of a data point. This is achieved by gradient descent using the derivative of the sum of the log likelihoods for each data point and the log prior. The derivative of the log likelihood is equal to

$$\begin{aligned}\frac{d\mathcal{L}\left(\mathbf{s},\mathbf{c},\theta\right)}{d\theta} &= \frac{d}{d\theta}\log\left(\frac{1}{Z(\mathbf{c},\theta)} \prod_{(i,j)\in\mathcal{E}} f_{ij}(s_i,s_j,\mathbf{c},\theta) \right) && (5.6)\\ &= \sum_{(i,j)\in\mathcal{E}} \frac{d}{d\theta}\left\{\log f_{ij}(s_i,s_j,\mathbf{c},\theta)\right\} - \frac{d}{d\theta}\log Z(\mathbf{c},\theta). && (5.7)\end{aligned}$$

[3]GnuGo is available at `http://www.gnu.org/software/gnugo/gnugo.html`

Where the differential of the log partition function is given by

$$\frac{d}{d\theta}\log Z(\mathbf{c},\theta) = \frac{d}{d\theta}\log\sum_{\mathbf{s}}\prod_{(i,j)\in\mathcal{E}} f_{ij}(s_i,s_j,\mathbf{c},\theta) \tag{5.8}$$

$$= \sum_{\mathbf{s}} \frac{\frac{d}{d\theta}\prod_{(i,j)\in\mathcal{E}} f_{ij}(s_i,s_j,\mathbf{c},\theta)}{Z(\mathbf{c},\theta)} \tag{5.9}$$

$$= \sum_{\mathbf{s}} \frac{\frac{d}{d\theta}\exp\left(\sum_{(i,j)\in\mathcal{E}}\log f_{ij}(s_i,s_j,\mathbf{c},\theta)\right)}{Z(\mathbf{c},\theta)} \tag{5.10}$$

$$= \left\langle \sum_{(i,j)\in\mathcal{E}} \frac{d}{d\theta}\log f_{ij}(s_i,s_j,\mathbf{c},\theta)\right\rangle_{p(\mathbf{s}|\mathbf{c})} \tag{5.11}$$

so the derivative of the log-likelihood of all the data with respect to the parameters can be written

$$\frac{d}{d\theta}\left(\sum_{(\hat{\mathbf{s}},\hat{\mathbf{c}})\in\mathcal{D}}\mathcal{L}(\hat{\mathbf{s}},\hat{\mathbf{c}},\theta)\right) = \left\langle\sum_{(i,j)\in\mathcal{E}}\frac{d\log f_{ij}(\hat{s}_i,\hat{s}_j,\hat{\mathbf{c}},\theta)}{d\theta}\right\rangle_{\mathcal{D}} - \left\langle\sum_{(i,j)\in\mathcal{E}}\frac{d\log f_{ij}(s_i,s_j,\hat{\mathbf{c}},\theta)}{d\theta}\right\rangle_{p(\mathbf{s}|\hat{\mathbf{c}})}. \tag{5.12}$$

where the expectation of the left hand term is simply the mean of its argument over the training data and the expectation of the right hand term is calculated under the model distribution, $p(\mathbf{s}|\mathbf{c})$. This is equivalent to minimizing the KL divergence between the data distribution and the model distribution. To evaluate the expectation under the model distribution we must calculate the pointwise and adjacent pairwise marginal probabilities of territory outcomes for each board position, which can be calculated using one of the inference methods described in the previous section.

The prior $p(\theta)$ is set to

$$p(\theta) = \frac{1}{Z_p(\alpha)}\exp\left(-\frac{1}{2}\alpha\sum_{\theta_i\in\theta}\theta_i^2\right) \tag{5.13}$$

so

$$\frac{d(\log p(\theta))}{d\theta_i} = -\alpha\theta_i \tag{5.14}$$

which corresponds to simple weight decay (MacKay, 2003).

5.4.2 Contrastive Divergence

A standard method for Boltzmann machine learning is to estimate the right hand term in equation 5.12 by Gibbs sampling so the resulting gradient will be a stochastic approximation to the true gradient. A simpler alternative to attempting to find an accurate estimate of the gradient of equation 5.12 is to calculate instead

$$\frac{d}{d\theta}\left(\sum_{(\hat{\mathbf{s}},\hat{\mathbf{c}})\in\mathcal{D}}\mathrm{CD}(\hat{\mathbf{s}},\hat{\mathbf{c}},\theta)\right) = \left\langle\sum_{(i,j)\in\mathcal{E}}\frac{d\log f_{ij}(\hat{s}_i,\hat{s}_j,\hat{\mathbf{c}},\theta)}{d\theta}\right\rangle_{\mathcal{D}} - \left\langle\sum_{(i,j)\in\mathcal{E}}\frac{d\log f_{ij}(s_i,s_j,\hat{\mathbf{c}},\theta)}{d\theta}\right\rangle_{q_t(\mathbf{s}|\hat{\mathbf{c}},\hat{\mathbf{s}})}. \tag{5.15}$$

where the expectation of the second term is calculated under an approximate distribution q_t . Each sample from q_t is calculated by simply initialising all the territory variables, $\mathbf{s}$, to the observed data, $\hat{\mathbf{s}}$, and then running the Gibbs sampler for t steps (Hinton, 2002), using the final sample in the Markov chain. The parameter, t, is typically set to 1. By performing gradient descent using (5.15) we are attempting to minimise an objective function called the 'contrastive divergence' which intuitively we can think of as a stochastic measure of the degree to which Gibbs sampling under the model distribution tends to cause the state of the model variables to diverge from the data. Training the model by minimising contrastive divergence with $t = 1$ while using Swendsen-Wang sampling to perform the Gibbs step and the adaptive meta descent scheme described below to perform the complete optimisation is roughly 1,000 times faster than fitting the model by MAP although, as we will see in the results section, the performance of the resulting system is slightly poorer. For the contrastive divergence experiments a weight decay setting of $\alpha = 0.001$ was used.

5.4.3 Gradient Descent Methods

If a deterministic method is used to estimate the gradient (5.12) (either loopy belief propagation or exact variable elimination) then we used a version of the conjugate gradient descent procedure which only requires gradient information in order to determine the optimal parameters[4]. This method performs poorly when the gradient is stochastic so we used an alternative adaptive gradient descent scheme where Gibbs sampling was used to estimate the gradient for MAP or CD learning. In that case the parameters were updated at each iteration n according to

$$\theta_i^{n+1} = \theta_i^n + \delta_i^n \cdot \nabla_i^n$$

where ∇_i is the gradient with respect to parameter θ_i and δ_i is the step size, and there is a separate step size for each parameter. The step size is adapted according to ALAP (Almeida *et al.*, 1998):

$$\delta^{n+1} = \delta^n + \frac{\gamma \cdot \nabla_i^n \cdot \nabla_i^{n-1}}{\tau^n}$$

where τ is also adapted according to

$$\tau^{n+1} = \mu\tau^n + (1-\mu)\nabla_i^n \cdot \nabla_i^{n-1}.$$

The meta-descent parameters are set to $\mu = 0.9$ and $\gamma = 0.01$.

5.5 Results

5.5.1 Learned Parameters for Boltzmann5

Table 5.1 gives the values of the parameters of the Boltzmann5 model after it was trained using MAP for 290 expert game positions. Separate experiments were performed for three game stages: move 20, move 80 and move 150. As expected, w_{chains} takes on a high value which corresponds to a strong correlation between the plausible territory outcome of two adjacent vertices in a chain of connected stones. This is due to the common fate property of chains. The value of w_{empty} is also interesting (corresponding to the coupling between territory predictions of neighboring vertices in empty space) which is close to the critical coupling for an Ising model, 0.441. Perhaps this is because an Ising model close to the critical temperature has clusters (contiguous spins all in the same state) of a similar size to typical regions of contiguous Go territory. If the temperature was too high then only small contiguous

[4]Macopt by David MacKay: http://www.inference.phy.cam.ac.uk/mackay/c/macopt.html.

parameter	move 20	move 80	move 150
h_{stones}	0.271	0.265	0.235
w_{empty}	0.406	0.427	0.496
$w_{\text{chain}-\text{empty}}$	0.410	0.442	0.426
w_{chains}	1.866	2.740	2.976
$w_{\text{inter}-\text{chain}}$	0.490	0.521	0.429

Table 5.1: Parameters learned for Boltzmann5 using maximum likelihood training (using Swendsen-Wang sampling for inference).

regions of territory would be possible (too small to contain two eyes and live) and if the temperature was too low then this would mean that with high probability the board would either be all white or all black territory (quite unlikely in Go). The positive value for the stone bias, h_{stones}, simply means that a stone is more likely to live than to die in the expert positions. This positive stone bias will be counteracted in cases where a chain is surrounded by opponent stones because of the positive value of $w_{\text{inter}-\text{chains}}$ which is the tendency for a chain to have the same territory outcome of a neighboring chain – i.e one chain survives at the expense of another.

5.5.2 Territory Prediction Performance

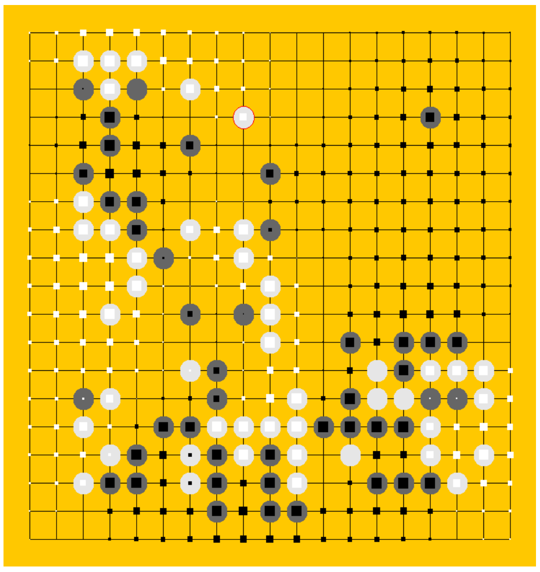
Territory Predictions

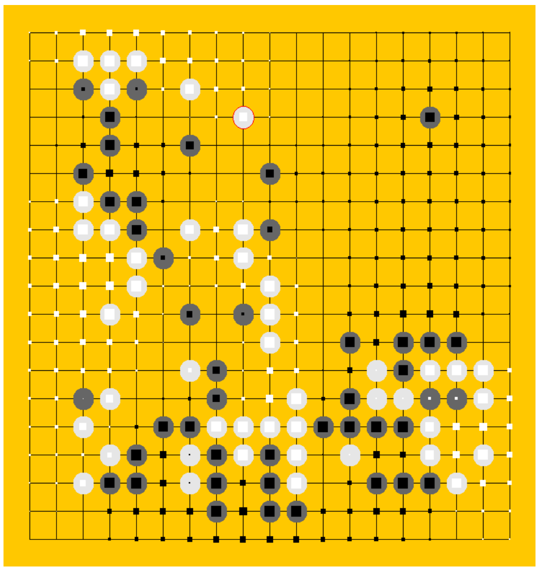
Figure 5.6 shows a single sample from the distribution over territory outcomes under the Boltzmann5 model generated by the Swendsen-Wang technique. The Swendsen-Wang bonds are also shown. This sample represents a hypothesis about what the final territory outcome of the game might be. The contiguous regions of the same colour within the territory prediction have a similar typical size to typical regions of territory in Go and the Swendsen-Wang bonds appear to cluster together regions which we might consider to have approximately common fate (due to stones becoming connected in the future). All connected chains of stones form single clusters. In general, more stones are alive than dead.

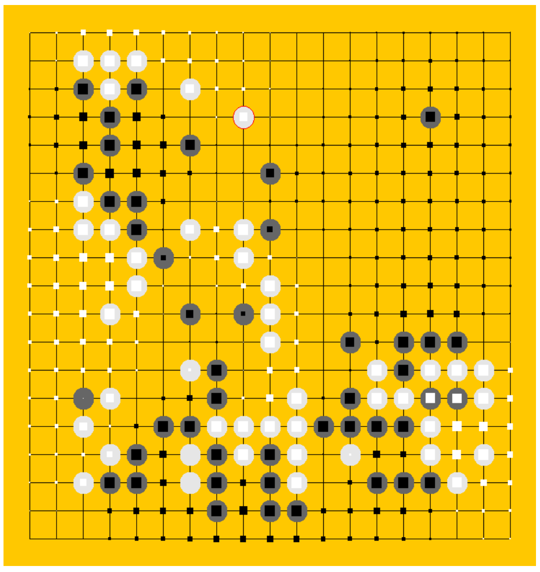
Figure 5.7 shows board positions labeled with the expected pointwise territory outcome under the 3 models introduced in this chapter (Boltzmann5, Boltzmann12 and BoltzmannLiberties).

Go players confirm that the territory predictions seem plausible. The common fate of chains is predicted correctly and chains that are more surrounded by opponent chains are more likely to be dead whereas chains with many liberties or with plenty of space are predicted to be alive. Comparing the figure for BoltzmannLiberties with the other Boltzmann models shows that the liberty information allows the model to correctly identify which of the two competing small chains in the bottom right of the board is dead.

The (non-probabilistic) territory prediction from the commercial Go program 'The Many Faces of Go' is also included for comparison (Fotland, 1993). However, it is difficult to objectively evaluate the territory prediction performance of the Boltzmann models against the one provided by this program as the 'Many Faces of Go' prediction is only available graphically. At any rate, I don't believe we would benefit much by comparing with other Go program's territory predictions as none of these predictions represents ground truth and it is unclear how accurate they are in the first place. The types of predictions produced by these programs will depend on how they are used to generate moves,


Boltzmann5


Boltzmann12


BoltzmannLiberties

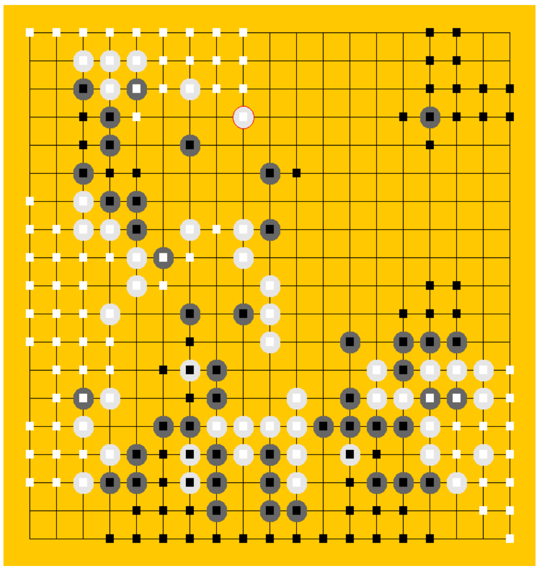
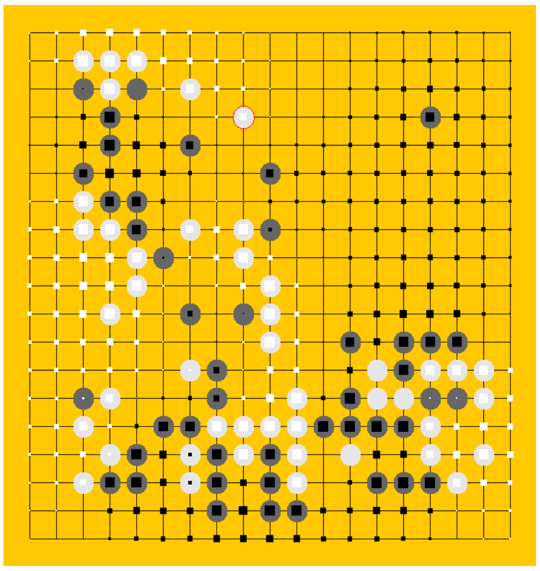

Many Faces of Go

Figure 5.7: Comparing territory predictions for a Go position from a professional game at move 90. The circles represent stones. The small black and white squares at each vertex represent the average territory prediction at that vertex, from -1 (maximum white square) to $+1$ (maximum black square). Exact inference was used for the Boltzmann models (this takes 10 minutes per position but indistinguishable results can be achieved in 5 seconds per position using Swensen-Wang sampling.)


Boltzmann5 (Exact)

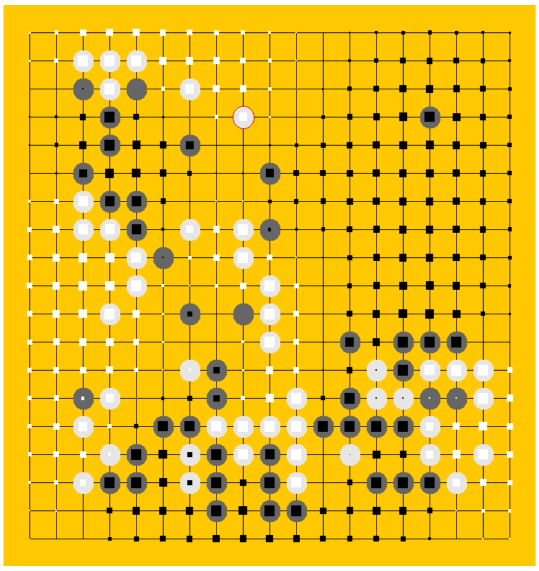

Boltzmann5 (Loopy BP)

Figure 5.8: Comparing territory predictions for a Go position using different inference methods from a professional game at move 90. The circles represent stones. The small black and white squares at each vertex represent the average territory prediction at that vertex, from -1 (maximum white square) to +1 (maximum black square).

and they will be tuned for that purpose rather than to accurately predict the final game outcome. Also, none of these programs produces a probability distribution over territory outcomes, only a point estimate. Therefore, for the rest of this chapter we use the predictive probability of ground truth (the actual territory outcomes of test games) to evaluate the different models.

Comparing Models and Inference Methods

Figure 5.8 compares the territory prediction when calculated exactly for the Boltzmann5 model (left) with the territory prediction when inference is performed using loopy belief propagation (right). In general loopy belief propagation tends to give marginals which are polarized towards $+1$ or -1. This 'over-confidence' (lack of uncertainty) is a known side-effect of loopy belief propagation (Weiss, 1998). The over-confidence is more prominent in regions of the board where fewer stones are present (for example see the top right of the board) and less prominent in regions where more stones are present (bottom left of the board). The parameters for the loopy BP results were trained by MAP, using loopy belief propagation for inference.

Figures 5.9 and 5.10 show the log predictive probabilities of the territory outcomes for a set of expert games. Since the performance of Boltzmann5 and Boltzmann12 appeared nearly identical, Boltzmann12 was not considered further. Figure 5.11 shows the results of the same experiment as Figure 5.10 but where each data-point used to generate the plot is the per-position mean of the log probabilities of the territory outcomes. For all the predictive probability plots we see that as we progress through a game, predictions unsurprisingly become more accurate but (at least when we take the per-position means) the variance of the accuracy increases, possibly because of the incorrect assessment of the life and death status of groups of stones. Swendsen-Wang performs better than loopy belief propagation which suffers from over-confidence although this difference becomes smaller

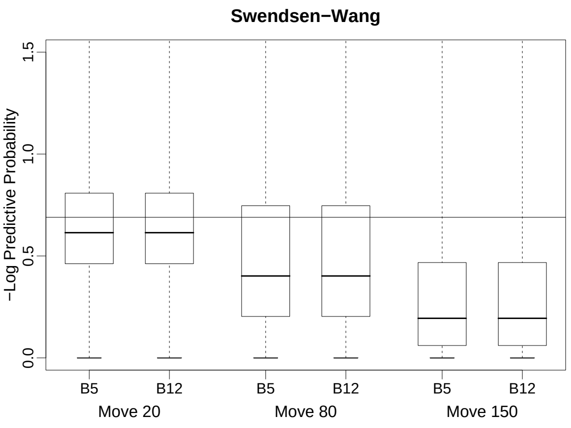


Figure 5.9: Log predictive probability of the territory outcome at each vertex of 327 unseen Go positions, $p(s_i^d|\mathbf{c}^d)$ where d is an index over data points. The horizontal lines in the boxes show the lower quartile, median and upper quartile. The whiskers extent to the most extreme data point. 3 board positions were analysed for each game (move 20, 80 and 150). The Boltzmann5 (B5) and the Boltzmann12 (B12) models are compared and found to have near identical performance. The horizontal line across the plot is at $-\ln(0.5) = 0.69$, the probability of the outcome if guessing randomly.

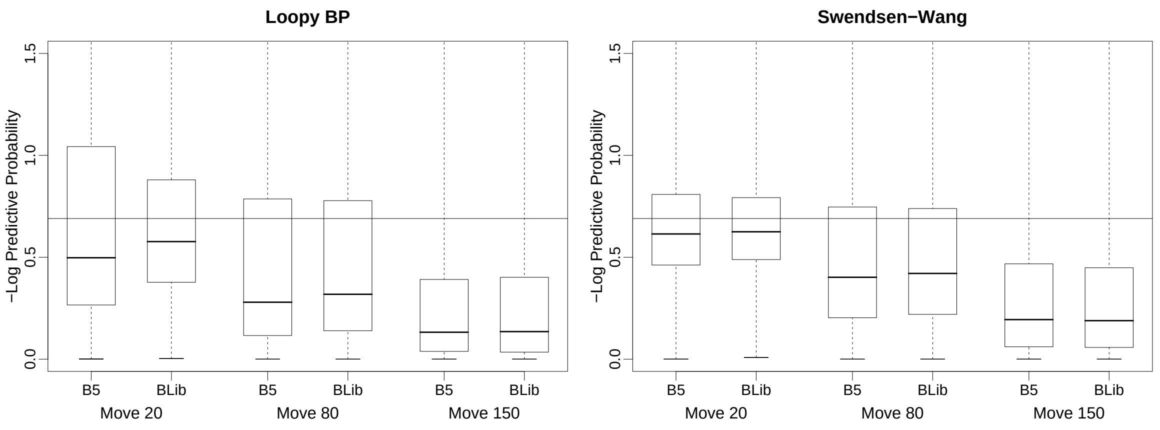


Figure 5.10: Log predictive probability of the territory outcome at each vertex of 327 unseen Go positions, $p(s_i^d|\mathbf{c}^d)$. The horizontal lines in the boxes show the lower quartile, median and upper quartile. The whiskers extent to the most extreme data point. Sampling is compared with loopy belief propagation. 3 board positions were analysed for each game (move 20, 80 and 150). The Boltzmann5 (B5) and the BoltzmannLiberties (BLib) models are compared. The horizontal line across the plot is at $-\ln(0.5) = 0.69$, the probability of the outcome if guessing randomly.

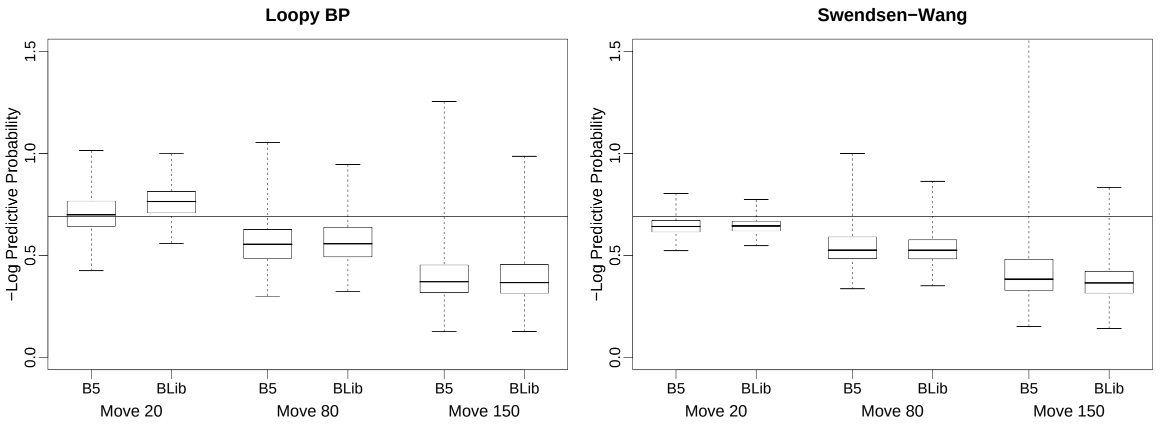


Figure 5.11: Each data-point is the per-position mean of the log predictive probability of the territory outcomes at the vertices of 327 unseen Go positions, $\frac{1}{N \times N} \sum_{\vec{v} \in \mathcal{N}} p(s^d_{\vec{v}} | \mathbf{c}^d)$. The horizontal lines in the boxes show the lower quartile, median and upper quartile. The whiskers extent to the most extreme data point. Sampling is compared with loopy belief propagation. 3 board positions were analysed for each game (move 20, 80 and 150). The Boltzmann5 (B5) and the BoltzmannLiberties (BLib) models are compared. The horizontal line across the plot is at $-\ln(0.5) = 0.69$, the probability of the outcome if guessing randomly.

in the later stages of the game when there are more stones present on the board. BoltzmannLiberties performs slightly better than Boltzmann5 (when using Swendsen-Wang for inference) according to Figure 5.11 and the variance of the predictions made by BoltzmannLiberties is smaller. The difference in performance between the two models increases at the end of the game when liberty counts become more important for assessing group safety.

Figure 5.12 compares the performance of the Boltzmann5 and BoltzmannLiberties model on move 80 for parameters trained using MAP with those trained using contrastive divergence. The Gibbs reconstruction step in contrastive divergence was performed using one full board Swendsen-Wang iteration. Minimising contrastive divergence is about 1,000 times faster but the results appear slightly worse with higher variance. Recent work has shown that it would probably be possible to improve performance by increasing t as the gradient descent proceeds so that we will gradually home in on the MAP solution (Salakhutdinov *et al.*, 2007).

5.6 Move Prediction

We can obtain a probability distribution over moves by choosing a Gibbs distribution with the negative energy replaced by one of the evaluations given in Section 5.1.1:

$$p(\vec{v}|, \theta, \mathcal{H}) = \frac{e^{\beta u(m(\vec{v}, \mathbf{c}), \theta)}}{\sum_{\vec{v}' \in \mathcal{L}(\mathbf{c})} e^{\beta u(m(\vec{v}', \mathbf{c}), \theta)}} \tag{5.16}$$

where the function $m : \mathcal{N} \times \mathcal{C}^{N \times N} \to \mathcal{C}^{N \times N}$ returns the new board position generated by applying a move to a board position. Directly applying the Boltzmann machine trained to predict territory in expert games to provide the evaluation function u for equation 5.16 resulted in very poor move

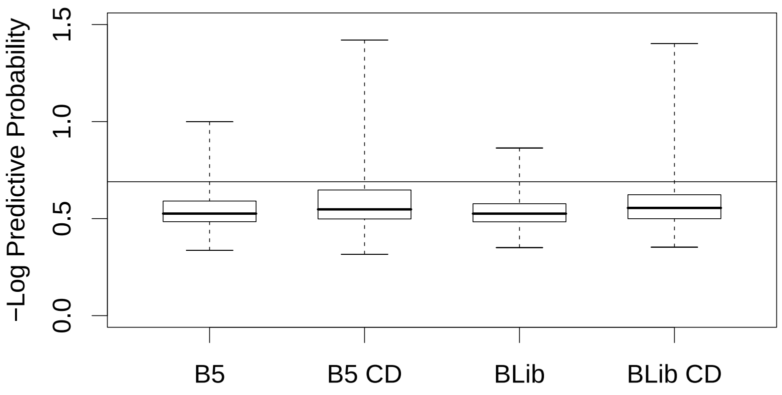


Figure 5.12: Per position mean log predictive probability of the territory outcomes at the vertices of 327 unseen Go positions, $\frac{1}{N \times N} \sum_{\vec{v} \in \mathcal{N}} p(s^d_{\vec{v}} | \mathbf{c}^d)$. Boltzmann5 (B5) is compared with BoltzmannLiberties (B78) for two different training methods (maximum likelihood and contrastive divergence (CD).

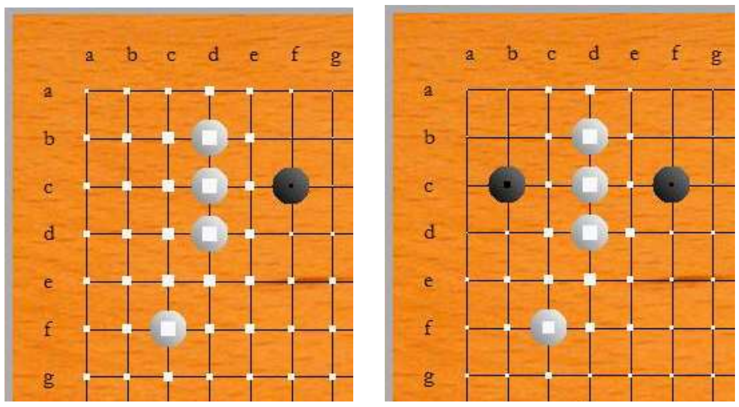


Figure 5.13: Stone life overconfidence example. This shows how the non-expert black move leading to the position on the right leads to the model predicting too much territory resulting from this black stone. The area of each square is the magnitude of the expected territory outcome at that vertex ($\in [-1, 1]$) and the colour gives the sign.

prediction performance. Since the model is trained to predict the final territory outcome given game positions from *expert* games there is a selection bias which leads to poor prediction of the territory outcome given a *non-expert* mid-game position. To be precise, the model is over-confident about the survival of stones on the board because experts tend to place stones such that they will live (so most stones in expert games do live) but if a stone is placed in a random location on the board then the chance of survival should be much smaller. Figure 5.13 gives an example where a black stone is placed in a location where it is likely to be captured and will not produce any territory. Here, the model is overconfident about its safety because it has learned that most stones live (as is the case if experts place them). Most of the positions generated to normalise equation 5.16 are not expert positions so the move prediction performance is poor.

5.7 Discussion

A simple Markov random field model can capture a number of the characteristics of plausible territory outcomes. In samples generated by the model the typical size of contiguous clusters of territory of the same colour are of a similar size to clusters of territory found in real games. Each chain of stones tends to fall within the same territory cluster as suggested by the common fate of chains. Also, the models are capable of predicting the life and death of stones in simple cases.

A number of model improvements are possible. The local factors could be conditioned on larger areas of the board position, rather than just the vertices corresponding to the points in the territory outcome these factors are responsible for. Also the model could be conditioned on more features in addition to liberty counts. These improvements could be made without increasing the complexity of inference and the efficient Swendsen-Wang sampling method could be applied. Sanner *et al.* (2007) combined the pattern system explained in Chapter 3 with the simple Boltzmann machine models of this chapter, using the patterns to determine the bias factors. The slow speed of learning in Boltzmann machines was not a serious problem for the models presented in this chapter as they have few parameters (shared across the factors) so few training games are needed. When using patterns to determine the biases, far more parameters must be tuned so more training games are needed and learning takes too long. To address this issue Sanner *et al.* (2007) made use of the pseudo-likelihood training method and another local training method ('piecewise' learning), both of which which resulted in quite poor performance (worse than the models presented in this chapter, despite having orders of magnitude more parameters). A possible future work direction would be to use Contrastive Divergence learning with Swendsen-Wang used for the reconstruction step to train the models presented by Sanner *et al.* (2007). I believe this would allow far more training data to be used in a given time so models with more parameters would work better.

Another way the model could be improved would be to increase the size of the factors (that is, making them a function of the territory outcomes at more than two adjacent vertices) which would allow the model to capture non-linear local effects. This would make inference more difficult. Adding hidden units would be another way of improving the representational power of the model, perhaps adding a number of layers and moving in the direction of Deep Belief Networks (Hinton *et al.*, 2006).

These model improvements might allow the model to predict the life and death of Go stones more accurately. However, one property of samples generated by these models would remain: there is no guarantee that they will be legal hypotheses. A 'group' of connected Go stones corresponds to a set of stones within a region of contiguous territory of the same colour in the final game outcome. For a group of stones to survive they must be connected to at least two eyes. However, as can be seen in Figure 5.6, there are clusters of contiguous regions of territory which are too small to contain two eyes so the corresponding groups of stones would have been captured and become part of the opponent's territory. As pointed out by van der Werf (2005), the final territory outcome and the life and death of chains of stones depend on each other. The life and death status of stones determines which stones will belong to each player and hence is the starting point for predicting the territory outcome. However, the territory prediction itself implicitly predicts which stones are grouped together into entities (groups) for which we can count eyes and thus predict life and death.

It is difficult to write down a probabilistic model that correctly models life and death (in the sense that every hypothesis is legal) and that is purely static in the sense it does not explore the game state space. In the next (and final) Chapter we consider a class of models that can predict the final outcome of games by quite a different approach. Rather than building complex models that are able to statically value a position, we generate samples from the probability distribution over final outcomes by stochastically generating the sequence of moves from the current position to the end of the game. Since the sequence of moves must be legal, every hypothetical final territory outcome will be legal. Even the simplest version of these so called Monte Carlo planning techniques (Brügmann, 1993) can predict the life and death of Go stones surprisingly accurately and according to Sanner *et al.* (2007) gives a better trade off between computational cost and accuracy than simple Boltzmann machine models such as the ones discussed in this chapter. In the next chapter we apply Monte Carlo techniques to move selection.

Chapter 6

Monte Carlo Planning

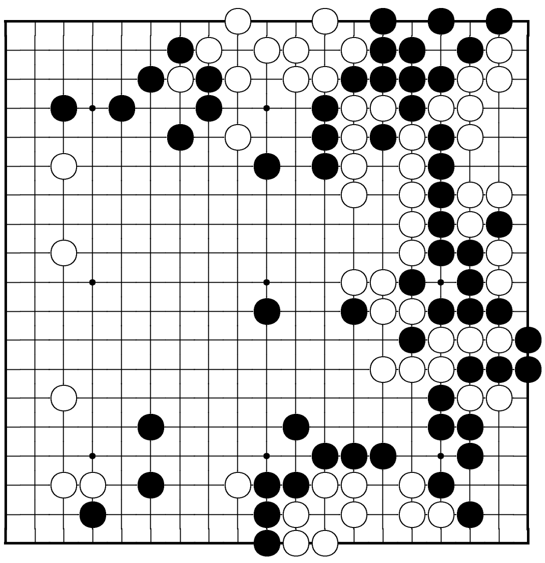


Figure 6.1: The positions displayed at the top of each chapter are from a famous game played in 1846 between Shusaku (black) and Gennan Inseki (white). Gennan was 50 years old and one of the most famous Go players in Japan, while Shusaku was just 17 years old. The isolated black stone in the center turned the game around and has become known as the 'ear-reddening move'. Many people were watching this match, mostly disciples of Gennan. A doctor who was present recalled the moment when that crucial stone was placed on the board: 'I don't know much about the game, but when Shusaku played, Gennan's ears flushed red. This is a sign he had been upset.' (Hollosi and Pahle, 2007)

This chapter returns to the task of move prediction. However we now consider the task of choosing the best move in a game rather than predicting the play of human Go players. In the previous chapter we considered models which can predict some aspects of the final territory outcomes of games well. However, these models would have to be made much more complex in order to accurately predict the life and death of Go stones and for this reason they would not correctly score a final board position. An alternative method to obtain samples from the probability distribution over final game outcomes which guarantees that each sample is legal is to play the game forward from the current position $\mathbf{c}$ to the very end using some stochastic policy for each player. This is called a 'playout' or 'rollout'. The final territory outcome of this game, $\mathbf{s}$, is a sample from $p(\mathbf{s}|\mathbf{c})$ and this distribution is determined by the policy used to play each rollout. If this policy is determined from evidence gathered from previous rollouts, that is, moves are chosen which have been seen empirically to lead to more victories, then the system may be able to 'bootstrap' itself to play stronger and stronger and this is the principle behind the 'Upper Confidence applied to Trees' (UCT) algorithm by Kocsis and Szepesvari (2006) which has recently been applied to Go with a great deal of success (Gelly *et al.*, 2006). This chapter presents some initial experiments which evaluate a selection of Bayesian alternatives to UCT on artificial game trees. I have not yet applied these ideas to Go. A possible advantage of using a Bayesian approach to this problem is that it would enable the principled addition of domain knowledge by the assignment of prior distributions on the values of moves and positions. I present some initial experiments on the effect of including prior knowledge in these algorithms.

6.1 Monte Carlo Planning

In general we are interested in selecting actions in Markov Decision Problems (Sutton and Barto, 1998) with large state spaces and delayed rewards. In each state, $s \in S$, a number of actions (moves), $a \in \mathcal{A}(s)$ are available. Here we are concerned with deterministic processes with a transition function $m : S \times \mathcal{A}(S) \to S$. We define terminal states, $t \in \mathcal{T}$, $\mathcal{T} \subset S$ to be states where no actions are available, that is, $\mathcal{A}(t) = \emptyset$. We define the reward function $r : S \to \mathbb{R}$ as

$$r(s) = \begin{cases} l(s) & \text{if } s \in \mathcal{T} \\ 0 & \text{otherwise} \end{cases}$$

so only terminal states are associated with rewards. In the simplest case of a game with a win/loss outcome the terminal function $l : T \to \mathbb{R}$ may be defined as

$$l(s) = \begin{cases} 1 & \text{for win} \\ 0 & \text{for loss} \end{cases} .$$

Alternatively the terminal function may return the game score. The *policy,* $\pi(s, a)$, defines a probability distribution over actions in a given state, i.e, $\pi(s, a) = p(a|s)$, $s \in S$, $a \in \mathcal{A}(s)$. Monte-Carlo planning is concerned with selecting the best action in a given state by sampling from the state space where each sample is one path from the starting state to the terminal state (a rollout) according to a policy π.

6.1.1 Uniform Monte-Carlo Planning

The simplest method is to use a fixed uniform stochastic policy: $\pi(s, a) = \frac{1}{|\mathcal{A}(s)|}$ for all states and to value a state by running a number of rollouts and taking the mean of the observed rewards. If the set of rewards observed in rollouts from a given state, s, is $X(s)$ then the value of a state, $V(s) = \frac{1}{|X(s)|} \sum_{x \in X(s)} x$. After the algorithm is terminated (after some time) the values of the states can be used for action selection. This technique was applied by Brügmann (1993) to Go with surprisingly good results considering the size of the state space. We refer to this algorithm as Uniform Monte Carlo Planning (UMC).

6.1.2 Upper Confidence applied to Trees

'Upper Confidence applied to Trees' (UCT) (Kocsis and Szepesvari, 2006) also estimates the value of each state as the mean value of the rewards observed in rollouts from that state. However, the policy is adapted according to observations made in previous rollouts. For each state s we store n_s, the number of times the state has been encountered so far and $R_s = \sum_{x \in X(s)} x$, the sum of the rewards received by rollouts passing through state s. The (deterministic) policy is given by $\pi(s,a) = \mathbb{I}(a = \text{argmax}_{\ a_i \in \mathcal{A}(s)} \{V(m(s, a_i(s)), s)\})$. The utility of moving to a state q is given by $V(q,s)$ where s is the previous state. This utility is defined as $V(q,s) = \frac{1}{n_q} R_q + c(n_q, n_s)$ where n_s is the number of times the previous state in the Markov chain has been seen which is equal to the number of times state q has been available as a choice. The bias, c, given by

$$c(n_q, n) = \sqrt{\frac{2 \ln n}{n_q}}$$

which ensures that confidence bounds are satisfied about the accuracy of the state value and hence ensures adequate exploration. Essentially, each decision is treated as a separate multi-armed bandit problem, where each move is a separate arm and the actions are chosen such as to balance exploration (determining which arm is actually best) and exploitation (reaping rewards by choosing the arm currently believed to be the best). After each rollout the values R_s and n_s are updated for each state, s, encountered along the way. By adapting the policy in this way the rollouts concentrate on promising lines of play and eventually the value of each state will converge to its minmax value. For certain state spaces (the target applications of UCT) action selection with very low error rates may be possible after far fewer iterations than would be needed to perform an alpha-beta search as we demonstrate empirically in Section 6.2.3.

6.2 Synthetic Game Trees

In this chapter we test various Monte-Carlo planning algorithms by applying them to the task of move selection in synthetic game trees. The tree is the set of possible paths through state space from a given root state and each Monte-Carlo rollout consists of one path from the root to a leaf of the tree. The depth (D) is the number of states visited by a rollout and the branching factor (B) of the tree is the number of actions available in each state. Here we will focus on adversarial game trees. We construct synthetic trees by 1) generating the tree with the desired dimensions, 2) assigning win/loss values to the leaves (using one of the two methods described below) and 3) determining which player can force a win from each state by minimax. The first player to play is called MAX and her opponent is called MIN.

6.2.1 Random Trees

A 'Random Tree' is a tree whose leaves are assigned binary win/loss status independently at random according to $p(\text{win}) = \pi_w$. If this value is set to a critical value, π_c, (the positive root of the equation $x^B + x + 1 = 0$) then the probability that the first player can force a win from the starting position, $P(w) = \pi_w = \pi_c$ (Pearl, 1984). In general, $P(w)$ is a function of π_w and the depth of the tree, D. As the depth of the tree increases this function rapidly approaches a step function

$$\lim_{D\to\infty} P\left(w|D,\pi_w\right) \simeq \begin{cases} \pi_c & \text{if } \pi_w = \pi_c \\ 0 & \text{if } \pi_w < \pi_c \\ 1 & \text{if } \pi_w > \pi_c \end{cases} .$$

For a binary ($B = 2$) tree π_c is equal to the golden ratio $(\sqrt{5}-1)/2 = 0.6180339$. For the experiments here π_w is set to π_c to ensure a reasonable value for $P(w)$.

6.2.2 P-Game Trees

A 'P-Game tree' (Kocsis and Szepesvari, 2006) simulates a two person, zero-sum, game where the outcome is determined by scoring the final position. Go is an example of such a game.

Only the value of the final board position is observed directly by the players. Each move in the game tree has a value drawn from the interval $[0, 1]$ for the MAX player and $[-1, 0]$ for the MIN player. The value of a terminal position is simply the sum of the values of the moves along the path from the root to the corresponding leaf. If this value is positive then the result is a win for MAX, otherwise it is a win for MIN. In other words we model a game as a random walk of a fixed number of steps. The values of the moves and the intermediate nodes in the game tree are unknown to the game players.

It is hoped that this simple model captures the qualitative nature of the Go game tree and in particular the smoothness in the value of states in a game tree. The notions of being 'ahead' or 'behind' in a game rely on smoothness in the value of nodes in the game tree. Another advantage of this model is that each move is associated with a value. We assume that this value is usually unobserved to the players of the game but in simulations a noisy observation of its value could be used to test the addition of domain knowledge.

6.2.3 Experiments with Existing MC Planning Techniques

Figure 6.2 compares the performance of UCT, Uniform Monte Carlo, and alpha-beta search on synthetic game trees. For each plot, $g = 2000$ trees are generated and the leaves are valued according the Random or PGame method (see Sections 6.2.1 and 6.2.2 respectively) using different random numbers for each tree. Next, the win/loss values of the internal nodes are determined by minimax and these 'delphic' values are used to assess the performance of the planning algorithms tested on the trees. Each algorithm is run on each tree and at each iteration (rollout) we record whether the move from the root currently valued as best by the model is actually a win. If it is a win then this is recorded as a success for that iteration. The 'mean error' rate is $\frac{\#\text{successes}}{g}$ and is plotted as a function of iteration number. Alpha-beta search is a depth first search so one iteration from root to leaf is considered equivalent to one rollout.

Both Monte-Carlo planning techniques appear to fail on Random trees (Figure 6.2 left). Firstly, compare the curves for Uniform Monte Carlo (MC) for the two tree types. On Random trees the error rate remains constant at the rate for random move selection whereas for P-Game trees the error rate decreases rapidly at first but then converges to a fixed positive value. We know empirically that this algorithm can be used to select reasonable Go moves (Brügmann, 1993) so it seems likely that the P-Game model is a better model of Go game trees than the random tree model. This suggests that smoothness in the underlying values of nodes in the tree is important (in the sense that if one is in a strong position then there are likely to be more moves available leading to other strong positions than

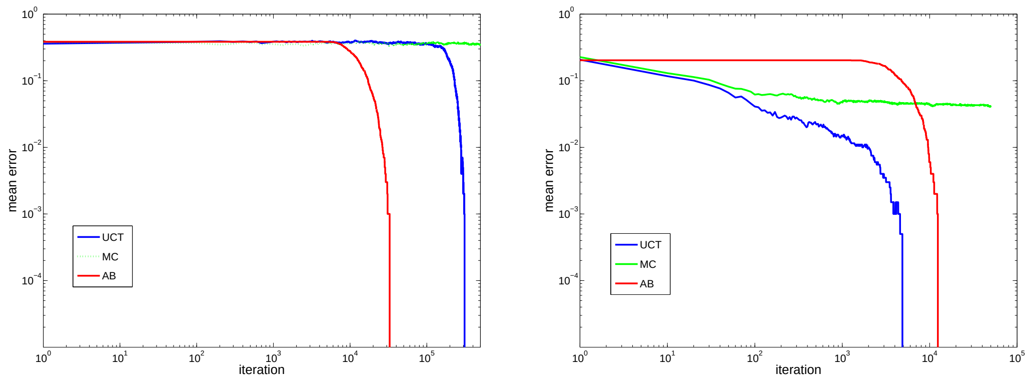


Figure 6.2: Comparing the performance of Monte Carlo planning on different synthetic game trees ($B = 2, D = 20$). Alpha Beta search (AB) is compared to Uniform Monte Carlo (MC) and UCT. For each plot 2000 trees are generated and the leaves are valued according the Random or PGame method (see Sections 6.2.1 and 6.2.2 respectively). Then the win/loss values of the internal nodes are determined by minimax. Each algorithm is run on each tree and at each iteration (rollout) we record whether the move from the root currently valued as best is actually a win. If it is a win then this counts as a success. The 'mean error' rate is $\frac{\#\text{successes}}{2000}$ and is plotted as a function of iteration number. **Left:** Random Tree with $\pi_w = 0.6180339$, **Right:** PGame Tree (see Section 6.2).

weak positions). This contradicts the independence assumption made by the Probability Propagation model of Chapter 4 and might provide a partial explanation for the fairly poor performance of that model.

Now compare the curves for UCT for the two tree types. UCT converges much slower than alpha-beta for the Random tree but much faster than alpha-beta for the P-Game tree. This result, in conjunction with the success of UCT for Go (Gelly *et al.*, 2006), also suggests that P-Game trees are a better model of Go game trees than Random trees. We use P-Game trees to test various approaches to Monte Carlo planning in this chapter.

6.3 Bayesian Models

UCT falls into a category of Monte Carlo planning techniques which we will call 'adaptive' because the policy is learned from previous rollouts. We now consider a number of alternative adaptive Monte-Carlo planning techniques which explicitly model the distributions over the values of each state, and balance exploration and exploitation by taking account of uncertainty using the standard rules of probability.

6.3.1 Independent Dynamic Counting Model

Model

For the first approach, we model the value of each game state separately (much like UCT) and do not explicitly model the correlations between states. Each state is valued by the number of wins and losses of games that encountered it, ignoring the path taken through state space between it and the end of the game. We model the distribution over the binary (win/loss) value, c, of games which encounter the state. This value is observed by performing rollout simulations. In other words, at the end of each

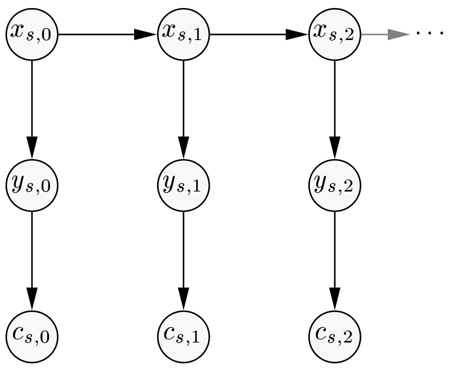


Figure 6.3: Bayesian Counting Model.

rollout the win/loss value of the terminal state is taken as an observation of the win/loss value of all states visited by the rollout.

We let each state, s, have a latent initial underlying value $x_{s,0}$, on which we place a Gaussian prior:

$$p(x_{s,0}) = \mathcal{N}(x_{s,0}; \mu, \sigma^2).$$

The purpose of learning a value of each state is that we are going to adapt the policy used for the rollouts. This means that the distribution over each node's value will be non-stationary as the policy changes. We model this nonstationarity by assuming the value of each state can drift by the addition of Gaussian noise. The distribution of the value of a state in rollout $i+1$ given its value in rollout i is:

$$p(x_{s,i+1}|x_{s,i}) = \mathcal{N}(x_{s,i+1}; x_{s,i}, \tau^2). \tag{6.1}$$

At each time step, i, a node has observed value, y_i, which is distributed as a Gaussian about its underlying value:

$$p(y_{s,i}|x_{s,i}) = \mathcal{N}(y_{s,i}; x_{s,i}, \gamma^2).$$

Each binary win/loss observation (provided by a rollout) gives a sign constraint on the node value: if the terminal position visited by the rollout is a win then we assume that $y_{s,i}$ must be greater than zero and if a loss is observed then $y_{s,i}$ must be less than zero. We use the boolean variable $c_{s,i} \in \{\text{TRUE}, \text{FALSE}\}$to indicate whether the rollout resulted in a win (TRUE) or loss (FALSE):

$$p(c_{s,i}|y_{s,i}) = \mathbb{I}(c_{s,i} \wedge (y_{s,i} > 0)) + \mathbb{I}(\neg c_{s,i} \wedge (y_{s,i} < 0)) \tag{6.2}$$

See Figure 6.3 for the corresponding graphical model. Inference is performed by the sum-product algorithm using the linear Gaussian framework given in Figure 2.5 and using the approximate update given in Figure 2.6 for the message from the sign constraint observation (equation 6.2).

Algorithm

MC planning using this counting model proceeds in a similar manner as UCT. We store the mean and variance of the Gaussian distribution over each node's underlying value. Actions are selected stochastically by sampling from the Gaussian distribution over the value of each available state and selecting the one with the highest sample value. That is $\pi(s,a) = p(a = \text{argmax}_{a_i \in \mathcal{A}(s)} x_{m(s,a_i(s)),t})$. In

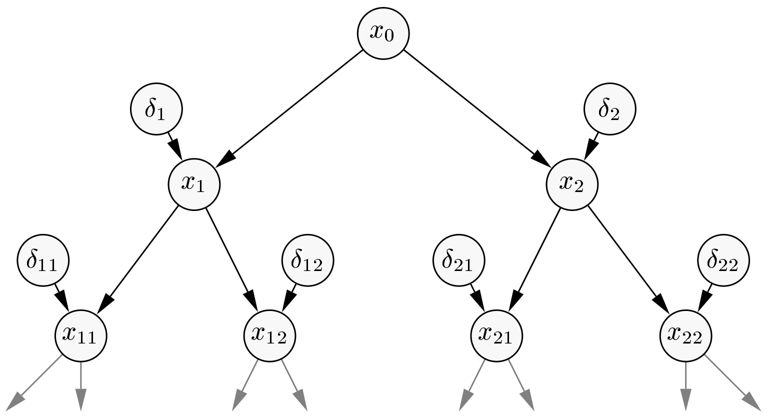


Figure 6.4: Inductive Game Tree Model. Each x variable is the value of a game position and each δ variable is the value of a move. The value of a position is modeled as the sum of the value of the previous position and the value of the move which transitions to the new position.

other words we assume that a player should always move to the most valuable state. Since uncertainty exists about the value of each state the policy is a probability distribution which automatically balances exploration and exploitation if we sample actions stochastically. States with low mean value but high variance may still be explored until their value is determined sufficiently accurately to be worse than other available states, much like what happens with UCT.

The models for all the nodes visited by a rollout are updated when the rollout is completed. The win/loss value of the terminal position of the rollout represents an observation of the sign of the value of each game state visited by the rollout.

6.3.2 Inductive Game Tree Model

Model

Next we consider a generative model of the value of all nodes in the game tree. The Bayesian network corresponding to this model is given in Figure 6.4. We place a Gaussian prior on the value of the root node:

$$p(x_0) = \mathcal{N}(x_0; 0, \sigma_0^2).$$

The value of the move generating state x_a has a value δ_a and the value of the new position created by making a new move is modeled as the sum of the value of the previous position and the value of the move:

$$p(x_i|\delta_i, \mathrm{pa}(x_i)) = \mathbb{I}(\delta_i + \mathrm{pa}(x_i)),$$

where $\mathrm{pa}(x)$ denotes the value of the previous position to the one with value x. We place a Gaussian prior on the value of each move:

$$p(\delta_i) = \mathcal{N}(\delta_i; \mu_i, \sigma_i^2).$$

Therefore the value of the terminal positions at the leaves of the tree is the sum of the values of all the moves made along the way. In other words, we are modeling the sequence of state values encountered in a game as a biased random walk, much like the P-Game described in Section 6.2.2 and used by Kocsis and Szepesvari (2006) to simulate game trees. As in the P-Game trees we treat a win as an observation that the value of the terminal node must be positive and a loss as an observation of the terminal node being negative:

$$p(c_l|y_l) = \mathbb{I}(c_l \wedge (y_l > 0)) + \mathbb{I}(\neg c_l \wedge (y_l < 0)).$$

where y_l is the value of the leaf node. Observation noise with variance γ^2 is included by setting

$$p(y_l|x_l) = \mathcal{N}(y_l|x_l, \gamma^2)$$

where x_l is the (latent) value of the leaf node. Inference proceeds as usual according to the updates in Figures 2.5 and 2.6. Each node stores the Gaussian message from that node towards the root of the tree and the messages are updated in the same manner for the Gaussian hierarchical model of Section 3.3 with cost linear in the depth of the tree.

Note that the game tree model of Chapter 4 was *deductive* in the sense that the values of nodes higher up in the tree are deduced from the values of nodes below them. The model presented in this section is *inductive* in the sense that the values of future states are estimated from evidence from previous states.

Algorithm

For this model we again use the stochastic policy $\pi(s, a) = p(a = \text{argmax}_{a_i \in \mathcal{A}(s)} x_{m(s,a_i(s))})$ for Gaussian distributed x, the value of each state according to the model above. An observation of the sign of the leaf of the tree is made after each rollout.

6.3.3 Dynamic Inductive Game Tree Model

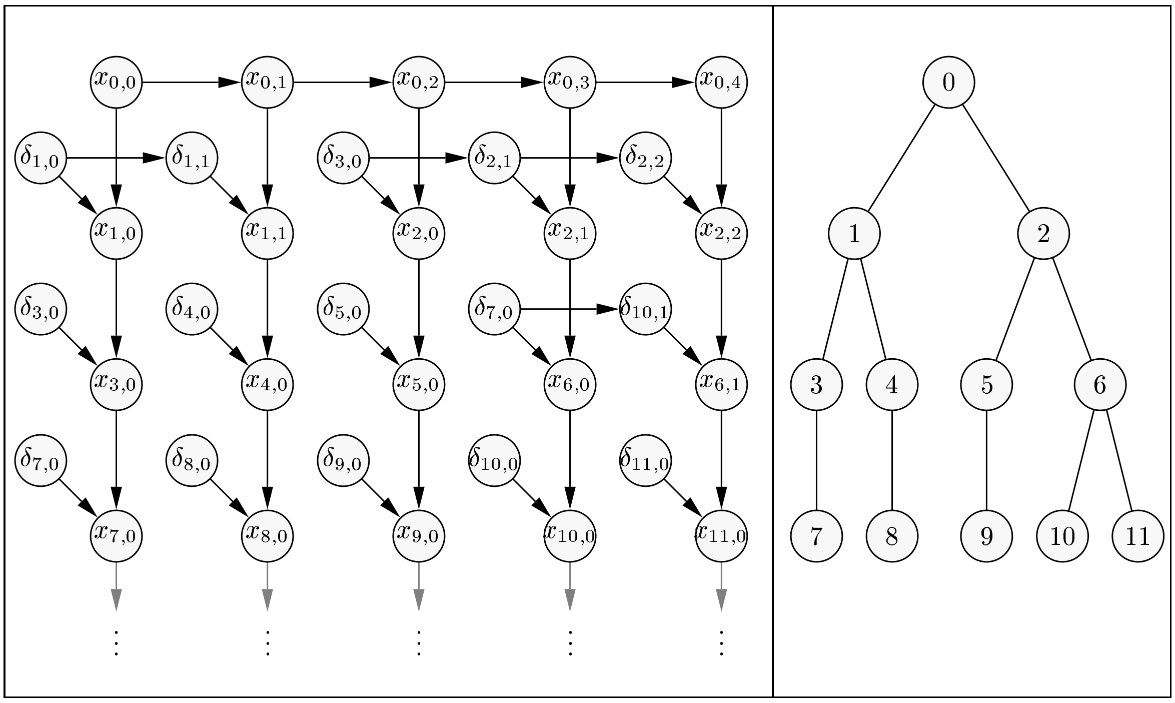


Figure 6.5: Dynamic Inductive Game Tree Model. Each vertical chain of nodes corresponds to a rollout. The horizontal arrows linking the δ (move value) variables indicates that that move is encountered in both of the rollouts linked by the arrow. **Left**: Graphical Model. **Right**: The part of the game tree that has been explored so far by the corresponding set of rollouts.

Lastly, we consider a model of the values of the states in the sequence of rollouts produced by the Monte Carlo planner. This model allows us to adapt the values of each move as rollouts are performed. Again, the values of the states visited in each rollout are assumed to follow a Gaussian random walk. The value of each visit of each game state seen in the sequence of rollouts is denoted x_{ij} where i is a unique index over states and j is the index over the observations of this state in rollouts. We place a Gaussian prior on the value of the root state before any rollouts are carried out:

$$p(x_{0,0}) = \mathcal{N}(x_{0,0}; \mu_0, \sigma_0^2).$$

The value of a move is denoted δ_{ij} for the move which generates position with value x_{ij}. Again, we place a Gaussian prior on the value of a move:

$$p(\delta_{i,0}) = \mathcal{N}(\delta_{i,0}; \mu_{i,0}, \sigma_{i,0}^2).$$

Also, as before, we model the value of the state generated by playing a move as the sum of the value of the previous state and the value of the move:

$$p(x_{ij}|\delta_{ij}, \text{pa}(x_{ij})) = \mathbb{I}(\delta_{ij} + \text{pa}(x_{ij}))$$

where $\text{pa}(x)$ denotes the previous state to x in the same rollout. The more rollouts are played, the more we learn about the value of each move. We take the prior on the value of each move to be the previous value of this move (the last time it was seen in a rollout) with some Gaussian noise added:

$$p(\delta_{i,j}|\delta_{i,j-1}) = \mathcal{N}(\delta_{i,j}; \delta_{i,j-1}, \tau^2).$$

Similarly we model the sequence of values of the root state (as successive rollouts are performed) as a Gaussian random walk in the same manner:

$$p(x_{0j}|x_{0(j-1)}) = \mathcal{N}(x_{0j}; x_{0(j-1)}, \tau^2).$$

The Bayesian network for this model is given in Figure 6.5. Again, inference is performed using the linear Gaussian framework. After each rollout, the constraint on the sign of the value of the last node visited by the rollout (with Gaussian noise with variance γ^2 added) is observed as before (positive for a win and negative for a loss). The cycles in the graph (Figure 6.5) are ignored because we only pass messages forward. Therefore, for each game tree node we only need store the parameters of the distribution over the value, δ_i, of the latest encounter of the move which generates the position.

Algorithm

For this model we use the stochastic policy $\pi(s,a) = p(a = \text{argmax}_{a_i \in \mathcal{A}(s)} \delta_{m(s,a_i(s))})$ to explore the state space. After each rollout we make an observation of the sign of the value of the terminal state of the rollout and then propagate beliefs up the graph to update the values, δ_i, of the moves played during the rollout.

6.4 Results

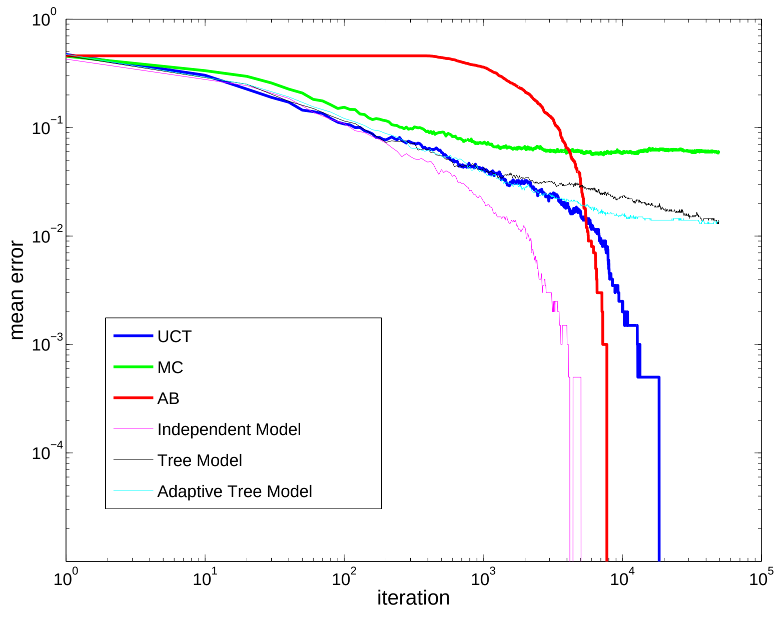
6.4.1 Comparing the Model Performances

Figure 6.6 compares the different models on P-Game trees using the method described in Section 6.2.3. The models have a number of hyperparameters which were tuned automatically by a simple gradient-free hill climbing approach. Each parameter was taken in turn and the error after 1000 rollouts was estimated using 2000 repeats. The same error was calculated for slightly higher and lower values of this parameter and then the parameter was adjusted in the direction of the best result and then the next parameter was adjusted. The values of the parameters for the models are given in Tables 6.1, 6.2 and 6.3. Since the error-rate is very noisy the optimisation process was troublesome and the precise values for these parameters should not be taken too seriously, however these values were found to be better than ones tuned by hand. Further experiments are required to determine the best method of setting the parameters of these models.

Weak Performance of Tree Models

The results show that the tree models (both the simple and dynamic versions) do not perform as well as UCT. Initially the error rate decreases at a similar rate to UCT as more rollouts are performed but UCT ends up converging to the correct solution more rapidly. The tree models tend to converge to a fixed non-zero error rate which is lower than the error rate obtained by Uniform Monte Carlo.

The reason for this state of affairs is as follows. The tree models each provide a generative model of the values of the terminal-nodes of the search tree. Only the values of the terminal nodes



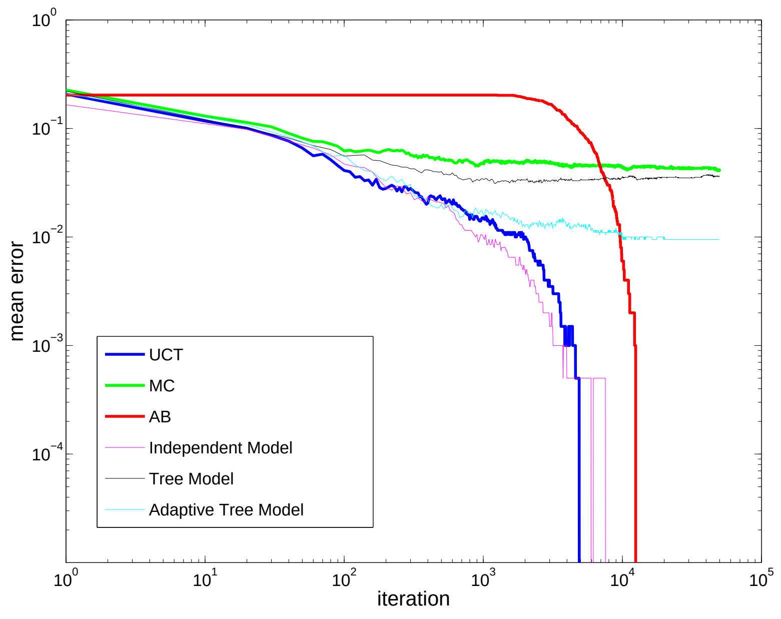


Figure 6.6: Comparing Monte Carlo planning models on PGame trees. The 'independent model' is the counting model of Section 6.3.1, the 'tree model' is the model of Section 6.3.2 and the 'adaptive tree model' is the model of Section 6.3.3. **Top:** $B = 7, D = 7$, **Bottom:** $B = 2, D = 20$

(corresponding to final game positions) are ever observed, the other node values (the x variables) are latent (unobserved) variables. In the exploration algorithm described above these latent values are used to guide search under the interpretation that they represent the underlying value of states in the game tree (so it is good to chose moves which lead to positions with higher latent values).

In general it is true that it is good to move to positions with higher latent values because these values are strongly correlated as we move down the tree so valuable positions do tend to lead to wins more often than losses (hence the fact that the algorithm does converge to an error rate better than uniform Monte Carlo). However, even if we observe the value of every leaf in the search tree by fully expanding it, the latent position values according to the model do not give us the means to solve the game and find the correct move in every position. To correctly predict the values of the internal nodes in the tree we would have to use minimax search based on the predicted leaf values obtained from the generative model (although this would be both intractable and pointless as it would require the expansion of the entire game tree - after which we could just observe the values of the leaves without needing a predictive model). Another way to think about it is to imagine that we know the actual underlying value of all the nodes in the PGame tree (by summing the additive value of each move from the root to the node in question). These values, while helpful in providing a heuristic to guide search, would not specify the correct move in each position - this can only be achieved by determining the truth values of the leaf nodes (by the sign of the accumulated value from the root to the leaf) and then propagating up these logical values by minimax.

For this reason the algorithms described above which use the latent variables in the tree models to guide search can never converge to the correct (deterministic) policy for exploring the tree. Further work would be needed to find a tractable method to correctly use the leaf values predicted by the model with minimax search without having to expand the entire search tree.

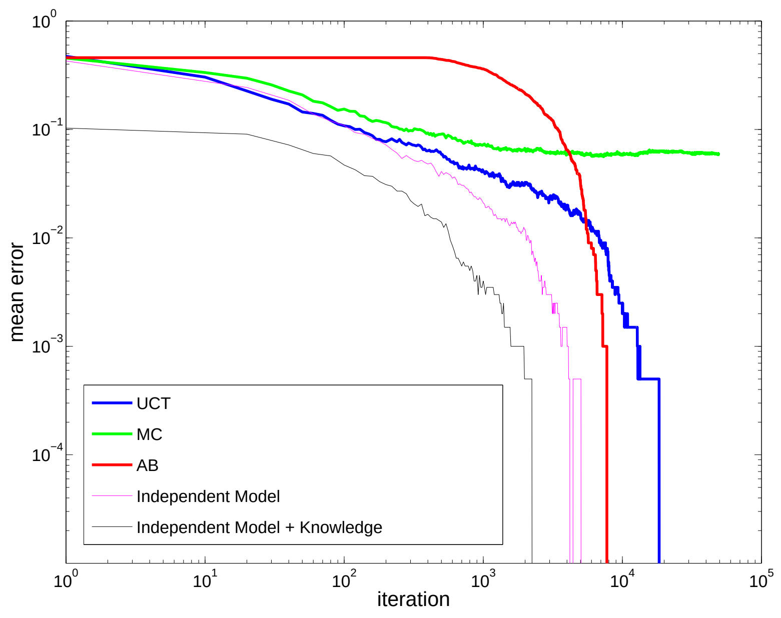
Strong Performance of Bayesian Counting Model

With appropriate setting of the hyperparameters, the simple independent Bayesian counting model (Section 6.3.1) converges faster than UCT for both tree shapes, more so for the wider ($B = 7$) tree. The optimal setting for the dynamics variance, τ^2, seems to be zero for this model. Despite the fact that this model ignores the correlations between node values it performs better than the tree models for the reasons discussed above.

6.4.2 Domain Knowledge

One possible advantage of these models over UCT is that domain knowledge can be included in the form of the prior distribution over the value of each position in the game tree. In the case of the inductive tree models prior knowledge can be included as prior distributions over the additive value of each move. Experiments were performed in order to simulate the addition of domain knowledge using the independent dynamic counting model described in Section 6.3.1. Recall that when each P-Game tree is created (see Section 6.2.2) a value for each move must be generated by drawing randomly from the interval $[0, 1]$ for MAX moves and $[-1, 0]$ for MIN moves. Domain knowledge is simulated by setting the prior mean value of each position to the additive value of the previous move. In a real game tree search these values could, in principle, be provided by a model which is trained from previous games.

It was unclear at the outset if this knowledge of the additive value of each move would help the



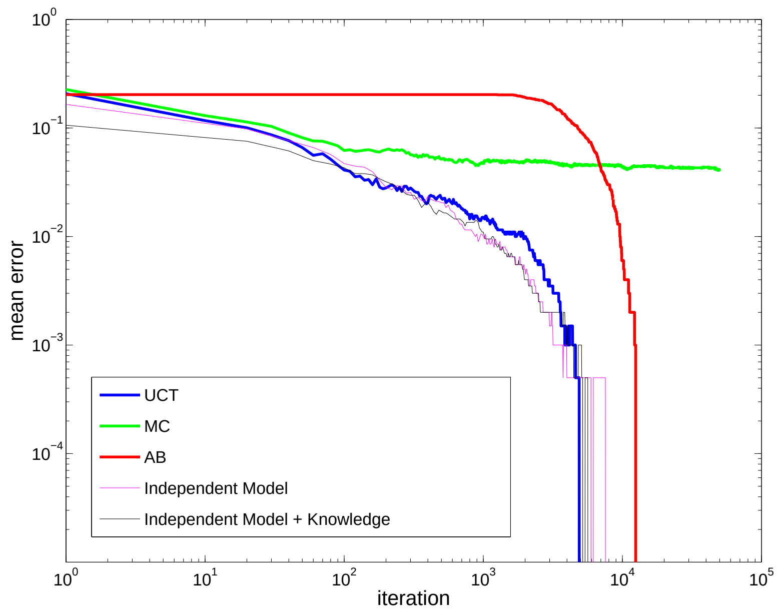


Figure 6.7: Testing the performance on PGame trees of the independent Bayesian counting model (Section 6.3.1) with prior knowledge. The 'knowledge' is included by setting the mean of the prior value of each position to the value of the corresponding move used when generating the P-Game tree (see Section 6.2.2). **Top:** $B = 7, D = 7$, **Bottom:** $B = 2, D = 20$

Parameter	Value for $B = 7, D = 7$	Value for $B = 2, D = 20$
σ	1.6	0.7
γ	1	1
τ	0	0

Table 6.1: Hyperparameters of Independent Dynamic Counting Model as trained by minimising error rate after 1000 iterations.

algorithm converge to the correct solution faster. It seemed likely that using these values might lead to an initially better policy (with lower error rate than if no knowledge was used). This is because the correlation in the values of nodes in the search tree means that that moves which add more value are more likely to lead to win positions. However, it was unclear if the algorithm would ultimately converge faster by making use of this knowledge to guide search. The experiments show, that both a lower initial error rate and a faster convergence are observed if we make use of the PGame move values in the manner described above. In other words, better moves tend to lead to more informative playouts.

The variance, σ^2, was set by hand to 0.25 (this value was found to work well for both trees). The results of this experiment are shown in Figure 6.7. For the tree with width and depth equal to 7, setting the prior in this way leads to much faster convergence to the correct solution. For the binary tree, adding this knowledge makes less difference.

This result is potentially useful for future work as it may be easier to obtain a model of the additive value of a move rather than a model of the absolute value of a position as the effect of a move is likely to be local (as suggested by the strong results of Chapter 3)

6.5 Discussion and Future Work

In this chapter we presented some initial findings of experiments into using explicit models of the value of nodes in game trees to guide Monte Carlo planning algorithms. The simplest model which treats each node in the tree independently performed the best, and for good settings of the parameters converged to the correct solution more rapidly than UCT. We also showed that domain knowledge can be included successfully by using it to set the prior distribution over the value of game positions. Future work could look to applying this algorithm to Go, inspired by the recent success of MoGo (Gelly *et al.*, 2006), and in particular combining domain knowledge with the planning algorithm. It might also be possible to learn from Monte Carlo rollouts themselves, in a similar manner to the method by which in Chapter 4 the model was able to learn from previous search tasks how to solve future unseen problems more efficiently. In that Chapter the model did not perform very well, in the sense that some problems took more time to solve after learning than before. However, the algorithms presented here correctly take account of uncertainty in order to balance exploration and exploitation so perhaps these models would be more robust after learning from small amounts of data.

Parameter	Value for $B = 7, D = 7$	Value for $B = 2, D = 20$
Move mean value, μ_i (MAX to move)	0.6	1.6
Move mean value, μ_i (MIN to move)	-0.4	-1.7
Move standard deviation, σ_i (tied across all moves)	0.2	0.1
Root prior, σ_0	0.015	0.21
γ	1	1

Table 6.2: Hyperparameters of Inductive Tree Model as trained by minimising error rate after 1000 iterations (γ was fixed to 1).

Parameter	Value for $B = 7, D = 7$	Value for $B = 2, D = 20$
Move mean value, μ_i (MAX to move)	-0.1	1.0
Move mean value, μ_i (MIN to move)	0.2	-0.1
Move standard deviation, σ_i (tied across all moves)	1	0.4
Root prior, σ_0	1.7	1.1
Dynamics factor, τ (tied across all moves)	0.02	0.01
γ	1	1

Table 6.3: Hyperparameters of Dynamic Inductive Tree Model as trained by minimising error rate after 1000 iterations (γ was fixed to 1). All τ parameters tied.

Chapter 7

Conclusions

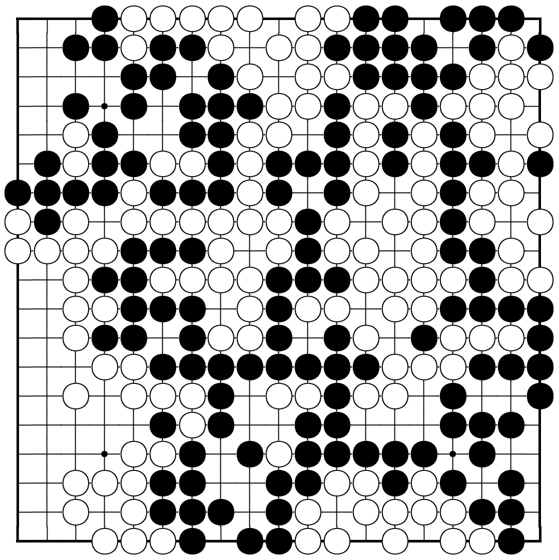


Figure 7.1: Final position from the game. Shusaku (black) won by 2 points.

Go is a game of perfect information. However, computers (and humans) have limited computational speed so from the point of view of any human or computer player there is uncertainty about the future course of the game due to the sheer complexity of the game tree. At the start of this thesis I proposed that probability theory might be a useful tool for representing and managing this uncertainty. I have presented a number of probabilistic models which were applied to different aspects of Go and this Chapter summarises some of the main findings.

7.1 Conclusions

In Chapter 3 I considered the task of obtaining a probability distribution over moves in a Go position and hence predicting Go moves. A great deal of knowledge about playing the game has been passed down over thousands of years. This knowledge is implicit in abundant game records. In order to build a model, each move was represented by the exact pattern of stones in the local area of the board centered on the proposed move. These patterns were automatically harvested from records of games between expert players. Each board position in a game record contains a set of available legal moves, only one of which was chosen by the human player. This information was used to learn values for each move (pattern) using a Bayesian ranking model. The simplicity of this system meant that it was sufficiently fast that hundreds of thousands of games could be used for training. The system was able to predict 34% of expert moves correctly and ranked the expert move within the top 10 moves on the board 76% of the time. These results were better than other published results of the time but were subsequently improved on by Coulom (2007). Using a simple linear model in conjunction with the pattern system I was able to obtain similar move ranking performance to Coulom. I also presented a hierarchical model which 'smooths' over the information from patterns of different sizes to take account of the fact that small patterns are seen more frequently but contain less information than larger patterns. This also improved performance over the original model.

Next, in Chapter 4 I focused on using probabilities to guide search in order to solve local tactical Go problems. Such problems are solved by searching the game tree until sufficient information is gathered that it is possible to prove by logical deduction whether a particular goal can be achieved. Before a proof is found there is uncertainty about the solution and probabilities are used to represent this uncertainty and a probability distribution over the possible solutions is inferred based on evidence propagated up from the leaves, under the assumption of independence between the values of available moves in a position. I showed that, when used to guide search, this model gives the same performance, on a set of Go capture problems, as the currently popular method for solving this type of problem (Proof Number Search). However, the independence assumption led to the values of nodes in the search tree taking on extreme, unrealistic values, which suggests that a better model would take account of correlations between the values of available moves. Next, I applied the pattern system developed in Chapter 3 to assign prior beliefs on the values of leaf nodes in the search tree so prior knowledge could be used to speed up search. The resulting composite system was capable of learning from previous search tasks how to solve future, unseen, problems more rapidly. Unfortunately there was also a difficulty: some problems took longer to solve after the model was trained than before, which suggests that the pattern matching system was a poor model for this task (where the data is much more limited than for the task of move prediction). By using the hierarchical model of Chapter 3 I partially addressed this issue; however further model improvements are necessary.

One source of information in the records of expert games which was not exploited in Chapter 3 was the final outcome of each game. In Chapter 5 I returned to the game records and presented a simple model for learning to predict the final territory outcome of games given a mid-game position. The model was a conditional Markov random field with the grid topology of the Go board where each variable represented the final territory outcome (black or white) at a particular point on the board. The couplings and biases of this Boltzmann machine depended on the current arrangement of stones on the board. Loopy belief propagation and a generalised Swendsen-Wang process were used for inference. The Swendsen-Wang process is a Gibbs sampling algorithm which replaces the original model by a more complex model such that the marginal distributions of the variables of interest remain unchanged. The more complex model has the property that it can be sampled from more efficiently. Go players confirm that the territory outcomes produced by this system are reasonable. A weakness of the model as it stands is that it cannot accurately predict the life and death of stones, apart from in simple cases, so cannot score final board positions accurately.

In the final Chapter I looked at a technique which addresses, in a unified manner, some of the shortcomings of the specialised models presented in earlier chapters: Monte Carlo planning. This method determines the value of Go positions by observing the outcomes of simulated games, played with some (usually stochastic) policy, from the position in question to the end of the game. Recently, Go researchers have achieved success with algorithms which bootstrap themselves by adapting the policy used for the Monte Carlo simulations based on the information gathered from previous simulations (Gelly *et al.*, 2006). Existing Monte Carlo planning algorithms were tested on two types of game trees: 'random trees' where the values of the leaves of the tree are assigned at random with the same probability that each leaf is a win or loss, and 'P-Game trees' where the sequence of the values of positions encountered in a game is modeled as a biased random walk. The Monte Carlo techniques failed on the random trees but found the best moves faster than alpha-beta search on the P-Game trees. In other words, Monte Carlo planning techniques require some 'smoothness' in the underlying values of nodes in the game tree in order to be successful (which contradicts the independence assumption made by the model of Chapter 4). Next, I suggested some Bayesian alternatives to these algorithms and tested their performance empirically on artificial game trees. With appropriate settings for the hyperparameters, the simplest of these models (a simple dynamic Gaussian model) achieved lower move error rates in less time than the current state of the art technique (UCT). Also I showed that this model can, in principle, be used to combine prior knowledge with searching the state space in order to improve performance.

7.2 Contributions of this Thesis to the State-of-the-Art in Computer Go

At the start of this thesis I pointed out that computer Go research has recently become dominated by Monte Carlo search techniques (Section 1.4). Whereas in the past the focus was on systems using a great deal of static knowledge, now the focus is on dynamic techniques that adapt to the position at hand. Current Go research is interested in finding efficient ways to search the game tree efficiently and extracting the relevant information from it. I believe that in order to produce a strong Go program on the full size board we need to re-introduce knowledge into Go programs and combine this knowledge with the efficient Monte Carlo search approaches. In this thesis I presented a number of novel probabilistic models which are relevant to this task.

The pattern system described in Chapter 3 provides an accurate distribution over Go moves in a position. If we are to extract the relevant information from the vast Go game tree given only limited computational resources (as discussed in Section 1.5) then it is essential to be selective about which parts of the tree are explored and the pattern model provides a way to do this. Coulom (2007) successfully used a pattern matching system to guide the stochastic rollouts for evaluating Go positions for UCT. The pattern system developed in Chapter 3 could be applied in a similar way since it provides a probability distribution over the legal moves in a position - exactly what is needed - and, crucially, inference is fast enough to be run in the inner loop of the search in this way. In addition, it is worth mentioning that the pattern system presented in this thesis is in general useful for proposing Go moves before performing more expensive analysis and currently I am working on applying the pattern system to prune moves at the top of the game tree in the hope of improving the performance of UCT on the full size Go board. It is hoped that the strengths of the pattern system are somewhat orthogonal to the strengths of UCT so the combination might be powerful. In particular, UCT is very weak at the opening on the full size board whereas the pattern system plays at expert level in the opening.

Most Go programs (apart from the recent Monte Carlo programs) use tactical search techniques to determine whether various properties of a Go position hold (such as whether a particular group will live). In Chapter 4 I presented a novel search algorithm which uses probabilities to guide search and performs similarly to Proof Number Search, a current popular method to solve this type of problem. This result has not been previously shown and may prove useful for future work in this area. While the latest Monte Carlo programs do not currently use exact search methods, I believe that in the future they may have to in order to achieve expert level strength on the full size board.

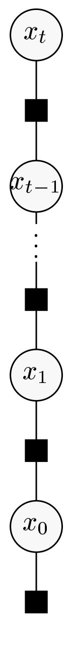
Chapter 5 presents a new approach to predicting the final territory outcome of Go games. The model provides a probability distribution over final game outcomes, as is provided by Monte Carlo rollouts, so the model provides an interesting counterpart to the Monte Carlo planning approach to position evaluation. The Boltzmann machine model is a static approach to evaluation whereas Monte Carlo rollouts provide a dynamic evaluation function. However, both approaches are stochastic (as the Boltzmann machine requires a Monte Carlo algorithm for inference) and become more accurate if more computational effort is put in. Therefore, one might speculate that models of the type presented in Chapter 5 could be applied in place of the stochastic rollouts of Monte Carlo planning and UCT used to determine which parts of the tree to develop and explore further. In other words samples from static Boltzmann models could provide a fast stochastic evaluation function for the full size Go board which could be combined with UCT in the same manner as MC rollouts are used in current approaches. Also, in addition to the possible application to Monte Carlo Go, another potentially useful contribution of this work is that even the simplest Boltzmann machine model could replace the influence functions used to estimate territory in most traditional Go programs. In that case I would envisage that one would condition the territory prediction on the results of local tactical searches, such as the output from a dedicated life and death module, which would improve the prediction (by pinning the state of some of the territory variables and then running the Gibbs sampler).

The experiments of Chapter 6 look at the Monte Carlo planning and the UCT algorithm. An important contribution of this chapter is that I show that smoothness in the underlying values of nodes in the game tree is important to allow these methods to work. This suggests that this fact could be exploited in order to design better algorithms. Next, I show that, at least for artificial game trees, it is possible to improve on the performance of UCT by explicitly modelling the value of a game position. This algorithm achieves a balance between exploration and exploitation in quite a different way to UCT: simply sampling from the distributions over each moves value and choosing the move corresponding to the highest sample seems to work well. In general, this chapter represents a general new direction to take research into exploring game trees using UCT-like algorithms which incorporate Bayesian models of the values of nodes in the game tree. Such models allow prior knowledge to be included in a principled way.

7.3 Future Work

I believe that all of the models presented in this thesis would benefit from extensions. An obvious direction for future work would be to further integrate the work presented in the different chapters. One possible combination is that the move prediction performance of the pattern system could probably be improved by the addition of some territory information. This could be provided by the Boltzmann machine models of Chapter 5 or the Monte Carlo algorithms of Chapter 6. The pattern system could also be used to improve the territory predictions of the Boltzmann machines and a step in this direction has already been taken by Sanner *et al.* (2007). Another possible combination would be to integrate the output of the local tactical search algorithms of Chapter 4 with the territory prediction models: if a stone is proved to be alive or dead then its life/death status could be pinned and the Boltzmann machine used to predict the territory outcome for the rest of the board. Many other such model combinations are possible, some of which have been discussed in the individual chapters. In the last few paragraphs of this thesis I outline what I consider to be some possible (speculative) new directions, above and beyond simple extensions of the work already presented.

In my opinion, Monte Carlo planning techniques are an interesting future direction for research. Despite the simplicity of the approach, this method has outperformed any other current method, including very complex expert tuned programs, at the task of playing Go (Gelly *et al.*, 2006). The world's strongest Go program uses Monte Carlo planning to select moves with only very simple Go knowledge included. This system effectively learns how to play Go from scratch with every move. If a method could be found to incorporate domain knowledge in the Monte Carlo planning algorithm then perhaps performance could be improved. I took a tentative step in this direction by experimenting with Bayesian versions of Monte-Carlo planning algorithms in Chapter 6 that, in principle at least, allow domain knowledge to be included via the prior distributions on the values of moves and positions. Future work would involve applying these techniques to Go and using a model (such as the pattern system) to provide a prior distribution on the value of each move so domain knowledge could be included. Perhaps this domain knowledge would be learned from expert games or perhaps it would be learned from previous Monte Carlo rollouts.



Each Monte Carlo simulation could be a considered a sample from the joint distribution over game histories. This distribution can be represented by a factor graph as shown on the left (T. Minka, personal communication) where each variable x_t is the state of the game at time t. The factors enforce the rules of the game as well as the normative constraints which determine which states are preferred by the players. Other approximate schemes, besides Monte Carlo, for performing inference on this (ideal) game model could be investigated as a possible direction for future research, perhaps starting with simpler games than Go.

7.4 Summary

I have presented a number of probabilistic models for various aspects of the game of Go. Much work remains to be done before we have a strong Go playing computer but I believe that Bayesian machine learning provides a useful set of tools for modelling uncertainty in the game of Go.

Appendix A

Moment Matching Derivations

A.1 Moments of a Fragmented Gaussian

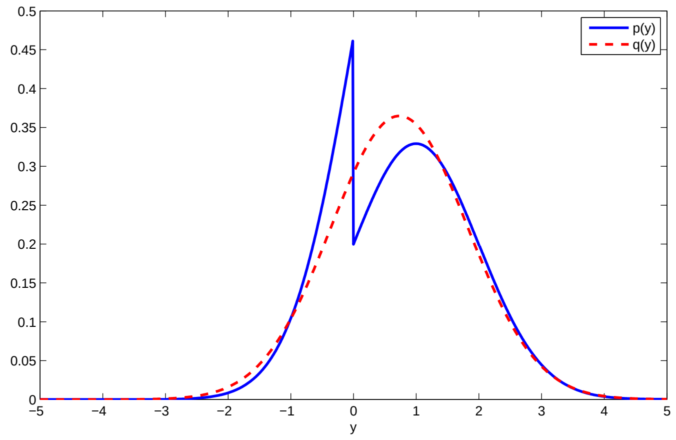


Figure A.1: Fragmented Gaussian, $p(y)$, and the Gaussian with matching moments, $\hat{p}(y)$. Here, $\mu = \sigma = 1$ and $q = 0.3$.

We wish to approximate the function

$$p(y) = \frac{\left[q\mathbb{I}(y > 0) + (1-q)\mathbb{I}(y < 0)\right] \cdot \mathcal{N}(y; \mu, \sigma^2)}{Z(\mu, \sigma, q)}$$

which we term a 'fragmented' Gaussian by the Gaussian, $\hat{p}(y)$, closest in terms of KL divergence (see Figure A.1). This is achieved by setting the moments of $\hat{p}(y)$ equal to the moments of $p(y)$, that is,

$$\hat{p}(y) = \mathrm{MM}\left[p(y)\right].$$

The first moment of $p(y)$ is given by

$$\langle y \rangle_{p(y)} = \frac{q}{Z} \cdot \int_0^\infty y\mathcal{N}\left(y; \mu, \sigma^2\right) dy + \frac{1-q}{Z} \cdot \int_{-\infty}^0 y\mathcal{N}\left(y; \mu, \sigma^2\right) dy \tag{A.1}$$

and the second moment by

$$\left\langle y^2 \right\rangle_{p(y)} = \frac{q}{Z} \cdot \int_0^\infty y^2 \mathcal{N}\left(y;\mu,\sigma^2\right) dy + \frac{1-q}{Z} \cdot \int_{-\infty}^0 y^2 \mathcal{N}\left(y;\mu,\sigma^2\right) dy. \tag{A.2}$$

Let $I_1 = \int_0^\infty y\mathcal{N}\left(y;\mu,\sigma^2\right) dy$ and $I_2 = \int_0^\infty y^2\mathcal{N}\left(y;\mu,\sigma^2\right) dy$. These integrals are calculated by considering the derivatives of $1 - \Phi(0,\mu,\sigma^2) = \int_0^\infty \mathcal{N}\left(y;\mu,\sigma^2\right) dy$. Starting with the first derivative,

$$\begin{aligned}
\frac{\partial}{\partial\mu}\left[1 - \Phi\left(0,\mu,\sigma^2\right)\right] &= \frac{\partial}{\partial\mu}\int_0^\infty \mathcal{N}\left(y;\mu,\sigma^2\right) dy \\
&= \int_0^\infty \frac{\partial}{\partial\mu}\mathcal{N}\left(y;\mu,\sigma^2\right) dy \\
&= \int_0^\infty \frac{y-\mu}{\sigma^2}\mathcal{N}\left(y;\mu,\sigma^2\right) dy \\
&= \frac{I_1 - \mu\Phi\left(\frac{\mu}{\sigma}\right)}{\sigma^2}.
\end{aligned}$$

We also have that

$$\frac{\partial}{\partial\mu}\left[1 - \Phi\left(0,\mu,\sigma^2\right)\right] = \frac{1}{\sigma}\mathcal{N}\left(\frac{\mu}{\sigma}\right),$$

so

$$I_1 = \mu \cdot \Phi\left(\frac{\mu}{\sigma}\right) + \sigma\mathcal{N}\left(\frac{\mu}{\sigma}\right) \tag{A.3}$$

Now we consider the second derivative:

$$\begin{aligned}
\frac{\partial^2}{\partial\mu^2}\left[1 - \Phi\left(0,\mu,\sigma^2\right)\right] &= \int_0^\infty \frac{\partial^2}{\partial\mu^2}\mathcal{N}\left(y;\mu,\sigma^2\right) dy \\
&= \int_0^\infty \frac{y^2 - 2\mu y + \mu^2 - \sigma^2}{\sigma^4}\mathcal{N}\left(y;\mu,\sigma^2\right) dy \\
&= \frac{1}{\sigma^4}\left[I_2 - 2\mu I_1 + (\mu^2 - \sigma^2)\Phi\left(\frac{\mu}{\sigma}\right)\right],
\end{aligned}$$

which is also equal to,

$$\frac{\partial^2}{\partial\mu^2}\left[1 - \Phi\left(0,\mu,\sigma^2\right)\right] = \frac{-\mu}{\sigma^2} \cdot \frac{1}{\sigma}\mathcal{N}\left(\frac{\mu}{\sigma}\right),$$

leading to,

$$I_2 = \left(\mu^2 + \sigma^2\right) \cdot \Phi\left(\frac{\mu}{\sigma}\right) + \mu\sigma\mathcal{N}\left(\frac{\mu}{\sigma}\right). \tag{A.4}$$

The other integrals are derived using the same technique to give

$$\begin{aligned}
\int_{-\infty}^0 y\mathcal{N}\left(y;\mu,\sigma^2\right) dy &= \mu \cdot \Phi\left(\frac{-\mu}{\sigma}\right) - \sigma\mathcal{N}\left(\frac{\mu}{\sigma}\right) && \text{(A.5)} \\
\int_{-\infty}^0 y^2\mathcal{N}\left(y;\mu,\sigma^2\right) dy &= \left(\mu^2 + \sigma^2\right) \cdot \Phi\left(\frac{-\mu}{\sigma}\right) - \mu\sigma\mathcal{N}\left(\frac{\mu}{\sigma}\right). && \text{(A.6)}
\end{aligned}$$

Now using (A.1), (A.2), (A.3), (A.4) , (A.5) and (A.6) we arrive at the first and second moments of the fragmented Gaussian:

$$\begin{aligned}
\langle y \rangle &= \frac{1}{Z}\cdot\left[\Phi\left(\frac{\mu}{\sigma}\right)(2q\mu-\mu)+\mathcal{N}\left(\frac{\mu}{\sigma}\right)(2q\sigma-\sigma)+\mu-q.\mu\right] && \text{(A.7)}\\
\langle y^2 \rangle &= \frac{1}{Z}\cdot\left[\Phi\left(\frac{\mu}{\sigma}\right)\left(\mu^2+\sigma^2\right)(2q-1)+\mathcal{N}\left(\frac{\mu}{\sigma}\right)\sigma\mu\,(2q-1)+\left(\mu^2+\sigma^2\right)(1-q)\right] && \text{(A.8)}\\
Z &= (1-q)\cdot\left(1-\Phi\left(\frac{\mu}{\sigma}\right)\right)+q\cdot\Phi\left(\frac{\mu}{\sigma}\right). && \text{(A.9)}
\end{aligned}$$

The mean and standard deviation of the approximating distribution, $\hat{p}(y)=\mathcal{N}(y;\mu_q,\sigma_q^2)$, are thus,

$$\begin{aligned}
\mu_q &= \langle y \rangle\\
\sigma_q^2 &= \langle y^2 \rangle-\mu_q^2.
\end{aligned}$$

A.2 Soft Update Equations for a Beta-Bernoulli Model

We wish to approximate the function

$$p(q) = \frac{1}{Z(\alpha,\beta,q_m)}\left\{q_m\cdot q+(1-q_m)\cdot(1-q)\right\}\cdot\text{Beta}(q;\alpha,\beta)$$

where

$$Z(\alpha,\beta,q_m)=\frac{\alpha}{\alpha+\beta}\cdot(2q_m-1)+1-q_m$$

by a Beta distribution $\hat{p}(q)=\text{Beta}(q;\alpha',\beta')$. We do this by finding the Beta distribution with the same moments as $p(q)$, that is,

$$\hat{p}(q)=\text{Beta}(q;\alpha',\beta')=\text{MM}\left[p(q|\alpha,\beta,q_m)\right].$$

This does not correspond to the distribution closest in terms of K.L divergence (that would be true of a Gaussian distribution) but as we will see, still results in a very close approximation.

A.2.1 Moment Calculations

Moments of Beta Distribution

The nth moment of the Beta distribution is given by:

$$\langle q^n\rangle_{q\sim\text{Beta}(\alpha,\beta)}=\frac{\Gamma(\alpha+\beta)\Gamma(\alpha+n)}{\Gamma(\alpha)\Gamma(\alpha+\beta+n)}$$

where $\Gamma(x)$ is the Gamma function where,

$$\Gamma(x+1)=x\cdot\Gamma(x),$$

so

$$\begin{aligned}
\Gamma(x+2) &= x(x+1)\cdot\Gamma(x),\\
\Gamma(x+3) &= x(x+1)(x+2)\cdot\Gamma(x).
\end{aligned}$$

These results allow us to derive:

$$\begin{aligned}
\langle q \rangle_{q\sim\mathrm{Beta}(\alpha,\beta)} &= \frac{\alpha}{\alpha+\beta}, & \text{(A.10)}\\
\left\langle q^2 \right\rangle_{q\sim\mathrm{Beta}(\alpha,\beta)} &= \frac{\alpha(\alpha+1)}{(\alpha+\beta)(\alpha+\beta+1)}, & \text{(A.11)}\\
\left\langle q^3 \right\rangle_{q\sim\mathrm{Beta}(\alpha,\beta)} &= \frac{\alpha(\alpha+1)(\alpha+2)}{(\alpha+\beta)(\alpha+\beta+1)(\alpha+\beta+2)}.
\end{aligned}$$

Moments of Posterior

Now we can determine the moments of $p(q)$. For the first moment we have:

$$\begin{aligned}
\langle q \rangle_{p(q)} &= \frac{1}{Z}\int q\cdot\{q_m\cdot q+(1-q_m)\cdot(1-q)\}\cdot\mathrm{Beta}(q;\alpha,\beta)dq\\
&= \frac{1}{Z}\left[(2q_m-1)\left\langle q^2\right\rangle_{q\sim\mathrm{Beta}(\alpha,\beta)}+(1-q_m)\langle q\rangle_{q\sim\mathrm{Beta}(\alpha,\beta)}\right]\\
&= \frac{1}{Z}\cdot\frac{\alpha(q_m\alpha+q_m+\beta-q_m\beta)}{(\alpha+\beta)(\alpha+\beta+1)} & \text{(A.12)}
\end{aligned}$$

and for the second moment:

$$\begin{aligned}
\left\langle q^2\right\rangle_{p(q)} &= \frac{1}{Z}\int q^2\cdot\{q_m\cdot q+(1-q_m)\cdot(1-q)\}\cdot\mathrm{Beta}(q;\alpha,\beta)dq\\
&= \frac{1}{Z}\left[(2q_m-1)\left\langle q^3\right\rangle_{q\sim\mathrm{Beta}(\alpha,\beta)}+(1-q_m)\left\langle q^2\right\rangle_{q\sim\mathrm{Beta}(\alpha,\beta)}\right]\\
&= \frac{1}{Z}\cdot\frac{\alpha(\alpha+1)(q_m\alpha+2q_m-q_m\beta+\beta)}{(\alpha+\beta)(\alpha+\beta+1)(\alpha+\beta+2)} & \text{(A.13)}
\end{aligned}$$

A.2.2 Update Equations

In order to find the desired approximation we must solve the moment matching equations for α' and β':

$$\begin{aligned}
\langle q\rangle_{\hat{p}(q)=Beta(q;\alpha',\beta')} &= \langle q\rangle_{p(q)}\\
\left\langle q^2\right\rangle_{\hat{p}(q)=Beta(q;\alpha',\beta')} &= \left\langle q^2\right\rangle_{p(q)}
\end{aligned}$$

From A.10 and A.11 we obtain the update equations:

$$\begin{aligned}
\alpha' &= \frac{\langle q\rangle_{p(q)}\left(\langle q\rangle_{p(q)}-\left\langle q^2\right\rangle_{p(q)}\right)}{\left\langle q^2\right\rangle_{p(q)}-\langle q\rangle_{p(q)}^2}\\
\beta' &= \alpha'\frac{1-\langle q\rangle_{p(q)}}{\langle q\rangle_{p(q)}}
\end{aligned}$$

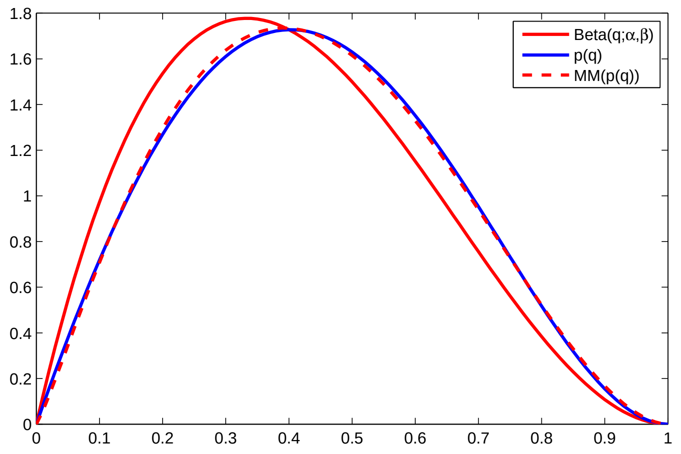


Figure A.2: The true posterior distribution $p(q)$ for a Beta prior $\sim$ Beta($\alpha = 2, \beta = 3$) is plotted as a blue line. The downward belief, q_m, is set to 0.7. The prior is plotted in a solid red line, and the approximate posterior as a dashed red line. Note that the approximations appears to be extremely good.

Appendix B

AND/OR Update Equations

We represent the Bernoulli messages in the search tree factor graph as log-odds ratios, $\text{logodds}(q) := L(q) := \ln(\frac{q}{1-q})$. In this appendix we derive the message calculations for the search tree factor graph for the log-odds domain. This ensures numerical stability because the log-odds representation means that the full range of the double precision floating point number can be used and good precision is possible for probabilities approaching 0 or 1.

B.1 Odds Ratios

First we derive the AND / OR rules in terms of odds ratios. The odds ratio of a probability, q, is given by $o(q) = \frac{q}{1-q}$ and we can invert this to give.

$$q = \frac{o(q)}{1 + o(q)}. \tag{B.1}$$

Now the update equations are calculated by substituting this result into the update equations of Section 4.3.1. Let $q_\wedge$ be the parameter of the Bernoulli message to the parent of an AND factor, and $q_\vee$ be the parameter of the Bernoulli message to the parent of an OR factor. We denote the parameter of the Bernoulli message to or from a variable n as $q(n)$ and for notational convenience we will often drop the functional dependence of the parameter, denoting $q_n := q(n)$

Upward Messages

AND

From 4.4 we have $q^{\mathrm{u}}_\wedge(n) = \prod_{i \in \mathrm{ch}(n)} q^{\mathrm{u}}_i$ and using B.1 we obtain the AND update in terms of odds ratios as:

$$o(q^{\mathrm{u}}_\wedge) = \frac{\prod_i o(q^{\mathrm{u}}_i)}{\prod_i (1 + o(q^{\mathrm{u}}_i)) - \prod_i o(q^{\mathrm{u}}_i)} \tag{B.2}$$

for $i \in \mathrm{ch}(n)$, the set of children. This is slow to calculate but using $q^{\mathrm{u}}_\wedge = q^{\mathrm{u}}_x \cdot q^{\mathrm{u}}_y$ for two children, x and y, we obtain the AND incremental update in terms of odds ratios follows from substituting B.1 to give:

$$o(q^{\mathrm{u}}_\wedge) \quad = \quad \frac{o(q^{\mathrm{u}}_x) \cdot o(q^{\mathrm{u}}_y)}{1 + o(q^{\mathrm{u}}_x) + o(q^{\mathrm{u}}_y)} \tag{B.3}$$

which is re-arranged to give the ANTI-AND inverse update:

$$o(q^{\mathrm{u}}_{\wedge\backslash y}) = o(q^{\mathrm{u}}_x) = \frac{1 + o(q^{\mathrm{u}}_y)}{\frac{o(q^{\mathrm{u}}_y)}{o(q^{\mathrm{u}}_\wedge)} - 1}. \tag{B.4}$$

Now we can update the message parameter $q^{\mathrm{u}}_\wedge$ to the parent (after the parameter, q^{u}_y, of a single child changes during the UpdateBeliefs phase) by first applying B.4 to remove the previous contribution from that child and then applying B.3 to add in the updated contribution from node y.

OR

Similarly we can translate the OR relationship $q^{\mathrm{u}}_\vee(n) = 1 - \prod_i (1 - q^{\mathrm{u}}_i)$ for $i \in \mathrm{ch}(n)$ into the log-odds domain as:

$$o(q^{\mathrm{u}}_\vee) = \prod_i (1 + o(q^{\mathrm{u}}_i)) - 1. \tag{B.5}$$

The OR for two children can be written as $q_\vee = q_x + q_y - q_x \cdot q_y$ so the OR incremental update is:

$$o(q^{\mathrm{u}}_\vee) = o(q^{\mathrm{u}}_x) + o(q^{\mathrm{u}}_y) + o(q^{\mathrm{u}}_x) \cdot o(q^{\mathrm{u}}_y) \tag{B.6}$$

which is re-arranged to give the ANTI-OR inverse update:

$$o(q^{\mathrm{u}}_{\vee\backslash y}) = o(q^{\mathrm{u}}_x) = \frac{o(q^{\mathrm{u}}_\vee) - o(q^{\mathrm{u}}_y)}{1 + o(q^{\mathrm{u}}_y)}. \tag{B.7}$$

which can be used to swap out the previous contribution from a child as with the AND factor.

Downward Messages

AND

We use equations 4.10 and B.1 to obtain the Bernoulli paramater of the downward message $q^{\mathrm{d}}_\wedge(c)$ from an AND factor to a child node, c, in terms of the parameter q^{d}_n of the message from the parent node, n, and the parameters, q^u_i, of the upward messages from the other children in the odds domain:

$$o(q^{\mathrm{d}}_\wedge(c)) = 1 + (o(q^{\mathrm{d}}_n) - 1) \prod_{i \neq c} \left(\frac{o(q^{\mathrm{u}}_i)}{1 + o(q^{\mathrm{u}}_i)} \right) \tag{B.8}$$

Now from equation B.2 we have the relationship:

$$\prod_{i \in \mathrm{ch}(n)} \left(\frac{o(q^{\mathrm{u}}_i)}{1 + o(q^{\mathrm{u}}_i)} \right) = \frac{o(q^{\mathrm{u}}_n)}{1 + o(q^{\mathrm{u}}_n)}$$

where $q^{\mathrm{u}}_n = q^u_\wedge(n)$ is the parameter of the upward message from the AND factor to its parent node, n. Substituting this into B.8 leads to the incremental update in terms of the upward message from the factor, $q^{\mathrm{u}}_n = q^{\mathrm{u}}_\wedge(n)$:

$$o(q^{\mathrm{d}}_\wedge(c)) = 1 + (o(q^{\mathrm{d}}_n) - 1) \cdot \frac{o(q^{\mathrm{u}}_n)}{1 + o(q^{\mathrm{u}}_n)} \cdot \frac{1 + o(q^u_c)}{o(q^u_c)}$$

or alternatively this can be written in terms of the message to the parent node with one child contribution removed using the ANTI-AND update B.4:

$$o(q^{\mathrm{d}}_{\wedge}(c)) = 1 + (o(q^{\mathrm{d}}_n) - 1) \cdot \frac{o(q^{\mathrm{u}}_{\wedge\backslash c}(n))}{1 + o(q^{\mathrm{u}}_{\wedge\backslash c}(n))} \tag{B.9}$$

OR

The downward message from the OR factor is given from 4.9 and B.1 as:

$$o(q^{\mathrm{d}}_{\vee}(c)) = \frac{o(q^{\mathrm{d}}_n)}{\frac{1-o(q^{\mathrm{d}}_n)}{\prod_{i\neq c}(1+q^{\mathrm{d}}_i)} + o(q^{\mathrm{d}}_n)}. \tag{B.10}$$

We also have from B.5 that

$$\prod_{i\in \mathrm{ch}(n)\backslash c} (o(q^u_i) + 1) = \frac{o(q^{\mathrm{u}}_n) + 1}{o(q^{\mathrm{u}}_c) + 1}$$

which when substituted into B.10 gives the incremental update in terms of the upward message from the OR factor:

$$o(q^{\mathrm{d}}_{\vee}(c)) = \frac{o(q^{\mathrm{d}}_n) \cdot (o(q^{\mathrm{u}}_n) + 1)}{1 + o(q^u_c) - o(q^{\mathrm{d}}_n) \cdot o(q^{\mathrm{u}}_c) + o(q^{\mathrm{d}}_n) \cdot o(q^{\mathrm{u}}_n)} \tag{B.11}$$

or alternatively we use B.7 to can write it in terms of the message to the parent node with the contribution from the child in question removed, $q^{\mathrm{u}}_{\vee\backslash c}$, which gives

$$o(q^{\mathrm{d}}_{\vee}(c)) = \frac{o(q^{\mathrm{d}}_n) \cdot \left(o(q^{\mathrm{u}}_{\vee\backslash c}(n)) + 1\right)}{o(q^{\mathrm{d}}_n).o(q^{\mathrm{u}}_{\vee\backslash c}(n)) + 1} \tag{B.12}$$

B.2 Log Odds Ratios

Now we derive the update equations for propagating messages up and down the search tree factor graph while storing the values of the parameters of the Bernoulli distributions in the log-odds domain: $L(o(q)) = \mathrm{logodds}(q) = \log\left(\frac{q}{1-q}\right)$.

Upward Messages

AND

Taking the natural logorithm of equations B.3 and B.4 gives us the incremental update of the messages from an AND factor to its parent variable in the log-odds domain:

$$L(q_{\wedge}) \quad = \quad L(q_y) - \log\left(1 + e^{-L(q_x)} + e^{L(q_y)-L(q_x)}\right) \tag{B.13}$$

which is applied such that $L(p_y) > L(p_x)$ and the inverse AND rule:

$$L(q_{\wedge\backslash y}) = L(q_{\wedge}) + \log\left(e^{-L(q_y)} + 1\right) - \log\left(1 - e^{L(q_{\wedge})-L(q_y)}\right). \tag{B.14}$$

OR

Similarly we take the natural logarithm of equations B.6 and B.7 to obtain the incremental update for the OR messages:

$$L(q_{\vee}) = L(q_x) + \log\left(1 + e^{L(q_y)} + e^{L(q_y)-L(q_x)}\right) \tag{B.15}$$

which is applied such that $L(q_y) > L(q_x)$ and the inverse OR rule:

$$L(q_{\vee\backslash y}) = \log\left(e^{L(q_{\vee})-L(q_y)} - 1\right) - \log\left(e^{-L(q_y)} + 1\right). \tag{B.16}$$

Downward Messages

AND

The incremental update for the downward messages from an AND factor in the log-odds domain is written in terms of the message to the parent node with the contribution from the child we are sending the message down to removed, $q_{\wedge\backslash c}$. The update is simply found by taking the natural logarithm of equation B.9 which gives:

$$L(q^{\mathrm{d}}_{\wedge}(c)) = L(q^{\mathrm{d}}_n) + L(q_{\wedge\backslash c}(n)) + \log\left(1 + e^{-L(q^{\mathrm{d}}_n)-L(q_{\wedge\backslash c}(n))}\right) - \log\left(1 + e^{L(q_{\wedge\backslash c}(n))}\right). \tag{B.17}$$

OR

The incremental update for the downward message from an OR factor is also written in terms of the message to the parent node with the contribution from the child in question removed, $q_{\vee\backslash c}$, by taking the natural logarithm of B.12:

$$L(q^{\mathrm{d}}_{\vee}(c)) = L(q^{\mathrm{d}}_n) + L(q_{\vee\backslash c}(n)) + \log\left(1 + e^{-L(q_{\vee\backslash c}(n))}\right) - \log\left(1 + e^{L(q^{\mathrm{d}}_n)+L(q_{\vee\backslash c}(n))}\right). \tag{B.18}$$

Bibliography

V. L. Allis. *Searching For Solutions in Games and Artificial Intelligence.* PhD thesis, University of Limburg, 1994.

L.B. Almeida, T. Langlois, J. D. Amaral, and A. Plankhov. Parameter adaptation in stochastic optimization. *On-line Learning in Neural Networks*, pages 111–134, 1998.

E. B. Baum and W. D. Smith. A Bayesian approach to relevance in game playing. *Artificial Intelligence*, 97(1-2):195–242, 1997.

T. Bayes. An essay towards solving a problem in the doctrine of chances. *Philosophical Transactions of the Royal Society*, 53:370–418, 1763.

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

E. Berlekamp and D. Wolfe. *Mathematical Go: Chilling Gets the Last Point.* Wellesley, 1994.

H. J. Berliner. The B* tree search algorithm: A best first proof procedure. *Artificial Intelligence*, 12, 1979.

C. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

M. Boon. A pattern matcher for Goliath. *Computer Go*, (13):12–23, 1990.

B. Bouzy and T. Cazenave. Computer Go: An AI oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

B. Bouzy and G. Chaslot. Bayesian generation and integration of K-nearest-neighbor patterns for 19x19 Go. In Graham Kendall and Simon Lucas, editors, *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games*, pages 176–181, 2005.

B. Bouzy. Go patterns generated by retrograde analysis. In *Proceedings of the Computer Olympiad Workshop in Maastricht*, 2001.

B. Bouzy. Mathematical morphology applied to computer Go. *International Journal of Pattern Recognition and Artificial Intelligence*, 17(2):257–268, 2003.

B. Brügmann. Monte Carlo Go, 1993. ftp://ftp.cgl.ucsf.edu/pub/pett/go/ladder/mcgo.ps.

T. Cazenave. Automatic acquisition of tactical Go rules. In *Proceedings of the Game Programming Workshop in Japan'96*, 1996.

T. Cazenave. Generation of patterns with external conditions for the game of Go. In H. J. van den Herik and B. Monien, editors, *Advances of Computer Games 9*, 2001.

P. C. Chi and D. S Nau. Comparison of the minimax and product back-up rules in a variety of games. *Search in Artificial Intelligence*, pages 450–471, 1988.

S. Cohen and Y. Matias. Spectral Bloom filters. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 241–252, New York, NY, USA, 2003. ACM Press.

R. Coulom. Computing Elo ratings of move patterns in the game of Go. In *Computer Games Workshop 2007*, Amsterdam/ The Netherlands, 2007. http://hal.inria.fr/inria-00149859/en/.

R. T. Cox. Probability, frequency, and reasonable expectation. *American Journal of Physics*, 14, 1946.

F. G. Cozman. Generalizing variable elimination in Bayesian networks. In *Proceedings of the IBERAMIA/SBIA 2000 Workshops*, pages 27–32, 2000.

F. A. Dahl. Honte, a Go-playing program using neural nets. In *Machines that Learn to Play Games.* Nova Science Publishers, Inc, 1999. http://citeseer.ist.psu.edu/dahl99honte.html.

J. Davies. *Tesuji.* Kiseido Publishing Company, 1975.

J. Davies. The rules of Go. In Richard Bozulich, editor, *The Go Player's Almanac.* Ishi Press, 1992.

F. de Groot. Moyogo studio, 2005. http://www.moyogo.com.

R. G. Edwards and A. D. Sokal. Generalisation of the Fortuin-Kasteleyn-Swendsen-Wang representation and Monte Carlo algorithm. *Physical Review Letters*, 38(6), 1988.

M. Enzenberger. The integration of a priori knowledge into a Go playing neural network, 1996. http://www.markus-enzenberger.de/neurogo.html.

M. Enzenberger. Evaluation in Go by a neural network using soft segmentation. In *10th Advances in Computer Games Conference*, pages 97–108, 2003.

J. Fairbairn. Go in ancient china., 1995. http://gobase.org/reading/history/china/.

D. Fotland. Knowledge representation in The Many Faces of Go. URL: ftp://www.joy.ne.jp/welcome/igs/Go/computer/mfg.tex.Z, 1993.

M. P. Frank. Advances in decision theoretic AI: Limited rationality and abstract search. Master's thesis, Massachusetts Institute of Technology, 1994.

S. Gelly and D. Silver. Combining online and offline knowledge in uct. In *ICML '07: Proceedings of the 24th International Conference on Machine learning*, pages 273–280, New York, NY, USA, 2007. ACM Press.

S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with patterns in Monte-Carlo go. Technical Report 6062, INRIA, France, November 2006.

T. Graepel, M. Goutrie, M. Kruger, and R. Herbrich. Learning on graphs in the game of Go. In *Proceedings of the International Conference on Artificial Neural Networks, ICANN 2001*, 2001.

R. Herbrich, T. Minka, and T. Graepel. TrueSkill: A Bayesian skill rating system. In *Advances in Neural Information Processing Systems 19*, pages 569–576, Cambridge, MA, 2007. MIT Press.

T. Heskes. Stable fixed points of loopy belief propagation are minima of the Bethe free energy. In *Advances in Neural Information Processing Systems 15*, 2003. citeseer.ist.psu.edu/heskes02stable.html.

G. E. Hinton, S. Osindero, and Y. W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18, 2006.

G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14:1771–1800, 2002.

A. Hollosi and M. Pahle. Sensei's library, 2007. http://senseis.xmp.net.

E. T. Jaynes. *Probability Theory: The Logic of Science.* Cambridge University Press, 2003.

S. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics Speech and Signal Processing*, 35(3), 1997.

D. E. Knuth and R. N. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

L. Kocsis and C. Szepesvari. Bandit based Monte-Carlo planning. In *Lecture Notes in Artificial Intelligence*, volume 4212, pages 282–293. 2006.

F. R. Kschischang, B. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans. Inform. Theory*, 47(2):498–519, 2001.

J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML '01: Proceedings of the 18th International Conference on Machine Learning*, 2001.

Y. Lecun, B. Boser, J. S. Denker, D., R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, Winter 1989.

S. Leys. *The Analects of Confucius.* W. W. Norton and Company, 1997.

D. J. C. MacKay and L. Peto. A hierarchical Dirichlet language model. *Natural Language Engineering*, 1(3):1–19, 1994.

D. MacKay. Information-based objective functions for active data selection. *Neural Computation*, 4(4):590–604, 1992.

D. J. C. MacKay. *Information Theory, Inference and Learning Algorithms.* Cambridge University Press, 2003.

D. A. McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35:287–310, 1988.

N. Metropolis and S. Ulam. The monte carlo method. *Journal of the American Statistical Association*, 44(247):335–341, 1949.

G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63:81–97, 1956.

T. P. Minka. *A family of algorithms for approximate Bayesian inference.* PhD thesis, Massachusetts Institute of Technology, 2001.

T. P. Minka. Divergence measures and message passing. Technical Report MSR-TR-2005-173, Microsoft Research, 2005.

M. Muller. Decomposition search: A combinatorial games approach to game tree search, with application to solving Go endgames. In *IJCAI-99*, volume 1, pages 578–583, 1999.

M. Muller. Computer Go. *Artificial Intelligence*, 134:145–179, 2002.

I. Murray and Z. Ghahramani. Bayesian learning in undirected graphical models: Approximate MCMC algorithms. In *Proceedings of the 20th Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, 2004.

N. J. Nilsson. *Problem Solving in Artificial Intelligence.* McGraw-Hill, 1971.

A. J. Palay. *Searching with Probabilities.* Pitman Publishing Ltd, 1985.

J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Assison-Wesley, 1984.

J. Pearl. *Probabilistic Reasoning in Intelligent Systems.* Kaufmann, 1988.

A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Exploiting graph properties of game trees. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 234–239, Portland, OR, 1986.

R. L. Rivest. Game tree searching by min / max approximation. *Artificial Intelligence*, 34:77–96, 1988.

S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

S. J. Russell and E. Wefald. *Do the Right Thing: Studies in Limited Rationality.* The MIT Press, 1991.

R. R. Salakhutdinov, A. Mnih, and G. E. Hinton. Restricted boltzmann machines for collaborative filtering. In *ICML '07: Proceedings of the 24th International Conference on Machine learning*, New York, NY, USA, 2007. ACM Press.

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

S. Sanner, T. Graepel, R. Herbrich, and T. P. Minka. Learning crfs with hierarchical features: An application to Go. In *Proceedings of the Workshop on Constrained Optimization and Structured Output Spaces*, 2007.

N. Sasaki, Y. Sawada, and J. Yoshimura. A neural network program of Tsume-Go. In *Computers and Games: First International Conference, CG'98*, pages 167–, Tsukuba, Japan, 1999. Springer Berlin/Heidelberg.

J. Schaeffer and A. Plaat. Kasparov verses DEEP BLUE: The rematch. *ICCA Journal*, 20(2):95–101, 1997.

N. N. Schrauldolph, P. Dayan, and T. J. Sejnowski. Temporal difference learning of position evaluation in the game of go. In *Advances in Neural Information Processing Systems 6*, pages 817–824, San Fransisco, 1994. Morgan Kaufmann.

D. Silver, R. Sutton, and M. Muller. Reinforcement learning of local shape in the game of Go. In *20th International Joint Conference on Artificial Intelligence*, pages 1053–1058, 2007.

H. A. Simon. How big is a chunk? *Science*, 183:482–488, 1974.

H. A. Simon. *Models of Bounded Rationality, Volume 2.* MIT Press, 1982.

D. H. Stern, T. Graepel, and D.J.C. MacKay. Modelling uncertainty in the game of Go. In *Advances in Neural Information Processing Systems 16*, pages 33–40, 2004.

D. Stern, R. Herbrich, and T. Graepel. Bayesian pattern ranking for move prediction in the game of Go. In *ICML '06: Proceedings of the 23rd International Conference on Machine learning*, pages 873–880, New York, NY, USA, 2006. ACM Press.

D. Stern, R. Herbrich, and T. Graepel. Learning to solve game trees. In *ICML '07: Proceedings of the 24th International Conference on Machine learning*, pages 839–846, New York, NY, USA, 2007. ACM Press.

D. Stoutamire. Machine learning applied to Go. Master's thesis, Case Western Reserve University, 1991.

R.S. Sutton and A.G. Barto. *Reinforcement Learning.* The MIT Press, 1998.

R. H. Swendsen and J-S Wang. Nonuniversal critical dynamics in Monte Carlo simulations. *Physical Review Letters*, 58:86–88, 1987.

R. M. Tanner. A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, IT-27:533–547, 1981.

G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

T. Thomsen. Lambda-search in game trees - with application to go. *ICGA Journal*, 23(4):203–217, December 2000.

J. van der Steen. Gobase, 1994. http://www.gobase.org.

E. van der Werf, J. Uiterwijk, E. Postma, and J. van den Herik. Local move prediction in Go. In *3rd International Conference on Computers and Games*, Edmonton, 2002.

E. C. D. van der Werf, J. van den Herik, and J. W. H. M. Uiterwijk. Learning to estimate potential territory in the game of Go. In *Proceedings of the 4th International Conference on Computers and Games*, Edmonton, 2004.

E. van der Werf. *AI Techniques for the Game of Go.* PhD thesis, Universiteit Maastricht, 2005.

Y. Weiss. Belief propagation and revision in networks with loops. Technical report, AI Lab Memo, MIT, Cambridge, 1998.

J. M. Winn and C. Bishop. Variational message passing. *Journal of Machine Learning Research*, 6:661–694, 2005.

T. Wolf. The program GoTools and its computer-generated tsume Go database. In H. Matsubara, editor, *Game Programming Workshop in Japan '94*, pages 84–96, Tokyo, Japan, 1994. Computer Shogi Association.

T. Wolf. Forward pruning and other heuristic search techniques in tsume Go. *Information Sciences*, 122:59–76, 2000.

G. Wood. An unreasonable game. *Living Dolls: a Magical History of the Quest for Mechanical Life*, pages 55–104, 2002.

L. Wu and P. Baldi. A scalable machine learning approach in Go. In *Advances in Neural Information Processing Systems 19*, pages 1521–1528, Cambridge, MA, 2007. MIT Press.

J. S. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalisations. In G. Lakemeyer and B. Nebel, editors, *Exploring Artificial Intelligence in the New Millenium.* Morgan Kaufmann, 2002.

A. L. Zobrist. *Feature Extractions and Representations for Pattern Recognition and the Game of Go.* PhD thesis, Graduate School of the University of Wisconsin, 1970.

A. Zobrist. A new hashing method with applications for game playing. *ICCA Journal*, 13(2):69–73, 1990.