Temporal Difference Learning in Complex Domains

Smith, Martin C.

Temporal Difference Learning in Complex Domains

by

Martin C. Smith

Submitted to the University of London
for the Degree of
Doctor of Philosophy in Computer Science

QUEEN MARY
AND WESTFIELD COLLEGE
UNIVERSITY OF LONDON

October 1999

This work is dedicated to

Bethina

Acknowledgements

I would like to thank my wife Bethina and my parents Margaret and Barry for their love and understanding throughout my student years. I would also like to thank all at the QMW Department of Computer Science. Most importantly I would like to thank my supervisor and friend Don Beal, without whose help, support and encouragement this thesis would not have been possible.

ABSTRACT

This thesis adapts and improves on the methods of TD(λ) (Sutton 1988) that were successfully used for backgammon (Tesauro 1994) and applies them to other complex games that are less amenable to simple pattern-matching approaches. The games investigated are chess and shogi, both of which (unlike backgammon) require significant amounts of computational effort to be expended on search in order to achieve expert play. The improved methods are also tested in a non-game domain.

In the chess domain, the adapted TD(λ) method is shown to successfully learn the relative values of the pieces, and matches using these learnt piece values indicate that they perform at least as well as piece values widely quoted in elementary chess books. The adapted TD(λ) method is also shown to work well in shogi, considered by many researchers to be the next challenge for computer game-playing, and for which there is no standardised set of piece values.

An original method to automatically set and adjust the major control parameters used by TD(λ) is presented. The main performance advantage comes from the learning rate adjustment, which is based on a new concept called *temporal coherence*. Experiments in both chess and a random-walk domain show that the temporal coherence algorithm produces both faster learning and more stable values than both human-chosen parameters and an earlier method for learning rate adjustment.

The methods presented in this thesis allow programs to learn with as little input of external knowledge as possible, exploring the domain on their own rather than by being taught. Further experiments show that the method is capable of handling many hundreds of weights, and that it is not necessary to perform deep searches during the learning phase in order to learn effective weights

Declaration

The work presented in this thesis is my own and is the result of research and experiments carried out by myself, with the exception of those parts noted below that are the results of work done in conjunction with my supervisor.

The majority of the program code used for this research was written by myself, including all of the temporal difference engine. A few sections, such as the interface for the chess program and parts of the shogi capture search, were written in collaboration with my supervisor. The various published papers arising from the work presented in this thesis were written jointly with my supervisor. Appendix E is a lightly edited version of a co-authored paper resulting from research undertaken prior to my PhD.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

1.1 Aims and Objectives

Temporal Difference learning is a natural method of reinforcement learning applied to prediction sequences. Sutton (1988) introduced the TD(λ) method which is an elegant integration of supervised learning with TD learning and which enabled Tesauro's backgammon program (1992, 1994) to reach World Championship standard. The aim of the research presented in this thesis was to adapt and improve on the methods that were used successfully for backgammon and apply them to other complex games that are less amenable to simple pattern-matching approaches. The methods used were designed to be highly game-independent and so are potentially applicable to a wide range of two-player perfect information games. The games chosen for investigation are chess and shogi, both of which (unlike backgammon) require significant amounts of computational effort to be expended on search in order to achieve expert play. The improved methods were also tested in a non-game domain.

An important additional aim of the research was for the programs to learn about the chosen domains with as little input of external knowledge as possible. A primary objective was for the programs to learn to improve their playing performance by exploring the domain on their own, rather than by being taught. Hence the focus of this work is on learning from self-play, without access to any form of expert knowledge such as well-informed opponents or recordings of play between experts. This method is of greater potential value for problems where existing expertise is not available, or where the computer program may be able to go beyond the level of existing knowledge.

A further aim of the work presented here was to reduce the amount of computational effort that was involved in learning useful values for the test domains. This aim can be achieved by either reducing the computational cost of each training sequence, or by reducing the number of training sequences required. To reduce the cost of each training sequence (game) investigations were carried out into methods of enhancing the primary experimental platform to increase the efficiency of the searches it conducted. These experiments influenced the design of the search engines used in the learning experiments and resulted in platforms that were both more efficient and more robust.

Of equal importance to engine efficiency is the question of the number of training sequences required for effective learning. Tesauro's world championship standard backgammon program was trained on 1,500,000 games, but such large numbers of training games are not feasible in the more computationally demanding domains of chess and shogi. Research was conducted into methods of automatically setting and adjusting the major control parameters for the learning algorithm, thus reducing the numbers of training sequences required.

1.2 Thesis Structure

Chapter two considers the long and illustrious history of game-playing computer programs. Such programs have been a primary area of interest for research in artificial intelligence throughout much of the last fifty years. Recently, there has been renewed interest in games from the AI community, as witnessed by the First International Conference on Computers and Games (van den Herik and Iida 1999), and a session devoted to games playing programs at the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99). Machine learning techniques for game-playing programs date back to Samuel (1959), and have recently been attracting more interest. Chapter two also discusses the history of temporal difference learning methods, and in particular their successful application to backgammon.

Chapter three introduces temporal difference learning as a concept, and then describes Sutton's TD(λ) formalism. The methods used in this thesis to apply TD learning to complex game domains are described, as are the search techniques commonly used in such domains. We also discuss why this thesis concentrates on learning without access to expert knowledge, and why lookahead search is required in the chosen test domains.

The fourth Chapter describes the search platform that was used as the primary test bed for the experimental work The basic search engine is described, as are a number of sophisticated enhancements to the basic engine which have the capacity to greatly improve the efficacy of the search. The work on multiple probes of transposition tables resulted in a paper published in the *International Computer Chess Association (ICCA) Journal* (Beal and Smith 1996). Various selective search techniques are discussed, and one such set of methods, search extension heuristics, are considered in detail. A selection of the research described in the search extension section was published in the *ICCA Journal* (Beal and Smith 1995). The application of the methods described in this Chapter to other domains is discussed.

Chapter five describes extensive experiments using TD(λ) to learn the relative values of chess pieces. Precise details are given of how the temporal difference learning platform described in Chapter three was combined with the search engine described in Chapter four. Results from these learning experiments are presented, as are results from matches designed to verify that the values learnt compared well with hand-rafted values. The experiments include using various search depths and comparison is made with programs using little or no search. The research and experimental results presented in this Chapter led to the publication of two papers, the first published in the *ICCA Journal* (Beal and Smith 1997) and the second accepted for publication by *Information Sciences Journal* (Beal and Smith 1999c).

Chapter six takes the methods that were successfully applied to chess in the previous Chapters, and applies them instead to the more computationally demanding domain of Shogi. An introduction to the game of Shogi is provided, and the major differences between shogi and chess are described, and their implications for learning discussed. Results are presented from a number of experiments that apply TD methods to the learning of shogi piece values, and also to the learning of more sophisticated positional features. The research described in this Chapter formed the basis of two papers, one published in the conference proceedings of the *First International Conference on Computers and Games* (Beal and Smith 1998b), and the other accepted for future publication in the journal *Theoretical Computer Science* (Beal and Smith 1999a).

The seventh Chapter describes the application of TD methods to learning more complex weight sets. Results are presented from experiments that successfully learn weights for a variety of evaluation coefficients, and also for learning one of the internal parameters of the squashing function described in Chapter three. Some of the results presented in this Chapter were included in a paper accepted for publication by *Information Sciences Journal* (Beal and Smith 1999c).

Chapter eight is the most important Chapter of this thesis. It describes novel extensions of the temporal difference learning method which automatically set and subsequently adjust the two major control parameters used by TD(λ). The main performance advantage comes from the learning rate adjustment, which is based on a new concept we call temporal coherence. Experiments are described which compare the performance of the temporal coherence algorithm with human-chosen parameters and with an earlier method for learning rate adjustment. The application of these parameter-adjusting methods to other domains is discussed. The research and experimental results described in this Chapter led to a paper presented at the *Sixteenth*

International Joint Conference on Artificial Intelligence (IJCAI'99) (Beal and Smith 1999b).

The final Chapter presents the conclusions of this thesis, and discusses areas for possible future work. There then follows a list of references used in the thesis, and appendices containing experimental details. Appendix E is a lightly edited version of a co-authored paper (Beal and Smith 1994) resulting from research undertaken prior to my PhD which turned out to be relevant to issues arising in Chapters 3, 5 and 7.

2 GAME PLAYING PROGRAMS AND MACHINE LEARNING

2.1 Fifty Years of Computer Chess and Artificial Intelligence

Creating a machine with the necessary qualities required for playing chess (which requires discovering what these qualities are) has been a major interest of many of the major figures in the history of computing.

Charles Babbage (1792-1871), in his *Passages on the Life of a Philosopher* (1864), described the possibility of making his Analytical Engine play chess and formulated some simple rules (including lookahead) that such a chess playing automaton would be required to consider.

Alan Turing is often considered as the founding father of artificial intelligence. In his seminal paper, *Computing Machinery and Intelligence* (Turing, 1950) he introduced his "imitation game", which subsequently became known as simply the *Turing Test*. This paper mentions chess as one of the domains suitable for the comparison of human and machine thinking. Turing (1953) subsequently wrote a simple chess playing program called TUROCHAMP, and simulated by hand its simple one-ply search plus evaluation. Levy and Newborn (1991) present the moves of a game played by Turing's program.

Other famous names from the field of AI who produced early chess-playing programs include Donald Michie, John McCarthy, and Newell and Simon.

Much of the research conducted in the field of game-playing programs can be traced back to a milestone paper by Shannon (1950) in which he described two possible strategies for computer chess. The first, which Shannon called "type–A", was essentially a brute force method consisting of a fullwidth minimax search and static evaluation. Shannon himself pointed out that such a search would be hugely computationally expensive because all lines of play would be searched to the same depth, and in some positions long forcing sequences would need to be considered. Shannon's "type-B" strategy involved searching some lines more deeply than others (i.e. selective search) and also introduced the concept of *quiescence* (see Chapter 3).

After fifty years of progress and innovation, computer chess remains an active research area. Whilst the best programs are now as strong as the best humans, their

play is far from perfect. The enormous size and complexity of the chess search space ensures that it will remain unsolved for the foreseeable future.

Whilst there is still an undeniable link between artificial intelligence and computer chess (most introductory AI textbooks have a section on computer chess), modern high-performance programs use little by way of AI techniques. The success of minimax, and in particular alpha-beta pruning, means that most competitive performance programs rely more on what is sometimes referred to as Brute Force and Ignorance (BFI). Human expert knowledge is applied to the identification and weighting of suitable evaluation terms for use by a powerful search engine, and little success has been had by those who have attempted to model the thought processes of human masters.

Now that programs have reached the strength of Grandmasters, questions arise about the limitations of human expert knowledge. The ability for machines to learn about evaluation functions for themselves, in the context of their own search regimes, is one obvious next step forward.

2.2 The Complexity of Chess and the Challenge of Perfect Knowledge

Over the last 100 years, some chess commentators have suggested that Grandmaster play is approaching perfection. They cite the preponderance of draws of the top level of play, and wonder if the top players now understand the game so well that they will only lose by making gross mistakes. As long ago as 1928 World Champion Capablanca wrote:

> "Of late we have lost a great deal of the love for the game, because we consider it as coming to an end exceedingly fast. In effect, if one were satisfied to draw, we believe that it would not be impossible to draw all the games. To avoid drawing variations one might have to enter into inferior lines of play which might lead to disaster against a first-class opponent. At present there may not be more than one or two players in the world who might do that, but within ten years there will probably be three or four." (Reinfeld 1953, p. 160).

Some have considered increasing the complexity of the game by making the board larger and adding extra pieces (this was Capablanca's preferred solution). Others have suggested alterations to the rules or starting position (e.g. former World Champion Fischer, Pritchard 1994). In the aftermath of the 1997 match in which the

World Champion Gary Kasparov was defeated by the computer Deep Blue, it has been suggested that chess now holds less challenge for researchers (Pitrat 1998). Evidence against the propositions that chess play is approaching perfection and that chess is now less of a research challenge may be obtained from endgame analyses.

With the advent of endgame tablebases constructed by retrograde analysis (e.g. Thompson, 1987), perfect information about non-trivial chess positions became available. The play of grandmasters in such endings, once assumed to be near perfect, was shown not only to be sub-optimal, but in many cases to include mistakes that might have resulted in won endings being drawn, and draw endings being lost. Even for the relatively simple 4-man ending of king and queen versus king and rook, the widely accepted methods (e.g. Keres 1973) were found to be insufficient to assure a win for the stronger side against perfect defence. Indeed, Grandmaster Walter Browne was once famously unable to win this ending against a tablebase (Kopec 1990). Despite thorough study of the endgame he struggled in the re-match, requiring exactly 50-moves to win from a position that would require only 31 moves with perfect play. Even programs with specialised endgame heuristics (which compare well with strong human play) find it difficult to make progress against the tablebase (Walker 1996). Some of the more complicated 5-man endings, e.g. king, queen and pawn versus king and queen (Roycroft 1986), require optimal play of such a complex and seemingly method-less nature that they may well be beyond the bounds of human understanding (Michie 1990). Levy and Newborn (1991) provide an equally baffling 153 ply sequence of optimal play extracted from the king, bishop and knight versus king and knight tablebase.

The above is not meant to belittle the achievements of great human players, but rather to highlight the rich complexity of the game, and to suggest that chess is much further from being 'solved' (in the lay sense) than has often been suggested in the past. Indeed, the sheer complexity of chess may well have been grossly underestimated. The fact that mere 5-man endgames are now known to be so complex and troublesome for humans implies that 32-man endings such as the chess starting position will remain challenging for the foreseeable future.

2.3 Machine Learning in Games

2.3.1 Samuel's checkers player

The work of Arthur Samuel with his programs that learnt to play checkers (draughts) was years ahead of its time and was the earliest example of machine learning in games (Samuel 1959).

Samuel wrote a checkers-playing program for the IBM 701 in 1952, and his first learning program in 1955. He constructed a checkers program instead of a chess program because the relative simplicity of checkers made it easier to concentrate on the learning aspects of the program (Samuel 1959). Samuel's programs used search methods similar to those described by Shannon (1950) (see section 2.1), incorporating minimax search with a polynomial evaluation function.

There were two forms of learning used by Samuel's programs, the first of which he called *rote learning*. This consisted of storing every board position reached during play along with its associated value as determined by the minimax search. If a previously stored position was encountered during a search its stored value would be used as its evaluation. This effectively increased the program's search depth, as the stored value represented the result of one or more previous searches. The stored values were decremented by a small amount every time the score was backed up a ply as part of the minimax search. This ensured that the program chose the shortest path to a favourable position. The rote learning program was trained by a combination of self-play, play against various humans, and supervised learning from records of games between human experts. Samuel found that this method resulted in slow but significant learning, especially in the opening and endgame, and produced a program that performed like "a better-than-average novice" (Samuel 1959).

A second, more sophisticated, form of learning was used by Samuel (1959) to modify the parameters of the program's evaluation function. This was a temporal difference method (although Samuel himself did not use that term) which was a predecessor of the TD methods used by Tesauro in TD-Gammon (Tesauro 1992). Samuel's program learnt via self-play, and sought to minimise the difference between successive evaluations of positions that occurred in the course of these games.

A significant feature of Samuel's temporal difference method was that the learning did not make direct use of the actual results of games, as the known game-theoretic values of the game-terminal positions were not used. The adjustments made to the

weights made by the learning process were driven by reducing the difference between successive evaluations, but because game-terminal positions were not taken into account, this could be achieved by finding a set of weights that ensured all positions were evaluated identically. Whilst such a set of weights would satisfy the learning process, it would be useless for producing improved play. Samuel was aware of this potential problem, and to alleviate it fixed the weight of the most significant evaluation feature, *piece advantage*. This measured the number of pieces the program had relative to its opponent, giving higher value to kings, and is a powerful heuristic in checkers which has a high correlation with winning the game. The fixing of this weight to a large positive constant meant that the program's play was determined by seeking to maximise the value of this feature. As Sutton (1997) points out, it would still have been possible for the program to learn a useless set of evaluation weights by setting the adjustable weights so that they always cancelled out the *piece advantage* term.

Samuel's program had a set of 38 carefully chosen weighted evaluation terms, (e.g. centre control), of which a subset of 16 where used at any one time. In the course of learning, features which were found not to lead to consistency of evaluation were replaced by others drawn from the "reserve pool". At times during the learning process when the program was not improving, Samuel found it useful to reset the largest weight to zero. His justification was that this prevented the weight set from settling into locally optimal values. Given that learning was driven only by producing consistent evaluations, with no regard for the outcomes of the games, it is also possible that this drastic method helped to prevent the program from adopting weight sets that produced consistent evaluations that did not correlate with winning the game (Sutton 1997).

Samuel's program learnt well enough to beat its creator regularly, but its playing strength did not approach the strength of human masters. Nevertheless, Samuel's achievement in producing a successful learning program is one of the notable early successes in the fields of both machine learning and game-playing programs.

2.3.2 Neurogammon

Neurogammon was written by Gerald Tesauro (1989) and was a backgammon-playing program that was the direct predecessor to Tesauro's TD-Gammon program. Neurogammon was a multi-layered neural network whose input representation included both the raw board position and a set of carefully chosen evaluation features that utilised the knowledge of expert human players. It learnt via supervised learning,

training on a set of expert game records and adjusting its weights using the back-propagation algorithm (Rumelhart, Hinton and Williams 1986). Neurogammon reached the strength of a "strong-intermediate" human player, and decisively won the backgammon tournament at the 1989 International Computer Olympiad (Tesauro 1989).

2.3.3 TD-Gammon

Tesauro's TD-Gammon (Tesauro 1992,1994,1995) was a notably successful application of machine learning techniques. Learning via self-play, it utilised little domain specific knowledge, yet learned to play backgammon at a level close to that of the world's best human players.

TD-Gammon combined Sutton's TD(λ) algorithm (Sutton 1988, see Chapter 3) with a three-layer neural network consisting of a layer of input units, a layer of hidden units, and an output unit. The input to the network was a representation of a backgammon position, and the output was an estimate of the probability of winning from that position.

The initial version of the program, TD-Gammon 0.0, used an input representation that contained only the raw board information and did not incorporate any specially-crafted evaluation features. The program learnt only by playing against itself, and after about 200,000 self-play games its performance was approximately equivalent to that of Neurogammon. This was a remarkable achievement given that Neurogammon had required substantial amounts of domain-specific knowledge, both in terms of its carefully constructed evaluation features and the expert-level games it trained on.

Subsequent versions of TD-Gammon used the same learning methods, but the input representation included the same set of hand-crafted evaluation features used by Neurogammon. TD-Gammon 1.0 used 80 hidden weights instead of the 40 used by the earlier version, and after 300,000 self-play training games was significantly better than both version 0.0 and Neurogammon.

Version 2.1 incorporated a simple two-ply selective search mechanism. This was not part of the learning process but was used for move selection during play. After 1,500,00 self-play training games TD-Gammon 2.1 had reached a level of play comparable to that of the best human players (Tesauro 1995).

The success of TD-Gammon has had a noticeable effect on the play of the world's top backgammon experts, causing them to revise their thinking about certain types of

position and even to amend the way they play certain opening rolls, the equivalent of 'book' chess openings (Tesauro 1995).

3 TEMPORAL DIFFERENCE LEARNING AND MINIMAX SEARCH

3.1 Temporal Difference Learning

Temporal difference (TD) learning methods are a class of incremental learning procedures for learning outcome estimates in multi-step prediction problems. Each prediction is a single number, derived from a formula using adjustable weights, for which the derivatives with respect to changes in weights are computable. Whereas earlier prediction learning procedures were driven by the difference between the predicted and actual outcome, TD methods are driven by the difference between temporally successive predictions (Sutton, 1988). Each pair of temporally successive predictions gives rise to a recommendation for weight changes. Kaelbling *et al.* (1996) give a survey of a wider range of reinforcement algorithms, including TD methods.

Sutton's TD(λ) algorithm is based on the following formalism. Let $P_1 \ldots P_t$ be a set of temporally successive predictions, indexed from time *1* to time *t*. The algorithm assumes each prediction is a function of a vector of adjustable weights *W*, so a prediction at time *i* could be written as $P_i(W)$. The algorithm further assumes that the prediction function is differentiable so there exist partial derivatives of the prediction value with respect to each weight element. $\nabla_w P_i$ denotes the gradient, or vector of partial derivatives of prediction *P* at time *i*, with respect to weight *w*.

Using this notation, the weight adjustments for Sutton's TD(λ) algorithm can be expressed as:

$$\Delta w_t = \alpha(P_{t+1} - P_t)\sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k \qquad (3.1)$$

where α is a parameter controlling the learning rate, and λ is Sutton's recency parameter, that introduces an exponential weighting with recency of predictions occurring *k* steps in the past.

The process can be applied to any initial set of weights. Learning performance depends on λ and α, which have to be chosen appropriately for the domain.

TD(λ) learning enabled Tesauro's backgammon program to reach master level (Tesauro 1994). The methods used by Tesauro are discussed later in this Chapter.

TD(λ) has some problems however. The learning rate parameter α is hard to get right. It needs to be as high as possible for rapid learning, but high rates lead to high levels of erratic movements, even after optimum values might have been reached. In effect, high learning rates lead to high levels of noise in the weight movements, and this means that the process does not produce stable values.

On the other hand, learning rates that are too low can lead to orders of magnitude more observations being required to reach optimum weight values. Practical experience with the TD(λ) method indicates that very different values of α are required in different domains, as shown by the different rates used in, for example, Sutton (1988, 1992).

An original method for determining both α and λ is presented in Chapter 8.

3.2 Comparison with other Learning Methods

Prediction-outcome learning methods are driven by the difference between prediction-outcome pairs. For example, one might make a prediction after every move and compare this prediction with the actual outcome of the game. The resulting error term can then be used to make adjustments to the prediction. Obviously this method can only be applied once the result of the game is known.

In contrast, the TD method is driven by an error term generated by the comparison of successive predictions, and need not wait for the actual outcome of the game. Sutton (1988) shows that TD methods make more efficient use of their experience than conventional prediction-learning methods. They converge faster and produce more accurate predictions. In addition Sutton shows that TD methods are easier to compute because they are incremental and do not require a final outcome.

3.3 TD(λ) and Games

Perhaps the most successful application of Sutton's TD(λ) method was Tesauro's backgammon program, TD-Gammon, which used TD(λ) to train a three-layer neural network (see section 2.3).

Schraudolph, Dayan and Sejnowski (1994) used TD(λ) to train a neural-network to play Go on a small 9x9 board via randomised self-play. This met with only limited success, producing a program that was no stronger than a weak human beginner and

requiring 659,000 training games to reach this standard. The use of carefully designed network architectures, and knowledge-intensive training strategies (such as playing training games against a top commercial go program) allowed a subsequent program to achieve a slightly better standard of play after 3,000 training games. The program was never scaled up from 9x9 Go to a full-size 19x19 board, and the far greater complexity of the 19x19 game suggests that performance on a conventional board would be well below that achieved on the much simpler 9x9 board.

3.4 Minimax Search and Alpha-Beta Pruning

All two-player, finite, zero-sum, deterministic, perfect-information games can in theory be solved by application of the minimax algorithm. In order to solve such games, minimax requires that all game-terminal positions that might be reached from the starting position be considered, and their known game-theoretic value (e.g. in chess one of *win*, *loss* or *draw*) be passed back towards the starting position, assigning a value to all intermediate positions en-route. Once a game-theoretic value has been assigned to the starting position, the game is solved. The outcome of the game with perfect play for both sides is known, as is the sequence of moves required to achieve that value. In the case of complex games the combination of the game's branching factor and the distance from the starting position of the game-terminal positions results in a combinatorial explosion that makes such an approach to solving these games infeasible.

Given that the game-terminal positions are too far from the root of the search to be examined exhaustively, it is possible to search only a fraction of the overall game-space. The same minimax method of 'backing-up' values from distant positions can used be used in conjunction with an heuristic evaluation of positions at any depth in the search tree, allowing for the search of arbitrarily sized subtrees, and providing an heuristically chosen 'best' move. The resulting minimax searches have been found to be remarkably successful in practice, even when the evaluation used is a crude estimate. Beal and Smith (1994) demonstrate that minimax search produces better than random play even when the evaluation function is replaced by a random number generator (see Appendix E). Minimax search forms the basis of many game-playing programs, including those discussed in this thesis. [Strictly speaking, minimax search is usually implemented as 'negamax' (Knuth and Moore 1975), which passes back the negative of the subtree value. This simplifies the program structure as the search engine is always trying to maximise this value.]

An examination of the minimax algorithm reveals that there are some sections of the search tree that need not be considered because they have no effect on the final outcome of the search. Alpha-beta pruning removes from the search tree those lines of play that are not relevant to the evaluation of the root position. Figure 3.1 gives an example of a minimax search tree, and shows that in this example, alpha-beta pruning reduces the number of positions for which an evaluation is required from 8 to 7, with node *L* being cut off.

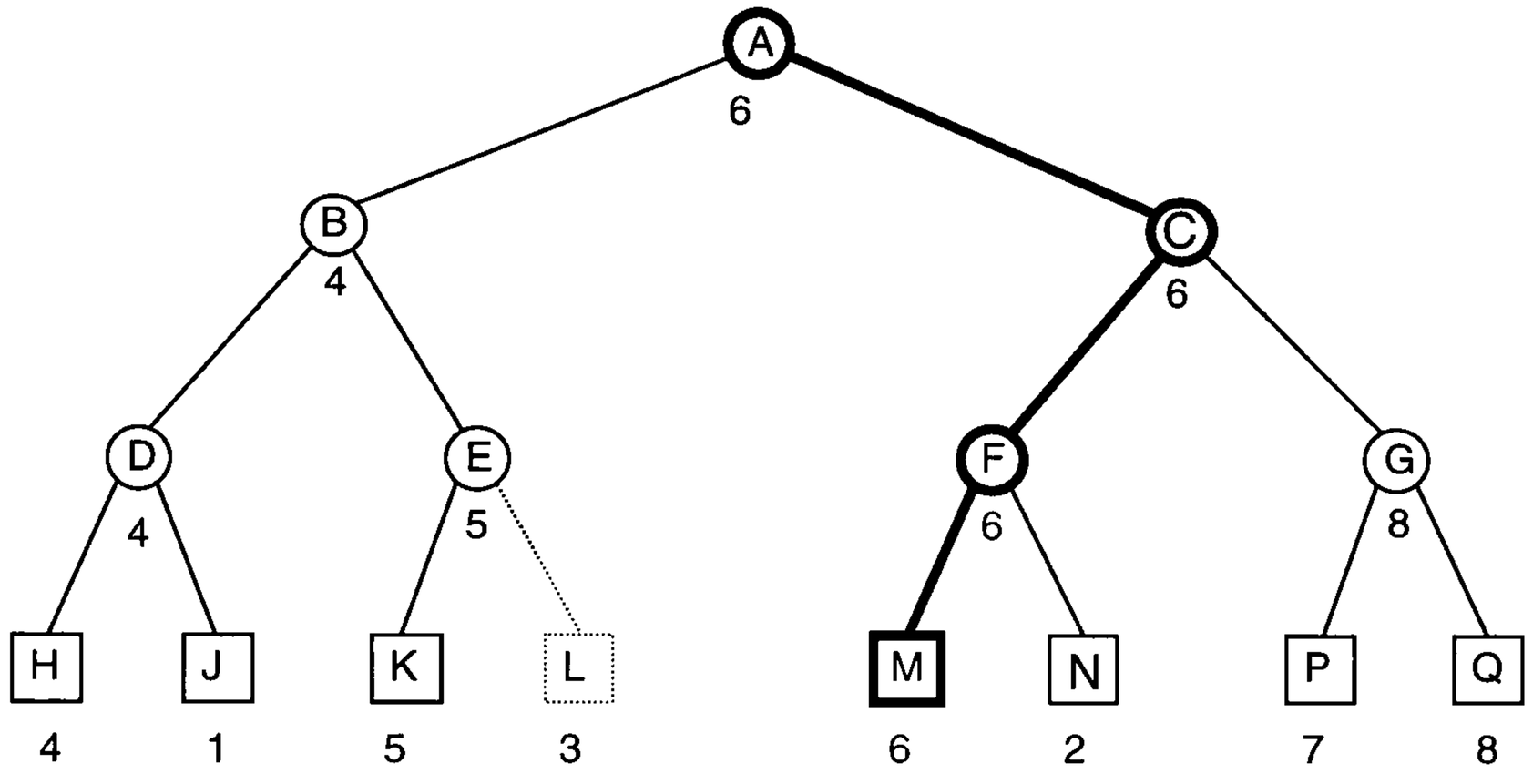

Figure 3.1: A minimax search tree incorporating alpha-beta pruning.

The most important method of reducing the size of a search using alpha-beta pruning is to improve the ordering so that stronger moves are tried before weaker moves, thus maximising the number of cut-offs that the alpha-beta mechanism is able to make.

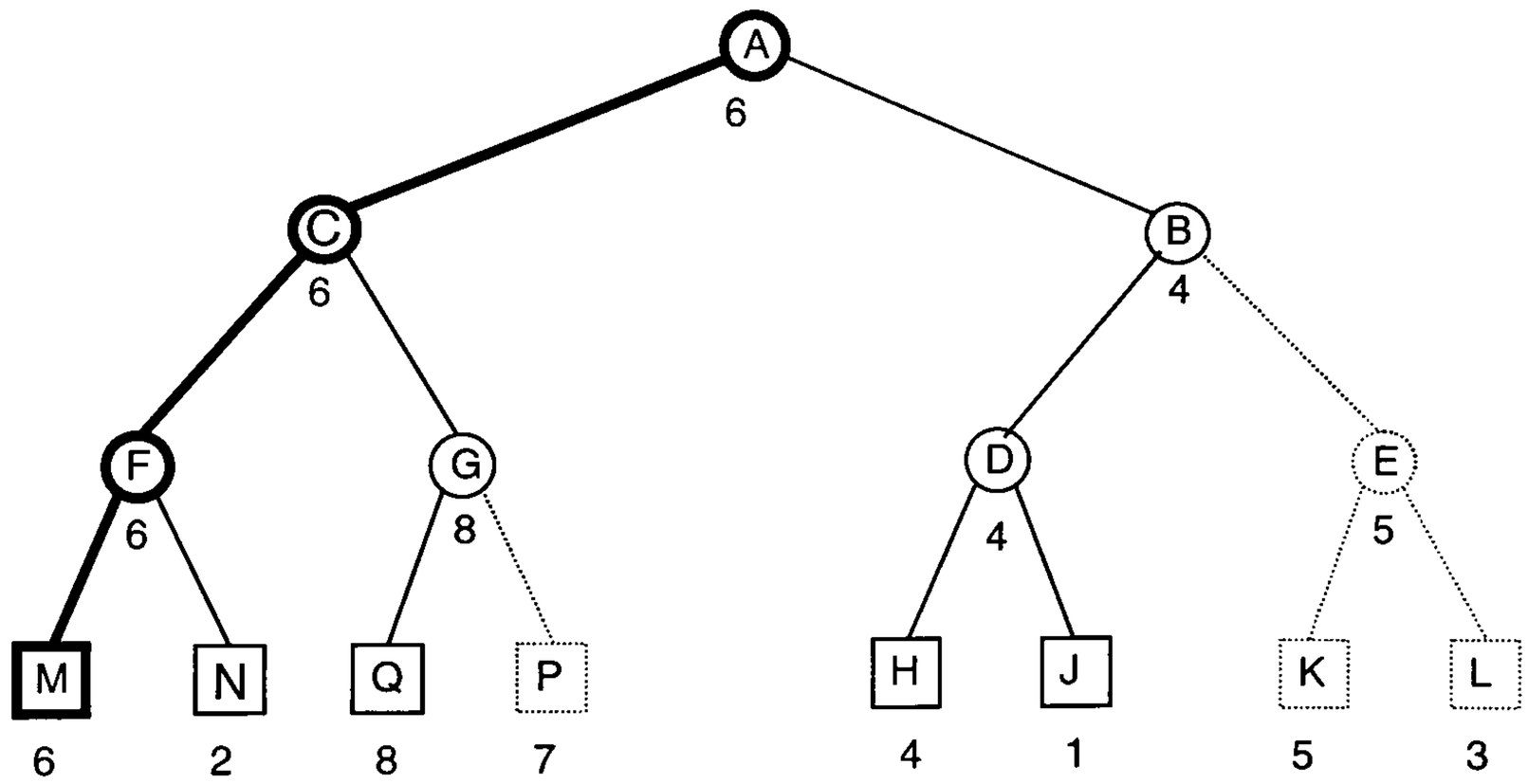

Figure 3.2: The same search tree as Figure 3.1, but with perfect move ordering.

Figure 3.2 shows essentially the same search tree as Figure 3.1, but with perfect move ordering. In this example the number of evaluations required is reduced to 5, with node *P* being cut off as well as the entire sub-tree of *E-K-L*.

Most chess playing programs achieve good move ordering by using iterative deepening in conjunction with transposition tables (see section 4.1). Other methods for selecting likely good moves include the killer heuristic (Frey 1977) and history tables (Schaeffer 1983). Typically the remaining moves are sorted so that captures are tried before non-captures (Slate and Atkin 1977).

In addition to providing information about the move that was found to be best on previous iterations of the search, the transposition table allows nodes that have previously been evaluated at the current depth of search to have those values recalled so that their subtrees need not be searched again. Unlike the move-ordering benefits of transposition tables, in the chess domain this usually provides only a modest increase in efficiency, except in some endgame positions where the number of possible transpositions (the same position reached via different move sequences) can increase dramatically.

3.4.1 The principal variation

The effect of the full-width minimax search algorithm is to select a sequence of moves that represents best play by both sides (as defined by the evaluation function). This line of play is referred to as the *principal variation*. The evaluation score from the position at the end of the principal variation (the *principal position*) is 'backed up' to the root of the search. Thus from a given position, a minimax search returns a value that corresponds to the evaluation of the position at the end of the principal variation.

3.4.2 Other search regimes

Over the years very many modifications have been made to the conventional minimax plus alpha-beta search regime applied to chess. Examples of the more successful enhancements are MTD (Plaat *et al.* 1994) and the popular Principal Variation Search (Marsland 1983). A number of best-first search methods have been proposed, including SSS* (Stockman 1979) and Berliner's famous B* algorithm (1979). A discussion of selective search is included in Chapter 4.

The experiments reported in this thesis were conducted using the basic iteratively-deepened depth-first, minimax plus alpha-beta algorithm, in an attempt to keep the method as game-independent as possible. Most of the methods presented here could be easily adapted to fit other, more specialised, search regimes.

3.5 Applying TD Learning to Complex Game Domains

In order to use temporal difference methods, the evaluation score for the position selected by each move is regarded as a prediction of the final outcome of the game. To be more precise, it is the evaluation score from the position at the end of the principal variation (the *principal position*) which is 'backed up' to the root of the search, and used as a prediction to be compared with future values.

3.5.1 Determining weights for evaluation terms

Central to all heuristic search, whether single-agent or adversarial, is an evaluation function. This estimates the distance to goal in single-agent searches, or the value of a game-state in competitive (zero-sum) tasks. The function is typically a polynomial formed from linearly weighted evaluation terms:

$$v(x) = \sum_{i=1}^{n} w_i c_i \qquad (3.2)$$

where c is a vector of n evaluation terms computed for the state x.

Setting the weights for evaluation terms has always been a significant problem, although often overshadowed by other concerns.

Game playing programmers have tried many schemes for automatically determining appropriate weights. Fürnkrantz (1996) gives a good summary of learning in game playing, including weight-learning algorithms.

Recently, one of the IBM programmers of Deep Blue, the program/hardware combination that defeated the human World Champion in a match in 1997 published a method to optimise linear discriminants for master-level moves (Anantharaman 1997). His method utilises the move-choice decisions of master players, aiming to obtain similar decisions. The learning optimises a linear combination of *Pscore* and *Mscore*, where *Mscore* is the number of move choices that agree, and *Pscore* the number of positions which the discriminant and chess master agree are worse than the move chosen. The end result is that the method determines evaluation weights for Deep Blue's evaluation function.

However, methods that do not require existing human expertise are required in unfamiliar domains, or if the method is aimed at levels of expertise which exceed human abilities.

To apply TD(λ) to game-playing there are some important requirements of the prediction function that need to be considered. The first of the requirements is that the function for predicting the game outcome should integrate smoothly with end-of-game values. This implies that the value for checkmate should be close to that for being heavily ahead in the summation of evaluation terms. Moreover, an additional pawn advantage when heavily ahead in material should have little effect on the prediction, whereas the gain of a pawn in an otherwise level position should have a relatively large effect on the prediction.

Another related requirement is that the prediction values should approximate the utility of the game result. For games such as chess and shogi, the utility is 1 for any win, 0 for any loss. (This contrasts with games such as bridge and backgammon, where winning a game brings a variable amount of reward.)

The foregoing requirements can be met by using a squashing function, which converts from the conventional polynomial evaluation function typically used in game-playing programs, to a probability of winning. It is convenient to use a standard sigmoid squashing function. The prediction P, the probability of winning from a given position x, is determined by using the function:

$$P(x) = S(v(x)) \tag{3.3}$$

where v is the evaluation value of position x. In typical games programs the evaluation function is often a polynomial (3.2).

S is defined by:

$$S(v) = \frac{1}{1+e^{-v}} \tag{3.4}$$

The sigmoid function (3.4) has the advantage that it has a simple derivative:

$$\frac{dS}{dv} = S(1-S) \tag{3.5}$$

Therefore the partial derivative of the prediction with respect to an individual weight w_i is:

$$\frac{\partial S}{\partial w_i} = \frac{\partial S}{\partial v}\frac{\partial v}{\partial w_i} = S(1-S)c_i \tag{3.6}$$

The derivative of S appears in classical supervised-learning procedures as well as TD(λ). The effect of the derivative in the weight adjustment formula is that weights receive adjustment in proportion to their effect on the prediction. On other words, weights that have little influence on the prediction are adjusted less than weights to which the prediction is more sensitive.

The function S can be elaborated to include an additional parameter to adjust the 'steepness' of the sigmoid. Section 7.6 describes some experiments to determine the usefulness of this modification.

3.5.2 Using the principal position, rather than the game position

An important aspect of applying TD(λ) to minimax search is selecting the correct position to use in the computation of weight adjustments. When performing minimax search to make move choices, the evaluation score from the position at the end of the principal variation (the *principal position*) is backed up to the root of the search.

Let $g_1, g_2, g_3 \ldots g_n$ be the positions of game G.

Let $h_1, h_2, h_3 \ldots h_n$ be the principal positions identified by the minimax searches from $g_1, g_2, g_3 \cdots g_n$.

Thus

$$P_i = S(v(h_i)) \tag{3.7}$$

At the end of the game, the outcome is defined by the rules of the game, thus $g_n = h_n = \{0 \mid 0.5 \mid 1\}$ where $\{0 \mid 0.5 \mid 1\}$ means one of the values 0, 0.5, or 1.

It is the value from the principal position that is the prediction of the final outcome of the game, to be compared with future values by the temporal difference method. Consequently, the computation of partial derivatives must be performed with evaluation terms calculated at the principal position h_i , not at the position in the game g_i.

There is a minor technical issue that arises here. In some positions, the principal position may be selected from an equivalence class of positions with equal scores, due to the existence of more than one move with the best score at some nodes along the principal variation. To be precise, in such cases the principal variation is selected from a principal tree. The selection will be determined by the move order at those nodes with multiple best moves. In such cases, the partial derivative at P_i may be

multi-valued. It is possible to define an algorithm that either disregards these cases, or examines all the principal positions of the equivalence class and computes the sum of all the partial derivatives. However, the simpler approach of accepting an arbitrary choice of principal position in these cases works well in practice.

3.6 Learning in the Absence of Expert Knowledge

A major aim of the research presented in this thesis was for the learning to occur with as little input of expert domain-specific knowledge as possible. The primary consequence of this is that all our training takes the form of self-play games, where the learning program plays against an identical version of itself. We chose not to aid the learning by playing training games against well-informed opponents, nor by replaying recordings of games between experts. The objective was for the programs to learn to improve their playing performance by exploring the domain on their own, rather than by being taught. In addition, all of the learnt weights were learnt 'from scratch' without any domain-specific knowledge being represented in their initial values. These knowledge-free methods are of greater potential value for problems where existing expertise is not available, or where the computer program may be able to go beyond the level of existing knowledge.

3.6.1 Self-play versus online play

The method of applying TD(λ) to minimax searches described in Chapter 3 was first published (Beal and Smith 1997) in the ICCA Journal. The same method was later reported to be successful in improving weights for a complex chess evaluation function consisting of positional terms as well as piece values in the program *KnightCap* (Baxter, Trigell and Weaver 1998) that was trained by playing games on the Free Internet Chess Server (FICS, `fics.onenet.net`) against human opponents. The program's blitz rating rose from Elo 1650 to 2150 after 308 games in one experiment, and after approximately 900 games in a second experiment. The rise of 500 rating points is good, but the final rating of 2150 for *blitz* chess (at which computers traditionally excel) is well below that achieved by conventional chess-playing programs.

Baxter *et al.* made use of online play against expert human opponents, whereas our experiments focus of learning in the absence of any domain specific knowledge and so training occurs via self-play. Training games against human experts require the availability and co-operation of both the experts and the chess server operators,

whereas self-play is completely self-contained and requires no human input whatsoever. Baxter *et al.* (1998) report that they found self-play to be ineffective for learning when compared with on-line play against human opponents. However, they used only a purely deterministic move choice for their self-play games, and so found that training produced a large number of substantially similar games (Baxter *et al.* 1997). The little variation that occurred was accounted for by changes in the evaluation weights between games. This method of self-play is clearly inferior to the randomised move choice used in the experiments reported in this thesis, and so the conclusion of Baxter *et al* that self-play is insufficient for adequate learning must be called into question. Indeed, our TD minimax methods in conjunction with self-play have been used successfully in the world-class competitive chess program *Cilkchess* (see section 7.5).

The quality of the games produced by play against human experts is one reason why online play produces faster (in terms of number of games played) learning than self-play. Baxter *et al.* note that an important feature of play against different human opponents is that their program is forced into positions that it evaluates highly but subsequently discovers are losing. These are precisely the types of position that learning programs need to see to learn rapidly and they occur more frequently when the program is being beaten by human experts than in the course of randomised self-play. Thus online play is *learning by being taught* as opposed to *learning by exploration* which is the focus of this thesis. The methods of this thesis are designed to be as domain-independent as possible, and so suitable for application to games (e.g. Shogi) for which large numbers of online human 'teachers' may not be available. There is also the theoretical question of how learning is to proceed once the program's level of play exceeds that of the best human players. For the domain of chess such a problem may well become a reality in the foreseeable future.

Baxter *et al.* comment that they found it necessary to provide piece weights as initial knowledge in order to obtain good performance. The performance of a program that was not seeded with carefully chosen 'intelligent' material weights was greatly inferior, achieving a rating of only Elo 1300 after 1000 games. In contrast, the experiments presented in this thesis achieved excellent performance gains without any knowledge being represented in the starting weights.

Baxter *et al.* also observe that another reason for the rapid rate of improvement of their program was that all the non-material weights were initialised to zero, rather than small random values. The means that "small changes in these parameters could cause very large changes in the relative ordering of materially equal positions. Hence

even after a few games [the program] was playing a substantially better game of chess." (Baxter, Tridgell and Weaver 1998 pg. 93). Experiments with random evaluations (Beal and Smith 1995) showed that even a set of randomly chosen positional weights would perform better than a set of all-zero weights (see Appendix E). Thus Baxter *et al.* were starting from a low-performance initial state, with easy initial improvements possible.

3.6.2 Why is search needed in some domains, but not others?

In some games domains the use of search algorithms is not required to produce expert level play. In backgammon, pattern matching techniques have proved themselves capable of producing extremely strong programs (see Chapter 2).

In domains such as chess and shogi the tactical complexity of the games makes a successful program that did not use search in some way inconceivable. Chess programs have very successfully used full-width minimax search with alpha-beta pruning (see section 4.1), and top commercial shogi programs rely on deep searches to avoid costly tactical errors and to detect mating sequences (Rollason 1999).

Other game domains have features that make the use of search, especially full-width search, impractical. The most notable of these is Go, where the high branching factor makes full-width search prohibitively expensive, and the lack of an obvious core evaluation term (such as piece count in chess) makes static evaluation of positions problematic. Nevertheless, search is used extensively in Go for calculating local tactical sequences, and the top commercial Go programs are increasingly turning towards selective search techniques for calculating global board sequences (Reiss 1999).

3.6.3 Learning in deterministic and non-deterministic games

Randomisation of the move selection during self-play in deterministic games such as chess and shogi is necessary to ensure that as large a section of the state space as possible is explored during learning. Such randomisation is not required in backgammon, because the stochastic nature of the die rolls naturally result in a large amount of variability in the positions reached during training games.

Learning in non-deterministic games such as backgammon is also made easier because the true expected outcome of a position given perfect play by both sides is a real-valued function with a great deal of smoothness and continuity (Tesauro 1995).

This means that small changes in the position produce small changes in the probability of winning. In deterministic games such as chess and shogi the outcome given perfect play (the *game-theoretic value*) is discrete (win, lose, draw) and therefore much more likely to be discontinuous and lack smoothness, with the result that such a function is much harder to learn.

4 A PLATFORM FOR EXPERIMENTAL WORK

This Chapter describes the minimax search platform that was used as the primary platform for our experimental work, and then goes on to investigate improvements to the basic engine.

As all other programmers of sophisticated minimax search engines have discovered: (a) the construction of an efficient and robust engine is a non-trivial problem; and (b) minimax search is very good at concealing obscure errors and bugs. Hence these preparatory experiments served not only as worthwhile research in their own right (both sections 4.2 and 4.4 were published in the ICCA Journal), but also as a development and testing ground for the engine that was to become the primary vehicle for the learning experiments presented in Chapters 5 through 8.

Section 4.2 reports on experiments with increasing transposition table efficiency; section 4.3 discusses issues related to selective search; and section 4.4 describes experiments with heuristic rules for search efficiency.

Chapters 5 to 8 concerning the learning algorithms can be read more or less independently from this Chapter. A reader with limited time who wishes to focus on the learning algorithms only could proceed directly to Chapter 5.

4.1 The Basic Minimax Search Engine

The basic search engine used for most of the experiments in this thesis is similar to that described by Slate and Atkin (1977) in their famous program Chess 4.5, which became a standard model for many subsequent programs. It uses a full-width iteratively deepened search, with alpha-beta pruning and a captures-only quiescence search at the full-width horizon (Marsland 1992). Methods other than full-width search (i.e. selective search) are discussed in section 4.3. The search was made more efficient by the use of a hash table that stores previous iteration results and detects transpositions (a *transposition table*). A simple method for improving the efficacy of transposition tables is investigated in section 4.2. An additional enhancement to the move ordering (thus improving the effectiveness of alpha-beta, see section 3.4) was provided by the implementation of a history table (Schaeffer 1983) which was used instead of the killer heuristic (Marsland 1992).

Various different evaluation functions were used for scoring positions. In some of the experiments we did not use a full positional evaluator, but only material scores.

4.1.1 The horizon evaluation and quiescence search

The searches all used a captures-only quiescence search at the search horizon. The quiescence tree consisted of all capture moves, with the side to play having the option of 'standing pat' instead of being forced to make an unfavourable capture. Pawn promotions are considered to be captures as they alter the material balance. The leaves of this capture tree are then scored according to the evaluation function.

4.1.2 Transposition tables

Conventional game-playing programs using minimax search will sometimes encounter the same position more than once, via a different sequence of moves. These *transpositions* can be detected by the use of a *transposition table,* thus avoiding duplicating search effort by repeating what has already been calculated. In addition, when using iterative deepening the transposition table can be used to aid move-ordering, thus increasing the number of alpha-beta cut-offs (see section 3.4) and reducing the size of the search tree.

Transposition tables are implemented as hash tables according to Zobrist (1970). The hash values are calculated incrementally as part of the move-making process, and the n least significant bits of the hash value (the *hash index*) are used as an index into a hash table of size 2^n. The remaining bits (the *hash key*) are used to distinguish between different positions that generate the same hash index. The hash key is stored in the table along with information about the position, including a move and a value (Marsland 1986).

If two different positions encountered during the search process generate the same hash values, then both the index and key for these two positions will be the same. This is a potentially serious problem, but the frequency of such false-match errors can be reduced to negligible proportions by increasing the number of bits in the hash value.

Hash *collisions* (Knuth 1973) occur when two positions generate the same hash index but different hash keys. There are then two positions competing for a single entry in the table, and a *replacement scheme* is needed to determine which entry is kept, and which discarded. In such cases, information is lost and the search becomes less

efficient. Transposition tables are usually made as large as memory capacity will allow, but they are rarely large enough to store the entire search tree. Typically (and especially with very fast processors or relatively small memories) only a fraction of the values encountered can be stored, resulting in numerous hash collisions. Under these conditions, search efficiency varies considerably with the choice of replacement scheme used to decide which entries are kept. In the next section we examine a simple method for reducing hash collisions, based on multiple probes of the table.

4.2 Multiple Probes of Transposition Tables

Hyatt, Gower and Nelson (1990) describe making up to eight probes of the transposition table in conjunction with a specialised replacement rule. This section describes experiments to investigate the performance of varying numbers of probes over a range of table and search sizes. The results show that considerable efficiency savings can result from the use of multiple probes, which can be of the order of 15% when the ratio of nodes searched to table size is high. These results also suggest significantly better performance than a two-level system (Breuker, Uiterwijk and Van den Herik, 1994).

4.2.1 Hash table saturation

If the hash table is sufficiently large for all the nodes encountered in the search to fit comfortably inside the table, then the number of collisions will be relatively small. When the number of nodes in the search exceeds the number of entries in the table, the table becomes *saturated.* We define the *saturation factor* of a table as the number of nodes in the search divided by the number of entries in the table.

We examine a simple method for improving decisions about which entries are retained in the table, based on multiple probes of the table. Once the table becomes saturated (saturation factor >1), the multiple probes allow us to consider a number of entries for replacement, and replace the least important entry (as determined by the replacement scheme). With a saturated table, one is not trying to locate a vacant slot (there will be few if any), but rather trying to ensure that the most important (expensive to reproduce) entries are not replaced. If the saturation factor of the hash table is less than 1, the multiple probes help to find unoccupied entries in the table. The main benefits of multiple probes occur when the table is highly saturated.

4.2.2 The transposition table experiment

This section examines the performance of a simple multiple-probe scheme, using varying table sizes, on a number of chess middle-game positions. Multiple probes of the table are shown to perform significantly better than the conventional single probe.

The evaluation function used in these experiments consisted of a material and a positional component. The positional component was restricted to the sum of piece-square table values and was dominated by the material component. The contents of the piece-square table were derived from centre control and mobility, and were calculated at the root of the search.

All positions visited during the full-width portion of the search were considered for inclusion in the transposition table. Positions visited during the captures-only quiescence search were not stored in the table. The table was used both to detect transpositions, and to aid move ordering as part of the iterative deepening process.

Each entry in the table contained: the *hash key*; the *value* of the position; a *bound flag* indicating whether the value represents an exact value, or an upper or lower bound; the best *move* from the position; and the *depth* of the full-width subtree searched to produce the entry.

4.2.3 The test set used in the experiment

There are a large number of test sets of positions that are readily available. The majority of these sets (e.g. Lang and Smith, 1993) are 'problem' positions where the task is to find the best/winning move. The experiments presented in this Chapter are concerned not with issues of move selection (indeed, the move chosen from a given position is identical with every scheme), but rather the efficiency with which the choice is made.

Breuker et. al. (1994, 1996) give a test set comprising many successive positions from a small number of games. In their experiments they measure the effects of the transposition table persisting from move to move, and so successive positions are required. In our experiments we wished to have a greater variety of positions and so chose each position from a different game.[1]

---

[1] For comparison purposes we tested the 18 position test set given by Breuker et. al. (1994). The results were very similar to those presented in this paper.

The test set used in this section comprises 30 middlegame positions (see Appendix A) and was derived from 30 different games at the 10th VSB tournament, 1996. This tournament featured 10 Grandmasters, including Kasparov, Kramnik and Anand, with an average ELO grade of 2679. Each test position is the White to play position after Black's 20th move. Of these 30 positions, 3 were searched to depth ten, 18 to depth nine, 8 to depth eight, and 1 to depth seven. The test set is also available online at `http://www.dcs.qmw.ac.uk/~martins/research/vsb.txt`

4.2.4 Search depths

The maximum search depth of each position was set to ensure that the maximum saturation factor was kept within an order of magnitude across the test set, for a given hash table size. (If all of the positions had been searched to the same depth, the average cost in terms of node-counts would have been dominated by the positions with the largest branching factors, and hence the largest saturation factors.)

Of course, because iterative deepening was used, positions that were searched to depth 10 were also searched to depths 1 through 9, and these results were used to calculate savings at lesser saturation levels (see Table 4.3).

4.2.5 Experimental issues

To understand the results and the impact of our experiment we identify and discuss some important issues below.

4.2.5.1 Measure of computational cost

We used the number of nodes visited during the search as our measure of computational cost. This includes both interior and leaf nodes, and all nodes visited during the quiescence search.

4.2.5.2 The replacement scheme

As noted in section 4.1.2 above, when a hash collision occurs it is the replacement scheme that determines which position is kept and which is discarded. (We say the replacement scheme determines *priority* for a position.) We used the traditional, basic scheme where the priority of a position is set by the depth of the subtree searched beneath that position (Marsland 1986, Hyatt *et al.* 1990). This scheme is based on the idea that deeper subtrees usually take more computational effort to reproduce than those searched to a shallower depth. If both competing positions have subtrees of equal depth, the new position replaces the old position in the table.

4.2.5.3 Multiple probes of the table

Standard implementations of transposition tables make a single attempt to place the new entry into the table. We examined a simple *multiple-probe* scheme, whereby *N* probes of consecutive entries in the table are made in order to find an entry with a lower priority that may be overwritten.

Efficient implementation of multiple probes of the hash table typically need only execute a single memory-read-and-compare operation for each probe. This means that even 8 or 16 probes use only a tiny percentage of the average time spent making move and processing a position.

When a new position (the *candidate* position) is considered for inclusion in the transposition table, the hash index of the position is used to find the corresponding entry in the table. This entry, plus its *N-1* immediate successors, are examined and the entry with the lowest priority score is chosen (the *potential victim*). The potential victim then has its priority score compared with that of the candidate position, and if the candidate position is of equal or higher priority then it replaces the victim in the table. We examined the performance of 1,2,4,8, and 16 probes. (*n.b.* 1 probe is equivalent to the commonly used single-probe method).

4.2.5.4 Two-level transposition tables (Twin)

Breuker *et al.* (1994) describe a two-level transposition table, based on a very similar scheme proposed by Ebeling (1986). This transposition table has two table positions per entry. The first table position is handled using the depth-only replacement scheme. When a first table position is overwriten, it is moved into the second table position. If the new position does not replace the first able position (using the depth-only replacement scheme), then it is always stored in the second table position, and so replaces any other position that might be there. Breuker *et al.* (1994) call this replacement scheme TWODEEP. In the results that follow, we refer to this scheme as *Twin*.

4.2.5.5 Time Stamping

We did not address the issue of entries persisting from move to move, and the hash table was cleared before each test position.

4.2.5.6 Table Sizes

To assess the effects of transposition table size on our results, we conducted our experiments using six different table sizes, ranging from 16K (1K=1024) to 1024K entries.

4.2.5.7 Move ordering

In order to measure the effect of move ordering on our results, we performed some experiments with the history heuristic disabled. The average search cost per position was more than doubled, but the savings achieved by the multiple-probe schemes were only marginally greater. From this we infer that move ordering does not play a significant part in our results, and that programs using more sophisticated move-ordering techniques than ours would still benefit from the use of multiple transposition-table probes.

4.2.6 Results

All of our results are compared to a baseline of conventional single-probe searches. Therefore, we first present the baseline node counts (Table 4.1), and then the percentage savings for the various multiple-probe schemes (Table 4.2).

16K	32K	64K	128K	256K	512K	1024K
34,760,717	30,083,815	25,943,307	22,299,509	19,572,931	17,374,383	15,853,931

Table 4.1: Average node counts per position for a single probe, by table size.

Table 4.1 shows the average search cost (measured in nodes per position) using a single probe, for each of the seven table sizes. From this Table we can see a reduction in search cost per doubling of table size of the order of 12%. Ebeling (1970) reported a 7% reduction per doubling. The difference may well be due to the greater hash table saturation factors included in our experiments.

	16K		32K		64K		128K		256K		512K		1024K	
P=2	11.6%	(5.2)	10.4%	(4.6)	10.7%	(5.9)	12.7%	(6.4)	13.4%	(5.5)	12.3%	(5.1)	10.5%	(4.2)
P=4	14.9%	(6.2)	12.9%	(5.6)	14.0%	(7.6)	17.1%	(8.8)	18.1%	(6.2)	15.2%	(5.7)	12.5%	(4.8)
P=8	15.9%	(6.2)	13.8%	(6.5)	15.4%	(8.1)	19.4%	(9.2)	19.6%	(6.4)	15.9%	(6.0)	13.0%	(4.8)
P=16	15.9%	(6.4)	14.3%	(7.1)	15.9%	(8.4)	20.6%	(9.5)	19.9%	(6.6)	16.0%	(6.0)	12.5%	(5.4)
Twin	0.3%	(12.5)	3.5%	(5.1)	3.8%	(4.2)	1.7%	(5.1)	1.8%	(4.0)	1.7%	(4.3)	0.8%	(6.6)

Table 4.2: Average individual percentage saving (standard deviation), by table size.

Table 4.2 shows the averages of the individual percentage saving compared with a single probe, for each of the multiple-probe schemes (P = 2, 4, 8, 16 and Twin), and across all table sizes. Shown in brackets are the standard deviations of these savings. We can see that substantial benefits may be obtained by using a multiple-probe scheme, and that the more probes used, the greater the saving. Using 4 or more probes instead of a single probe results in savings greater than those achieved by doubling the size of the table. There is a diminishing return, so that 16 probes perform only slightly better than 8 probes. The results from the Twin scheme showed

less saving than expected, but are within one standard deviation of those obtained by Breuker, Uiterwijk and Van den Herik (1996).

	0 to 0.1		0.1 to 1		1 to 10		10 to 100	
P=2	0.1%	(0.2)	1.8%	(2.2)	6.7%	(4.4)	10.8%	(5.8)
P=4	0.1%	(0.2)	1.9%	(2.3)	8.0%	(5.1)	13.9%	(7.3)
P=8	0.1%	(0.2)	1.9%	(2.3)	8.4%	(5.4)	15.0%	(7.7)
P=16	0.1%	(0.2)	1.9%	(2.3)	8.4%	(5.6)	15.4%	(7.9)
Twin	0.1%	(0.2)	1.3%	(1.8)	0.8%	(3.6)	0.0%	(5.0)

Table 4.3: Average individual percentage saving (standard deviation), by saturation factor.

Table 4.3 combines results from all search iterations completed during the experiments, across all table sizes grouped into saturation factor bands: 0 to 0.1; 0.1 to 1; 1 to 10; and 10 to 100. The average of the individual percentage savings compared to a single probe (and their standard deviations) are shown. From this Table we can see that the savings increase as the saturation of the table increases.

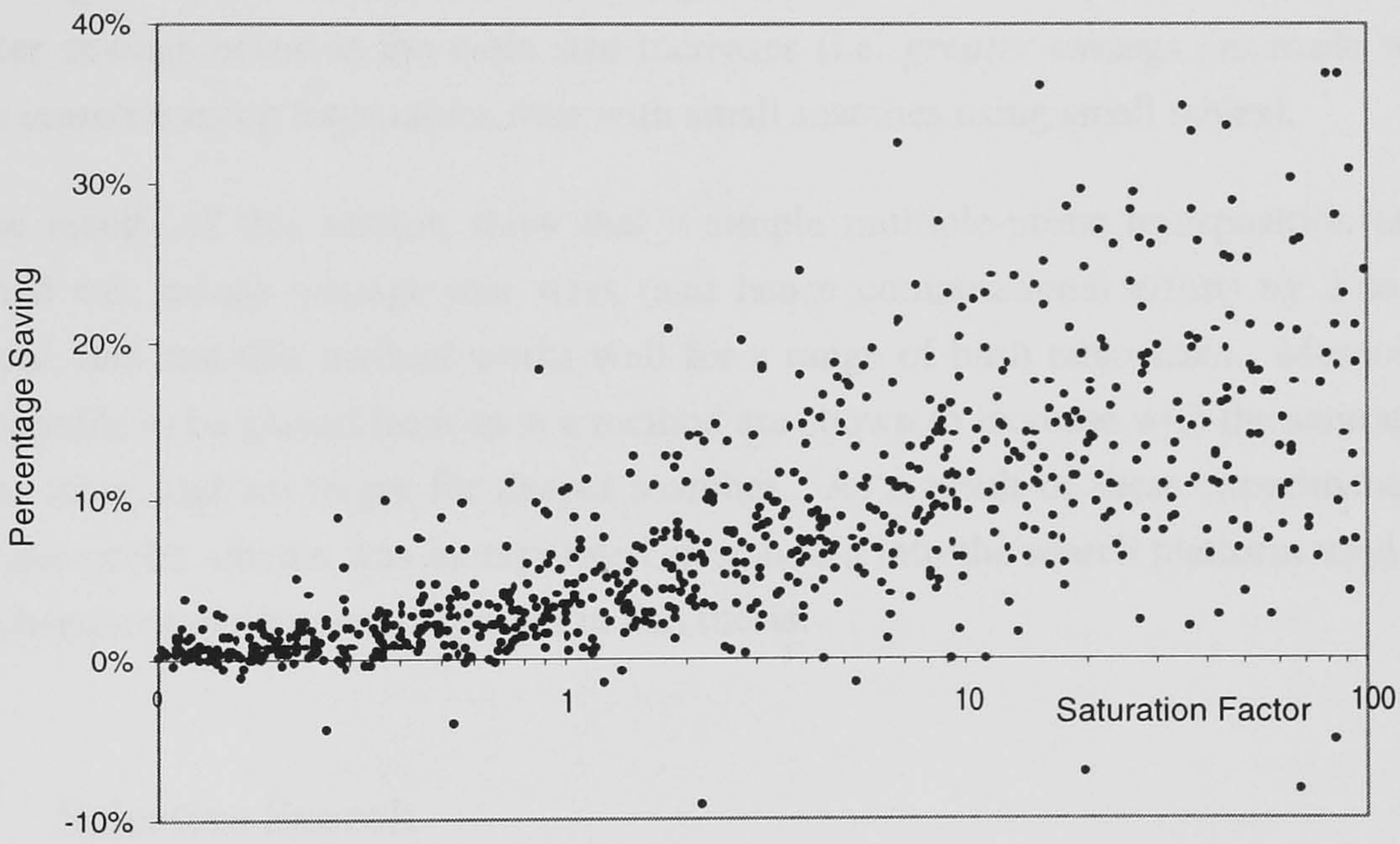

Figure 4.1: Individual iteration savings made by eight probes (P=8) compared to a single probe, plotted against saturation factor (all table sizes are included).

Figure 4.1 is a scatter plot of the savings made using 8 probes versus the saturation factor (on a logarithmic scale), combining results from all table sizes. Greater savings than those shown were achieved when the saturation factor was in excess of 100, but as this level of saturation was only reached with table sizes of 128K and below (tiny by the standards of modern computers), these results have been excluded.

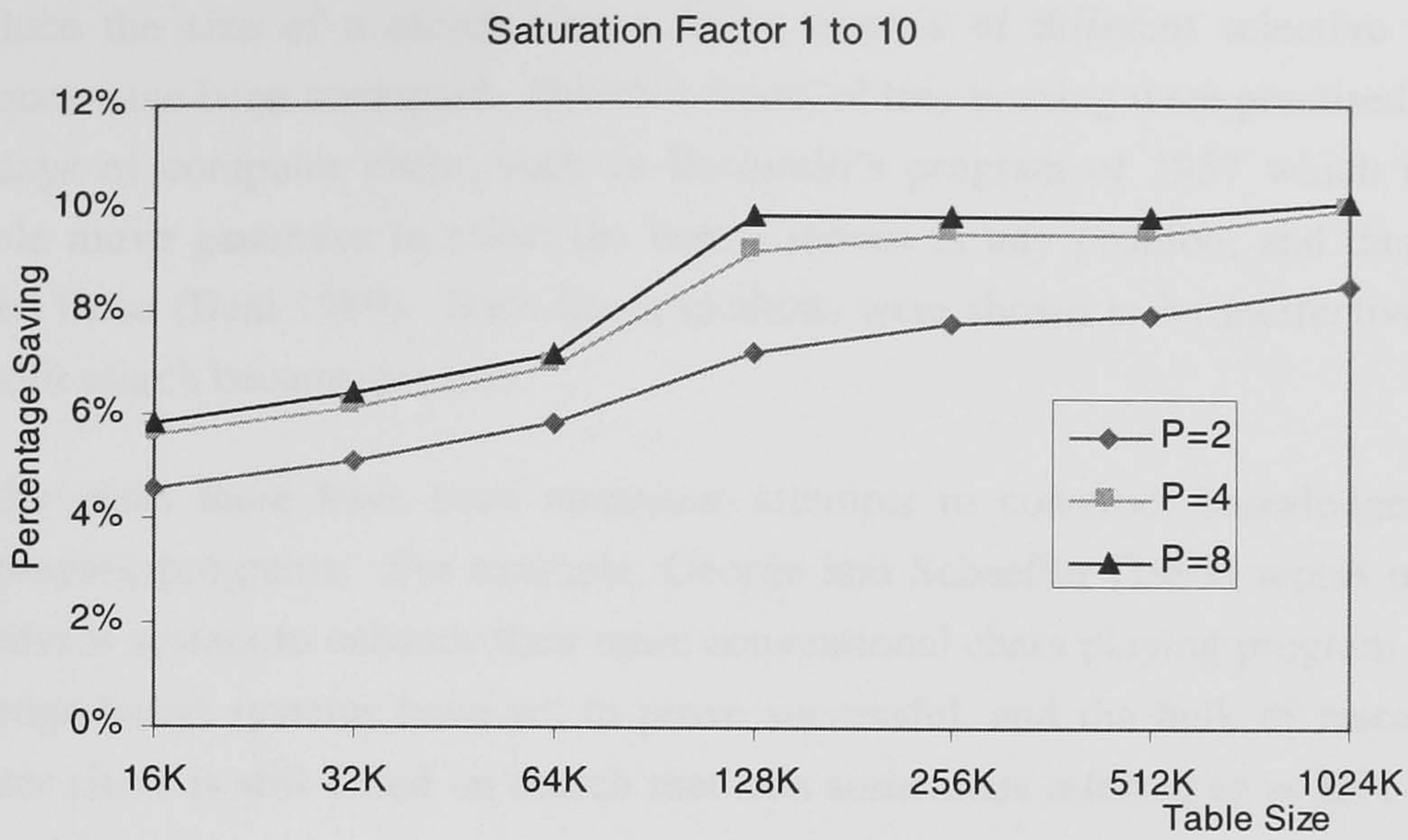

Figure 4.2: Average percentage saving by table size, for saturation factor range 1 to 10.

From Figure 4.2 we can see that within a given saturation band (in this case 1 to 10), greater savings occur as the table size increases (i.e. *greater* savings are made with large searches using large tables than with small searches using small tables).

These results of this section show that a simple multiple-probe transposition table method can reduce average tree sizes (and hence computational effort) by 2 to 15 percent, and that this method works well for a range of hash table sizes. Moreover, the benefits to be gained from such a method are shown to increase with the saturation of the table, and are larger for deeper searches. As a result of these experiments, a multiple probe scheme was incorporated as standard into the search platform used for all subsequent experiments reported in this thesis.

4.3 Selective Search

The majority of chess programs use a fixed-depth full-width search as their primary search regime, and the branching factor for the resulting search tree is on average around 35. The use of alpha-beta pruning in conjunction with good move-ordering techniques can reduce this figure to an effective branching factor in the region of 6 to 8. Although this is a substantial improvement it does not avoid the problem of combinatorial explosion, but merely delays it. For this reason, chess programmers have always been interested in investigating methods of reducing the size of the search tree, and in doing so allowing deeper searches in a given amount of time.

To reduce the size of a search tree, a large number of different selective search techniques have been attempted. Extreme forms of tree pruning were practised in the early days of computer chess, such as Bernstein's program of 1957 which used a plausible move generator to select the best 7 moves in any position, and then only searched these (Beal 1989). Such harsh methods were shown to be ineffective once full-width search became popular.

Over the years there have been numerous attempts to construct knowledge-based chess playing programs. For example, George and Schaeffer (1991) report using a chess advice system to enhance their more conventional chess playing program. Such knowledge-based systems have yet to prove successful, and the bulk of research in computer chess is still based on search methods sometimes referred to as BFI (Brute Force and Ignorance).

Other domains (e.g. *Go*) have game trees with branching factors so great that brute-force search is not a feasible approach. Selective search methods represent one possible approach to solving this problem and represent a step away from the 'ignorance' of brute-force methods, introducing a modicum of knowledge into the search in an attempt to improve its performance.

4.3.1 Alpha-beta is not selective search

It should be noted that selective search consists of pruning methods fundamentally different to that of alpha-beta. Alpha-beta is performing exactly the same computation as full-width minimax search, it is just pruning off branches of the tree that are irrelevant to the computation. This is sometimes referred to as a *backward-pruning* of the tree. Selective search techniques on the other hand perform a different computation to full-width minimax search. Of course, selective searches often themselves use alpha-beta pruning, but as Beal (1989) points out, the issue at stake is which tree should alpha-beta be used on: "what shape and size should the search tree be, or in other words, which moves should be considered for searching" (Beal 1989, pg. 67).

4.3.2 Selective search methods

It is clear that the major problem with forward-pruning of the search tree is that a good move might be excluded from the search. To avoid this happening at an early stage in the search (where of course it is most harmful), numerous selective search methods have been tried. A few examples of such methods are given below.

Tapered forward pruning (e.g. Marsland 1986) uses a shallow full-width search, after which only the N-best moves at each node are considered, with N decreasing as the search deepens.

Best-first search is a form of selective search that is closer to the methods used by human chess players. Berliner (1979) suggested the B* algorithm, which estimates pessimistic and optimistic bounds for a position, and Palay (1983) later modified B* to replace these bounds with probability distributions which were found by shallow depth-first searches. This modified B* was tested on the 300 positions from Reinfeld's *Win at Chess* (1945) and solved 81% of them (Palay 1983). The major problem with best-first searches is that the tree needs to be stored as it is searched. These techniques are unlikely to become popular among computer chess programmers unless a method can be found to guide the search and prevent the tree from growing exponentially as the search deepens (Kaindl *et al.* 1986).

4.3.3 Search extension heuristics as selective search

It is possible to view selective search from two perspectives. On the one hand we can regard it as forward-pruning branches of the search tree, throwing away what is not wanted and continuing the search along the remaining branches. Techniques such as tapered-forward pruning seem best thought of in this fashion. On the other hand, we can equally well think of selective search extending the search down certain carefully chosen branches, whilst stopping short down others. The selection of which moves to extend is what search extension heuristics are concerned with.

4.4 Search Extension Benefits: Comparison and Quantification

This section considers several search extension rules and one pruning rule that have been described in the literature. An experiment was performed to see how effective each rule was in isolation and in various combinations. This experiment was performed on a fixed test set of positions, and results were measured using node counts. The emphasis of the work was to make repeatable measurements on well-defined tasks, for future comparison with other search extension rules. In the chosen test domain chosen some extension rules were strongly advantageous compared with fixed-depth search, but disadvantageous in combination with others. Notably, singular extensions were strongly beneficial if added to a fixed depth search, but detrimental if added to a search already using check extensions, recaptures and null moves.

4.4.1 The performance of search extension heuristics

Over the past thirty years there have been numerous papers on computer chess that describe search extension rules used in various chess programs. Most of them report on performance improvements obtained within a particular program containing many other heuristics. More often than not, they do not give the games or set of positions on which they were tested. Although many valuable ideas have been successfully conveyed this way, experiments using well-defined test sets and simpler, well-specified programs have the advantage that the research community can more readily compare results and explore the circumstances in which different heuristics do well or poorly.

This section reports on an experiment to examine, on a well-defined test domain, three search extension rules and one pruning rule that have appeared in the literature. They were examined singly, including variants using alternative definitions, and in many combinations.

The goal of search extension mechanisms (and of 'forward' pruning rules) is to obtain better cost-effectiveness from searches by searching deeper down some lines than others, using some selection criterion to determine which moves are favoured. In this sense, search extension rules and 'forward' pruning rules all produce selective searches. (As noted in section 4.3.1, alpha-beta pruning is a separate matter, and can always be operated within the tree determined by the selection rules, no matter what the shape of the selected tree).

Selective searches can be compared with fixed-depth searches by examining the *effort* required to make significant discoveries. Clearly, for the same total effort to do a fixed-depth search, a selective search will search deeper down some lines and stop before the fixed-depth horizon down others. If the selection rule is beneficial, the selective search will, averaged over a large number of positions, consume less effort than the fixed-depth search to make the same discoveries.

4.4.2 The extension rules and test domain

For our experiment we chose to examine three simple-to-specify extension rules and one pruning rule that have been reported in the literature, and are commonly used in chess programs. They were: (1) check extension - moves out of check are not counted towards depth; (2) singular extensions - moves which are the only way to obtain the best minimax value; (3) capture or recapture extensions - if a move is a qualifying capture or recapture it does not count towards depth. The pruning rule (4)

was a form of null-move pruning. It was tested on its own, and in various combinations with the extension rules. All of the rules were tested in more than one variant using alternative definitions.

We examined how effective these different rule combinations are at bringing tactically significant variations within the scope of a given search effort. For this purpose we selected as a test domain 563 positions where a tactical combination was present, at varying depths, taken from the book *1001 Winning Chess Sacrifices and Combinations* (Reinfeld 1995). The positions to be used for our experiment were selected by performing an iterative-deepening full-width fixed-depth search on every WCSAC position, and retaining for the test set all those which had a well-defined tactical solution and could be solved in less than 36,000,000 nodes of search effort. (The positions selected are given in Appendix A, and are also available online at `http://www.dcs.qmw.ac.uk/~martins/research/wcsac563.txt`)

This choice was intended to ensure that all positions in the test set would be solved by all variants of the searches. This meant we could use average *effort-to-solution* as our criterion of effectiveness rather than number-of-positions-solved. Effort-to-solution was preferred over number-solved, because effort-to-solution gives credit for all reductions in search effort, and makes a useful comparison between alternative regimes that solve a given problem.

4.4.3 The baseline search, horizon evaluation and quiescence search

The baseline search used in these experiments consists of a full-width search as described in section 4.1.

The searches all used a simple captures-only quiescence search as the horizon evaluation. The quiescence tree consisted of all capture moves, with the side to play having the option of 'standing pat' instead of being forced to make an unfavourable capture. Pawn promotions are considered to be captures because they alter the material balance. The leaves of this capture tree are then scored according to the material balance at that position. For these experiments we did not use a full positional evaluator, but only material scores.

For the purposes of this experiment, we regarded the whole quiescence search as being a 'horizon evaluation'. The various search extension rules being investigated were only applied in the main part of the search - they were not applied within the quiescence search.

4.4.4 Measuring performance

The 'gold-standard' method for measuring performance is for the program to play a large number of rated tournament or match games, and measure its performance on the internationally recognised ELO rating scale. To obtain a useful rating by this method would be extremely time-consuming, especially if dozens of putative program enhancements need to be compared in various combinations. Other methods of evaluation are required. Two other commonly used techniques are playing a modified version of a program against a copy of its old unmodified self, (e.g. Anantharaman *et al.* 1988), and measuring the program's performance by testing its ability to predict moves made by human Grandmasters (e.g. Marsland and Rushton 1973). Both of these approaches have been found to be problematic (Berliner *et al.* 1989; Anantharaman 1991a, 1991b).

The most commonly used method for measuring the performance of a chess program is to provide a set of chess problem positions, and see how well it performs in trying to find the solution. There are a number of such test sets available, ranging in size from the 24-position Bratko-Kopec test to the suite of 5,500 test positions described by Lang and Smith (1993) that were optically scanned from a number of chess books.

The test-sets approach suffers from potential difficulty in comparing results. Results have often been given in terms of *number of positions solved* within a given effort budget, usually CPU time. However, this does not discriminate well between algorithms which solve nearly all the test, and does not discriminate at all between algorithms which solve the complete set. Moreover, measuring CPU time, whilst the best choice for determining competitive effectiveness for given hardware, does not allow comparisons between algorithms on different hardware, and hence between algorithms past and present. For these purposes, node counts are preferable.

Ye and Marsland (1992) used a *hit ratio* combining *number of position solved* with *node counts*. This discriminates between algorithms which solve all or nearly all the test set, but it is not clear how an effort limit can be set so that solving one extra problem is weighted fairly against a reduction in the effort overall.

Bearing in mind all these considerations, we chose to measure *total-effort-to-solution* for a specific test set, preferably one widely available or used already by other researchers. We avoided the problem of assigning *effort-to-solution* to situations when the algorithm cannot find a solution within an affordable time for the experiment by allowing the test set to consist only of positions solvable by all variants of the search algorithms.

The *effort* of finding the solution to a problem is taken to be the number of capture-tree positions (c-nodes) visited during the search, up to and including the iteration in which the correct move was found. This definition counts whole iterations only (the search has to *prove* there was no better move, not merely *prefer* it for a while), and to qualify as solved the search had to identify the correct move with the correct amount of gain.

4.4.5 The test set

We obtained problems from *1001 Winning Chess Sacrifices and Combinations* (Reinfeld 1955). This (WCSAC) test set is available in machine-readable form, (Lang and Smith 1993) with the *target* move specified along with each position.

To ensure that our test set consisted only of *solvable* positions and that each position had a well-defined solution, a preliminary program scanned all 1001 problems and selected positions that satisfied the following constraints:

(a) There had to be a single move that was tactically better than all its alternatives, and this move had to match the solution given by Reinfeld (1955). In cases where the best move found by the program differed from that given by the book (suggesting either multiple solutions or an error in the book), the position was excluded from the test set.

(b) The fixed-depth search had to return a *stable* evaluation for the target move. We defined stability as the search returning an evaluation that remained at the same value over 3 consecutive iterations, once the Reinfeld move had been found. This value was recorded as a *target gain* that had to be found by a search, as well as the *target move* to count as a solution of the problem.

(c) The position had to be such that all the search algorithms could find the specified target move, within a predetermined limit of 36 million capture-tree-generation nodes (c-nodes). This number is arbitrary and was chosen to allow the experiment to complete in reasonable time on the machines used.

Any problem that was solved by the fixed-depth search within the first three iterations (i.e. a problem of depth 1, 2 or 3) was excluded, as these were regarded as being too simple to be of much interest.

The final test set consisted of 563 problems, of which 206 were of depths 4-5 (as measured by the fixed-depth search); 176 were of depths 6 or 7; 136 were of depths 8

or 9; and 45 were of depth 10 or greater. Of these 563 problems, 99 had a solution that led directly to checkmate. These positions are detailed in Appendix A.

4.4.6 The search extension heuristics

Search extension heuristics can be domain-specific or domain-independent. Other things being equal, domain-independent heuristics are more useful. Two of the heuristics investigated here are domain-independent and, although the others are domain-specific, they are potentially applicable to a wide range of other games.

The extension heuristics were implemented so that they were applied at all levels of the main search tree. This means that any given path down the search tree may be extended more than once, and in some cases many times. All extensions are exactly one ply, in other words if an extension is applicable that move does not count towards depth. To prevent the search from being extended explosively, each move is only extended by one ply, even if more than one extension rule calls for its extension.

4.4.6.1 Check extensions

Check extensions extend the search down lines containing checks. This heuristic is known to be widely used in chess programs. We implemented it in two forms. In the first, moves out of check are not counted towards depth (e.g. Levy *et al.* 1989). In the second, it is the checks themselves that are not counted towards depth.

We found the two versions of the rule to have very similar performance effects, provided that the first tests for eligible moves at depth 0 (i.e. test for *in-check* before the node is declared to be a horizon node). From the programming point of view, this interferes with a uniform definition that depth-0 nodes are always horizon nodes, but it is still straightforward to program. It is arguable that the first definition is marginally more natural from a human perspective, whereas the second happens to have a cleaner mathematical description. After establishing that the performance effects were similar, we used the first form when testing rule combinations.

4.4.6.2 Capture extensions

The simplest form of capture extensions is to extend on all captures, i.e. no captures count towards depth. We initially included this in the experiment, but found that the effect was surprisingly bad. The cost to solution increased for nearly all positions in the test set, often by more than a factor of ten. We therefore eliminated this extension rule from the main experiment as it would have required excessive computer time to obtain detailed numbers.

4.4.6.3 Recapture extensions

A more refined idea is to extend only on *selected recaptures* (e.g. Berliner 1989). The basic idea is to extend only on captures (including promotions) that return the value of the current search to the root value stored by the previous iteration. In intuitive terms, this will have the effect that each pair of captures making a level exchange of material (i.e. capture and recapture) will receive an extra ply (from the recapture) whereas meaningless sacrificial sequences will not be extended.

Berliner (1989 p. 287) gives a rule which can be paraphrased as: "We know the expected value of the search from the previous search. A recapture is any capture that produces an evaluation within a quarter-pawn window of the expected value." Our experiment uses material-only evaluation, so the rule requires a recapture to achieve the material score exactly. Our results (given in Table 4.4 and the Figures 4.3 to 4.9) fail to show a consistent benefit, although the overall average is slightly positive.

The above rule does not say whether the *evaluation after the capture* means the backed-up value from the search subtree that follows, or a static evaluation, or some other evaluation (e.g. capture tree or equivalent). We tried many variations on this theme in an attempt to identify a version that showed a clear benefit, but were unable to find a version better than the one reported in Table 4.3. Our best rule requires both the static evaluation and the subtree-search result to achieve the expected root result. Under our version of the recapture rule, a move is considered a recapture if it is a capture or a pawn promotion, and the static material balance after it equals the stored root value R, and the search after the recapture returns a value of R.

The search used to detect recaptures is identical to the main search. Thus, there is opportunity for numerous recursive levels of recapture detection. A separate node count was maintained to establish the cost of the recapture detection search (this cost is included in the total count reported). The results show that these overhead nodes are a relatively small proportion of the search effort.

4.4.6.4 Recapture variations

We examined the performance of the recaptures rule without the static material balance requirement, and found that the number of extensions became excessive in many positions, and the resulting combinatorial explosion led to a greatly degraded performance.

We also examined other methods for detecting suitable recaptures, including use of the current search bounds for a recapture-detection search, and also performing just a

capture-tree search below each candidate move instead of a full search of depth *d*-2. These methods performed less well than our chosen criteria for recapture detection, and are not included in the results presented here.

4.4.6.5 Singular extensions

Singular extensions are a domain-independent heuristic, first described by Anantharaman et al. (1988). The concept is that a move is singular if there is no other move that achieves the same result. In practice, a move is described as singular if a search of that move to a certain depth returns an alpha-beta value significantly better than that returned by any of the other moves from a search of equal depth. In the context of our material-only evaluation, we define *significantly better* as simply meaning *greater*.

At a search depth *d*, the move *m* that is to be tested for singularity is that move which was previously found to be the best from the current position by a search of depth *d* - *r*. This move and its value *v* are usually available from the hash table. We then perform a search of depth *d* - *r* to verify that none of the alternative moves has an alpha-beta value equal or greater than *v*.

Note that we are not required to establish an exact value for these moves, only to verify that they all have values less than *v*. This enables us to use a minimal search window of [*v* - 1, *v*], thus maximising the number of alpha-beta cutoffs in the search. If *v* is an upper bound, then the position is what Anantharaman *et al.* (1988) call a fail-low node, and as the position was rejected by the earlier search, the move *m* is not tested for singularity.

The singular-detection search is performed in a manner identical to that of the main search. The extension rules that are operating in the main search are also employed in the singular detection search. This means that any given singular detection search may recursively spawn other singular detection searches lower in the tree.

We implemented two versions of singularity detection in the context of our full-width search, one which tested for the singularity of a move with a search at the current depth - 1 (i.e. $r = 1$) and the other at depth - 2 ($r = 2$). We found that *d* - 1 performed significantly better than *d* - 2, despite the fact that it incurred a much higher cost for detecting singularity. These two variations are shown in the results as SingEx(1) and SingEx(2) respectively.

If, when testing for singularity, a move is found that produces a value of greater or equal value to the previous-best move, then we refer to that move as a *singular*

exterminator. This singular exterminator is stored in a separate entry in the hash table, and in subsequent iterations can be recalled to be the first move tried in singularity detection, thus reducing the average cost of singular detection searches just as trying the previous-best move from earlier iterations reduces the average cost of regular searches.

4.4.6.6 Null moves

Null move pruning is another domain-independent heuristic that has been found to be an effective pruning method in many programs. The basic idea is to obtain, at low cost, a lower bound on the search value by performing a reduced-depth search after making a null move. There are several variants reported in the literature. Beal (1989) describes a radical approach called null-move quiescence which applies the same search depth to null moves at all levels in the tree. Goetsch and Campbell (1990) describe a more popular form in which the search depth after the null move is one less than after regular moves. Donninger (1993) describes a similar technique, but allows recursive application of null moves.

For these experiments we tried null move at reduced depths 1 and 2, and allowed recursive application. They appear as Null (1) and Null (2) respectively in the result tables.

4.4.7 Search extension results

In this section we present the results for the various rules and rule combinations, with a breakdown according to the depth of the problem.

4.4.7.1 Search efficiency

The hash table had space for 256k entries. Although not large by modern standards, this ensured running without virtual-memory paging on the machines used for the experiment, and was sufficient for all but the largest searches to complete without significant loss of efficiency due to hash table saturation. The search engine used was the same as the one described in section 4.1, and so the results are derived from reasonably efficient search implementations.

4.4.7.2 Results for each rule and rule combination

We first compared every variant of the extension and pruning rules with the baseline fixed-depth search. We then carried forward one version of each rule into various combinations with other rules. Table 4.4 lists the results for each rule on its own, and also the results for rule combinations. Overall results are given in the last column,

with a breakdown according to the problem depths in the columns headed 4,5 6,7 8,9 and 10+. The numbers are average tree sizes for each problem, measured in thousands of c-nodes.

The first line in the Table (no extensions) shows an average size of 739,000 nodes. This is the largest entry in the Table, which means all the extensions produced some benefit. The first section of the Table shows each rule applied separately. Here it can be seen that check extensions are the most effective, reducing the average tree size to 89,000 or 84,000 depending on which definition of check extension was used.

It can also be seen from Table 4.4 that tree sizes increase as expected with increasing depth of problem, averaging around an order of magnitude increase for each depth band of 2 ply. It should be noted that the figures for depth band 10+ are more erratic, being based on fewer positions. Line two of the Table gives the number of positions in each depth category.

Looking at the combinations, it can be seen that the best combination overall is *checks, recaptures and null-move pruning* (average 31,000 nodes) and the simpler *checks and null-move pruning* is second best with 47,000 nodes overall.

Depth of search (in ply)	4,5	6,7	8,9	10+	all
Number of positions (see 4.4.5):	206	176	136	45	563
Extension(s) applied:					
None	5	58	938	6,164	739
Checks (1)	2	17	256	262	89
Checks (2)*	2	16	234	271	84
Recapts	6	65	1,012	4,936	661
SingEx (1)	6	55	789	1,880	360
SingEx (2)*	6	86	1,286	2,115	509
Null (1)*	4	33	381	3,029	346
Null (2)	4	23	198	915	130
Checks+Recapts	2	20	180	149	62
Checks+SingEx	2	23	178	403	83
Checks+Null	2	9	141	116	47
Recapts+SingEx	6	69	976	1,412	372
Recapts+Null	4	26	259	991	151
SingEx+Null	7	38	344	736	156
Checks+Recapts+SingEx	3	26	265	344	101
Checks+Recapts+Null	2	11	75	105	31
Checks+SingEx+Null	3	17	120	358	64
Recapts+SingEx+Null	6	48	422	552	163
Checks+Recapts+SingEx+Null	3	20	145	340	70

* These extension heuristic variants were not carried forward into combinations.

Table 4.4: Average number of c-nodes explored (in 1000s), by search depth and by the extension(s) applied. (Appendix A contains a fuller table giving node counts for singular and recapture detection components as well.)

4.4.7.3 The bar chart view, and variation with depth of problem

It is easier to see the relationship between these numbers using bar charts. Figure 4.3 presents the last column of Table 1 as a bar chart. Here we can see, for example, the relative disappointment of re-capture extensions on their own, and easily identify the best rule combination (which includes re-captures) of *Checks, Recapts and Null.* It should be noted that the total length of the bar represents the search cost. The shading distinguishes between main search and auxillary searches to detect singular moves and recapture moves. All nodes count as part of the search effort.

Figures 4.4 to 4.6 show bar charts for the various depths of problems, in order to see whether the relationship between rule combinations looks the same with deeper problems as when averaged over all depths. It can be seen that the checks, recaptures and singular extensions show greater relative advantage on the deeper problems, whereas null-move pruning stays about the same. These differences between problem depths are mainly a consequence of the exponential nature of the search trees, which cause increasing numerical changes as extensions or pruning operate on larger and larger subtrees. This continual exponential change means that node counts and ordinary averages are an inconvenient method of comparison.

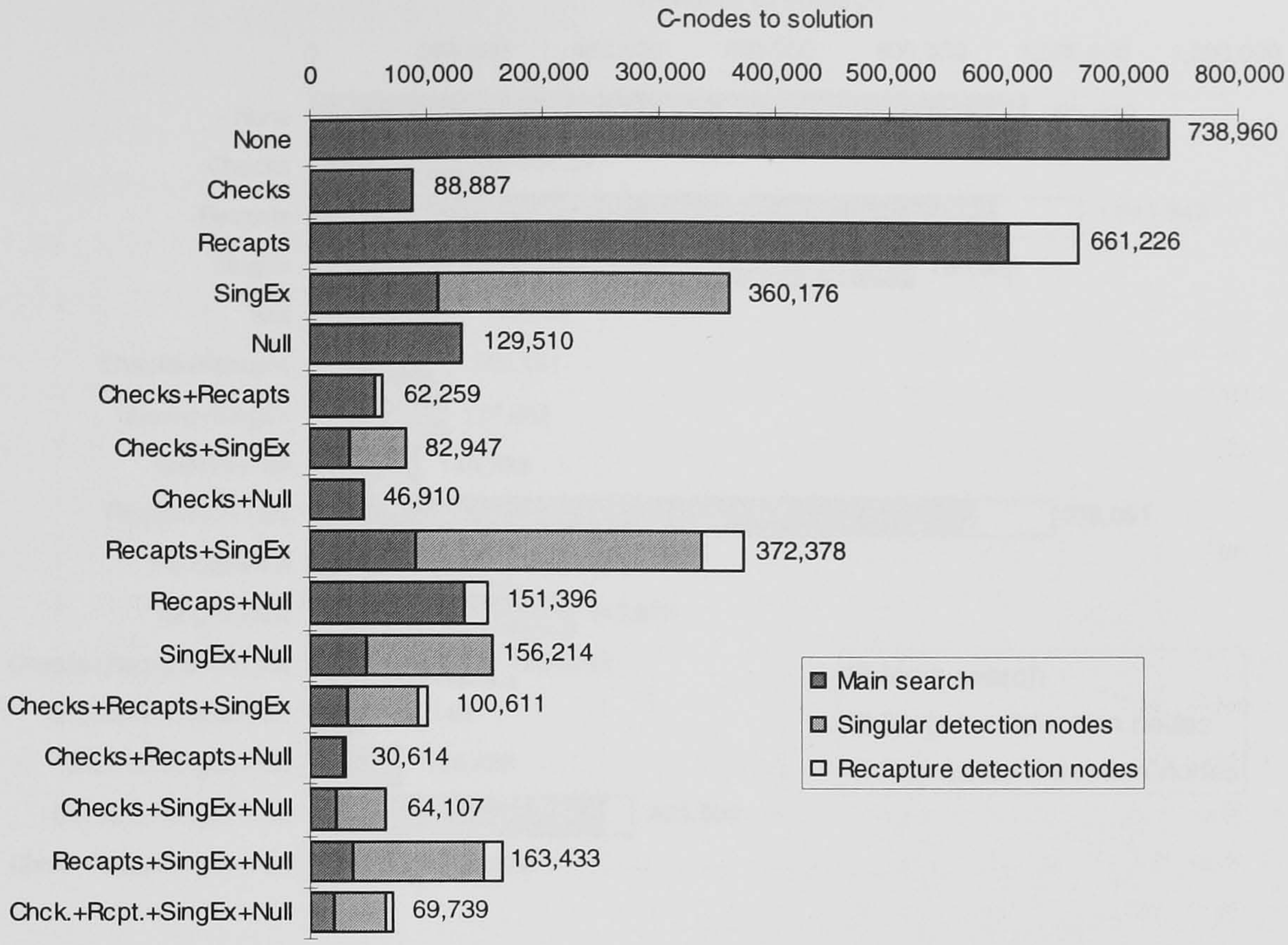

Figure 4.3: C-nodes to solution (overall).

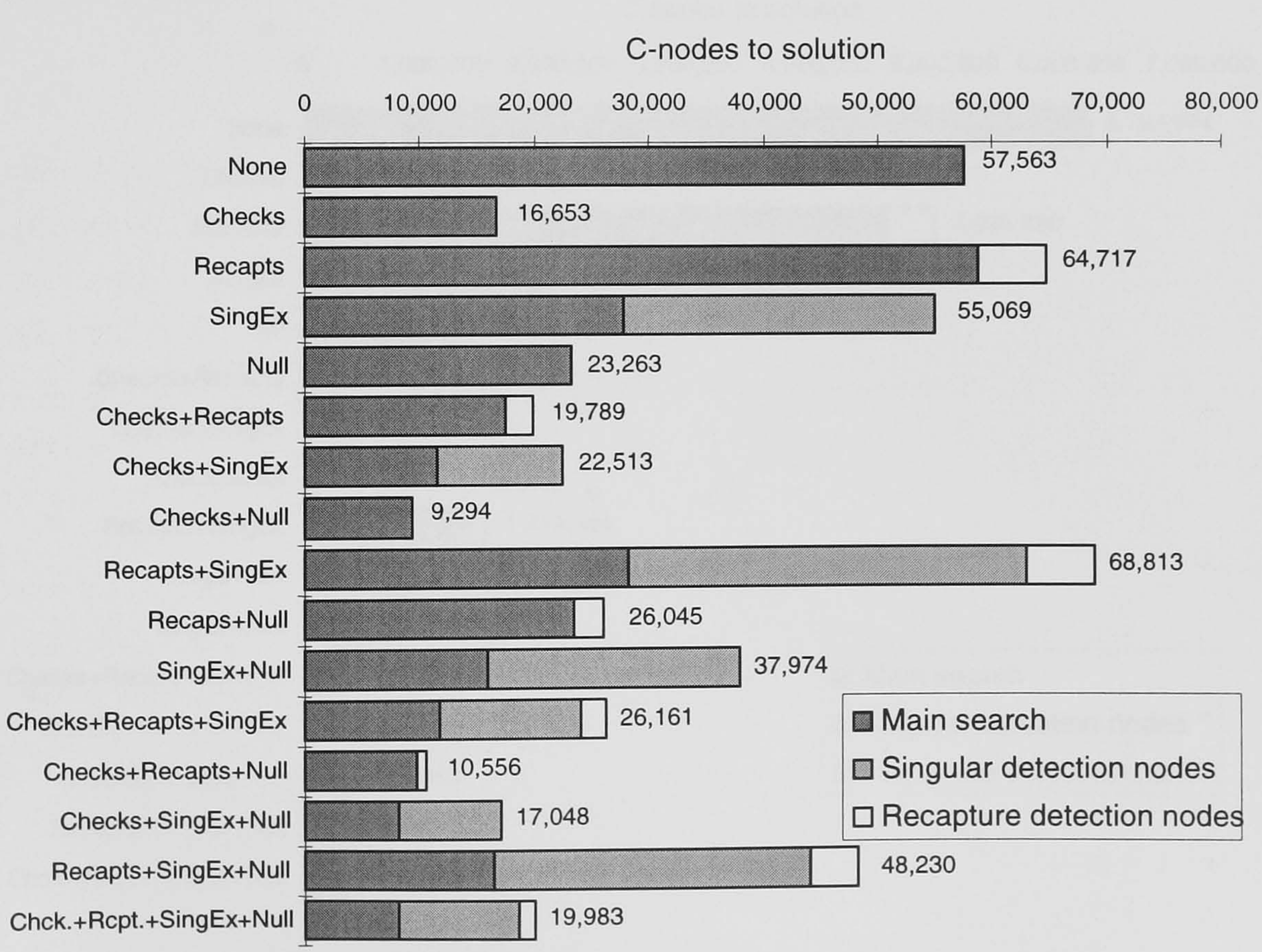

Figure 4.4: C-nodes to solution (depths 6-7).

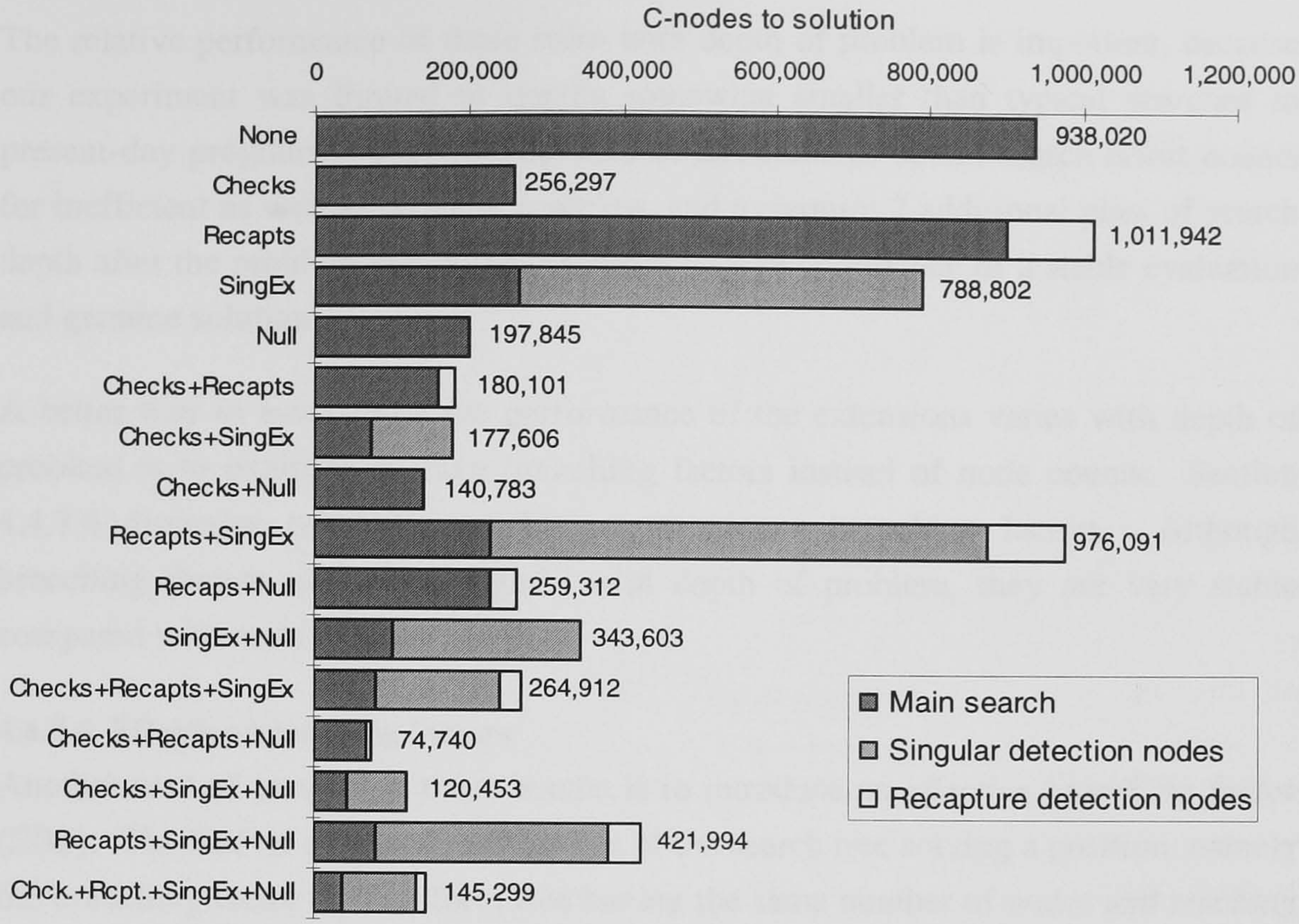

Figure 4.5: C-nodes to solution (depths 8-9).

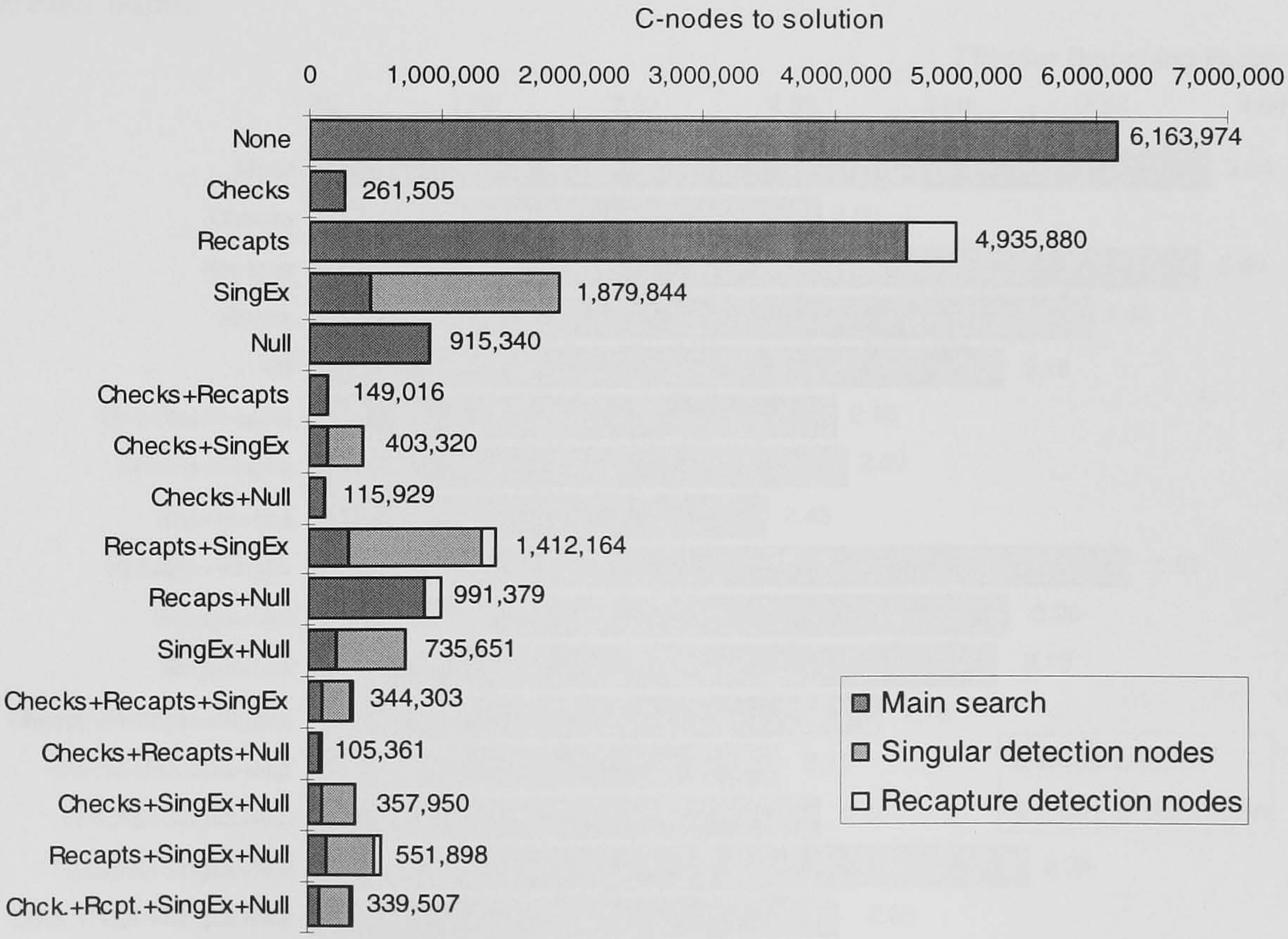

Figure 4.6: C-nodes to solution (depths 10+).

The relative performance of these rules with depth of problem is important, because our experiment was limited to depths somewhat smaller than typical searches in present-day programs. This was because of decisions to obtain search effort counts for inefficient as well as efficient searches, and to require 2 additional plies of search depth after the problem was solved in order to give confidence of a stable evaluation and genuine solution.

A better way to assess how the performance of the extensions varies with depth of problem is to examine average branching factors instead of node counts. Section 4.4.7.4 discusses one way to obtain approximate branching factors. Although branching factors also vary slightly with depth of problem, they are very stable compared with node counts.

4.4.7.4 Effective branching factors

Another way of presenting these results is to introduce an *effective branching factor* (EBF). The EBF is a measure of the size of the search tree solving a position, namely the branching factor of a uniform tree having the same number of nodes and reaching the same maximum depth. EBFs are potentially more useful than node counts in estimating the scaling effect of selective search rules when applied to problems of greater depth.

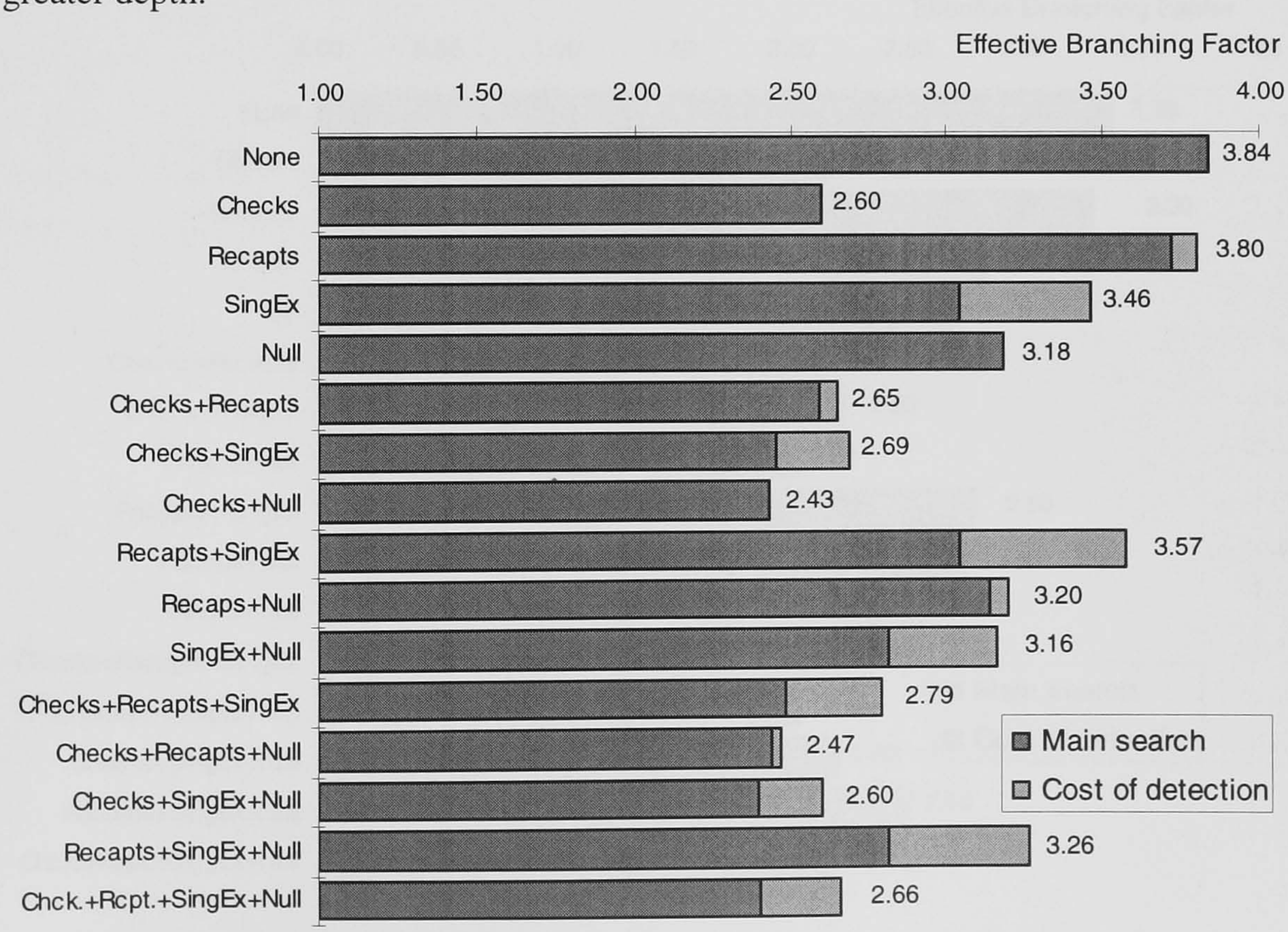

Figure 4.7: Effective Branching Factors (depths 6-7).

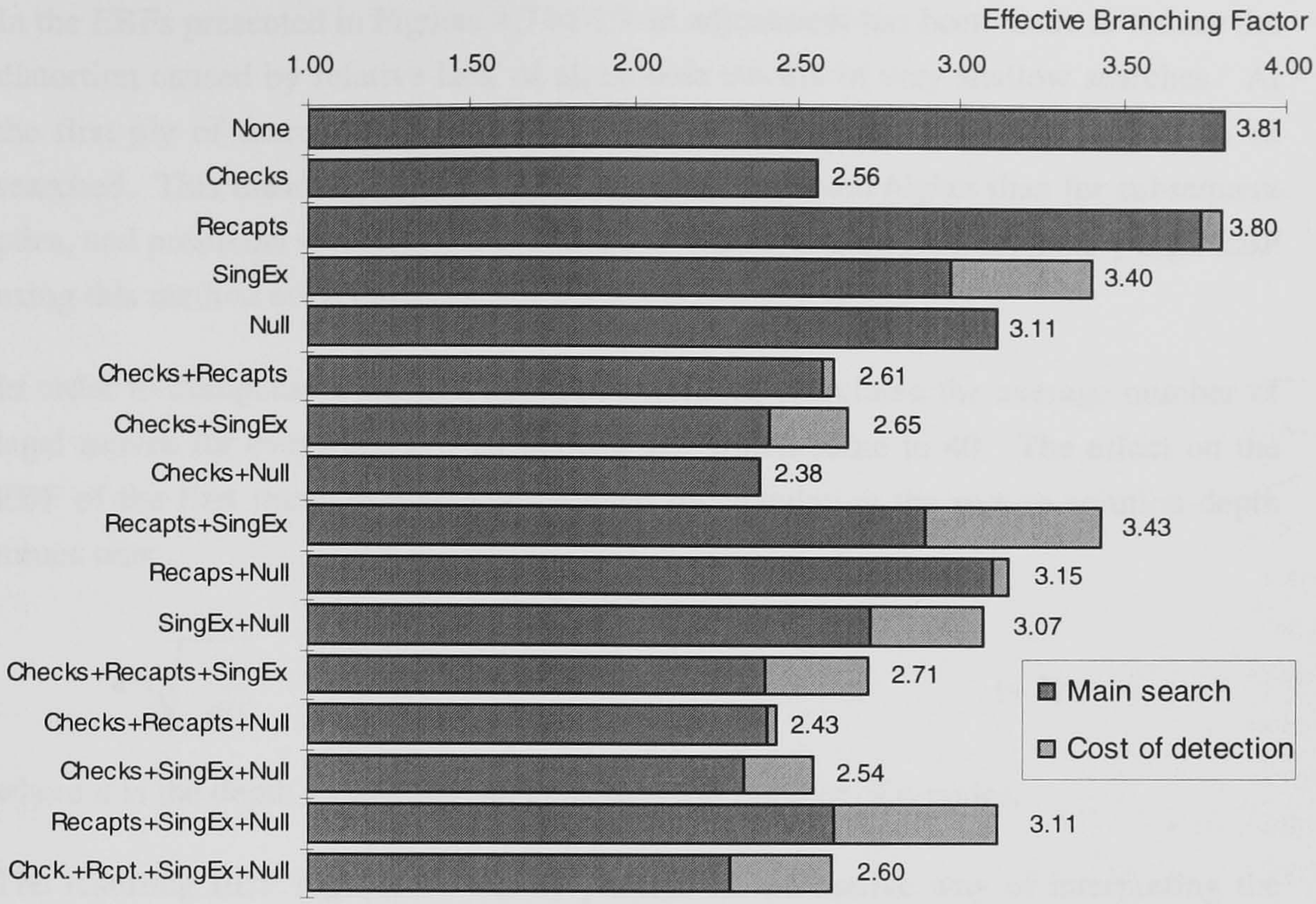

Figure 4.8: Effective Branching Factors (depths 8-9).

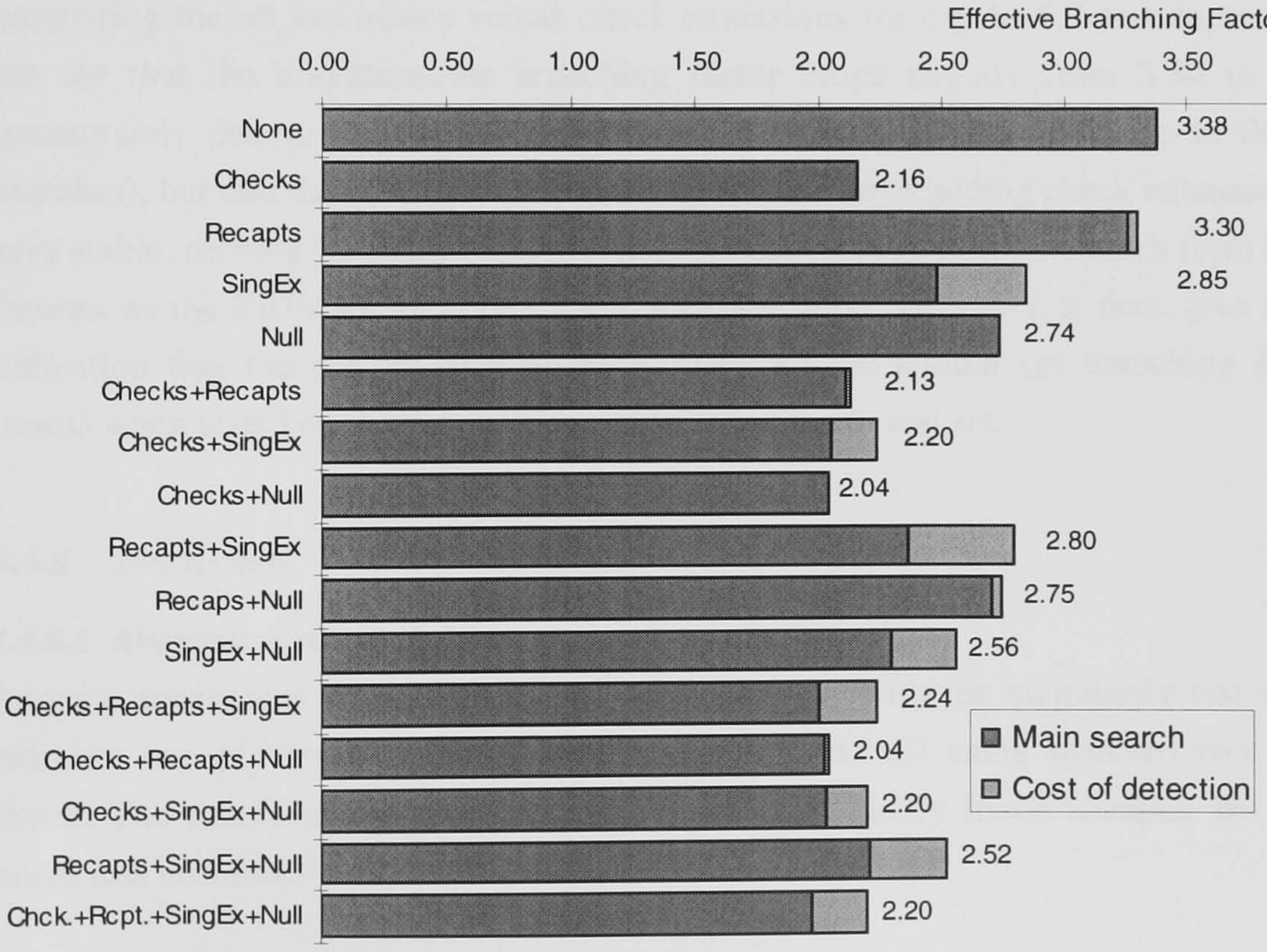

Figure 4.9: Effective Branching Factors (depths 10+).

In the EBFs presented in Figures 4.7 to 4.9 an adjustment has been made to reduce the distortion caused by relative lack of alpha-beta cutoffs in very shallow searches. At the first ply of search, no alpha-beta cutoffs are possible, so every move has to be searched. This means that the EBF for ply 1 will be much higher than for subsequent plies, and problems solved in fewer iterations will have a disproportionately high EBF using this method of calculation.

In order to compensate for this approximately, we calculated the average number of legal moves for every position in the test set, which came to 40. The effect on the EBF of the first iteration was then reduced by calculating the root to solution depth minus one:

$$\sqrt[d-1]{\frac{c}{40}} \qquad (4.1)$$

where d is the depth to solution, and c is the total number of c-nodes.

The resulting EBF Figures 4.7 to 4.9 provide an alternative way of interpreting the results. They show that the reduction in EBF achieved by the rules and rule combinations are remarkably independent of depth of problem. For example, comparing the no extensions versus check extensions for depths 6,7 and depths 10+ we see that the no-extensions branching factor drops slightly from 3.84 to 3.38 (presumably due to increasing opportunities for transpositions to occur in deeper searches), but that the reduction in branching factor due to adding check extensions is very stable, moving only from 1.24 to 1.22. It is not safe to draw too much from these figures, as the EBFs are an approximate measure only. However, it does give some indication that the performance of these rules will be similar (in branching factor terms) when tested on deeper problems than those in our test set.

4.4.8 Discussion

4.4.8.1 Alternative rule definitions

Singular extensions were tried in two versions: (1) with the singularity test using searches one ply fewer than the regular search; and (2) using searches two plies fewer. The search using two plies less is computationally much cheaper, but also much less accurate.

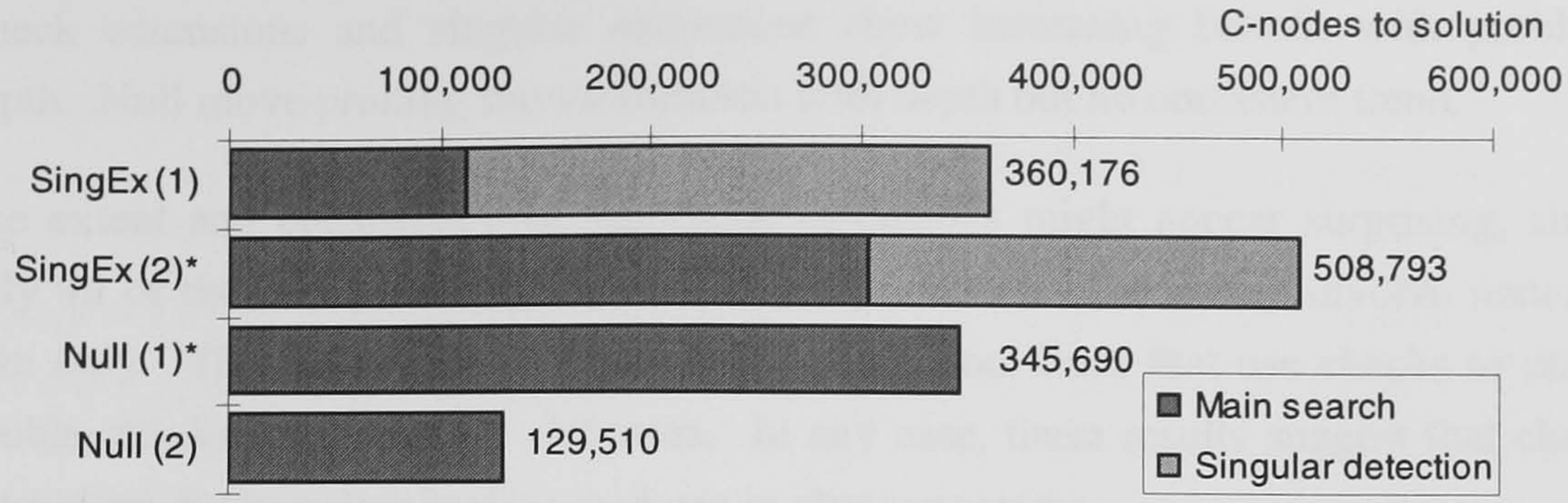

Figure 4.10: Singular detection and null move variants (overall).

Figure 4.10 shows that testing using 1-ply depth reduction produces better results, even though nearly 70% of the total effort goes into testing for singularity. With 2-ply reduction, the cost of singularity detection reduces to 40% of the total, but that total is 40% larger due to less effective extension decisions.

Singular extensions using depth-1 searches were carried forward into combinations with other rules, as was null-move pruning using depth-2 searches.

Similarly, null move pruning was tried in two versions: (1) with the null-move searched using one ply fewer than the main search; and (2) two plies fewer. The shallower the search depth after the null move, the cheaper the test, but the greater the risk that threats will be overlooked, resulting in a need for greater depth in the main search before the position is evaluated correctly. Figure 4.10 shows that for null-move pruning on our test set, 2-ply depth reduction saved more than it cost, with the total effort reduced to 40% of the 1-ply version.

4.4.8.2 Effects of each rule independently

Figure 4.3 shows that all the rules tested have beneficial effects when added to the baseline fullwidth search. These range from recaptures, which averaged a 10% reduction, to check extensions, which produced a large 88% overall reduction in tree sizes.

The breakdown according to depth category (Table 4.4) shows that check extensions, singular extensions, and null moves have large gains in all depth categories. Recapture extensions are not consistently beneficial: they showed a net loss on depth categories up to depths 8,9, and only a (relatively modest) gain for depths 10+.

The 10+ depth category has to be treated with some caution, since it consists of fewer problems than the others - only 45 positions compared with 136 in depth category 8,9.

Check extensions and singular extensions show increasing benefit with problem depth. Null move pruning shows variation with depth but no consistent trend.

The extent and consistency of the check extensions might appear surprising, since only 99 of the 563 problems are checkmate problems. The others involve material gain only. The advantage in these cases comes from lines that use checks to make double attacks or undermine defences. In any case, these results suggest that check extensions deserve their widespread use in chess programs.

4.4.8.3 Extensions in combination

Figure 4.3 shows that singular extensions added to a fixed-depth search produced an overall reduction of about 30%. Figure 4.3 also shows that adding singular extensions to a search already including check extensions only produced a further reduction of 7%. Table 4.4 shows that this reduction came mainly from problems of medium depth (8,9) with increases in effort when adding singular extensions to check extensions on shallower and deeper problems.

Singular extensions produced even less advantage on searches already combining check extensions with other rules. In particular, adding singular extensions to a search using check extensions, recapture extensions and null move pruning resulted in considerably worse performance in all depth categories.

The augmentation of check extensions with a rule similar to singular extensions was reported as beneficial as a specialised aid to detecting mating sequences in the quiescence search (Beal 1984). This used the special case of singular 'out-of-check' moves which have close to zero-cost detection, and therefore might be expected to show more net benefit.

One factor that would perhaps reduce the measured benefit of singular extensions in our experiment compared with results reported by Anantharaman, Campbell and Hsu (1988), is that they reduced the cost of singularity detection by not testing moves at the root which, as they have the largest sub-trees, have the highest cost to test. We performed singularity detection at all nodes, including the root.

Considering the extent of variation between problems, our overall conclusion is that singular extensions added no significant advantage to searches that already included check extensions. In contrast, null-move pruning shows a benefit in all combinations tested.

Table 4.4 and Figure 4.3 show that recapture extensions achieved a benefit when added to fixed-depth search, when added to check extensions alone, and when added to an existing combination of check extensions and null-move pruning, but a loss when added to every other combination. The benefit or loss from recapture extensions was relatively small and varied with depth category. The overall conclusion has to be that no significant advantage or disadvantage was observed.

4.4.8.4 Overall comments regarding search extensions

The experience with singular extensions and check extensions is a reminder that performance advantages from selective search techniques are always highly sensitive to the combination of co-existing selection rules and the choice of tests. It is necessary to perform thorough tests with all combinations before concluding any overall utility from any selection rule.

It is perhaps surprising that the singular extension and recaptures rules we have examined do not show clearly decisive benefits, since they have been used in many chess programs. Of course, the experiments reported here have been limited to tactically decisive problems with material-only evaluations. Singular extensions and recapture extensions might have been expected to work well on tactical sequences, so part of the result is unexpected.

4.5 Benefits of the Preliminary Experiments

The investigation of search engine improvements in sections 4.2 to 4.4 was worthwhile (and resulted in two publications). As a result of this experience we obtained not only the more sophisticated engine containing the search enhancements, but also simple robust versions of the search engine and tools for collecting and processing search statistics. After this early work, we were able to conduct the published learning experiments using the simplest feasible engine that was robust and had been well tested.

4.6 Shogi and the Shogi-Playing Search Engine

The search-related methods described in this Chapter are all essentially game-independent. As a result, many of the above techniques would be useful in the construction of programs to play other games.

In addition to conducting experiments in the well-established domain of chess, we also investigated *shogi*, another complex game that is considered by many researchers to be the next challenge beyond chess for computer game-playing (Matsubara, Iida and Grimbergen, 1996). The features of shogi that provide this challenge and the results of experiments conducted in this domain are described in detail in Chapter 6.

5 LEARNING CHESS EVALUATION COEFFICIENTS

This Chapter introduces the main topic of this thesis – learning evaluation coefficients in a complex domain. The methods for utilising temporal difference learning in complex game domains detailed in Chapter 3 are combined with the chess-playing platform described in Chapter 4. We describe experiments where we attempt to learn the relative values of chess pieces. The learning is obtained entirely from a series of randomised self-play games, without access to any form of expert knowledge. The learning system does not benefit from seeing the play of a well-informed opponent against it, nor does it examine games played by experts. The only chess-specific knowledge is provided by the rules of the game. We show that we are able to learn suitable piece values, and that these values perform at least as well as piece values widely quoted in elementary chess books.

A combination of machine-learning methods, including TD learning, was earlier used to learn chess piece values (Levinson and Snyder 1991), and coarse-grained piece values (Christensen and Korf 1986).

The same method of applying TD(λ) to minimax. as described in Chapter 3 and first published by Beal and Smith in 1997, was later reported to be successful by Baxter, Tridgell and Weaver (1998). They improved weights for a complex chess evaluation function consisting of positional terms as well as piece values, when playing online against knowledgeable opponents (see section 3.6), but they provided piece weights as initial knowledge in order to obtain good performance.

5.1 The Relative Value of the Pieces

Probably the first heuristic to be taught to most beginners is the value of the pieces: that knights and bishops are worth about three pawns; rooks about five pawns; and the queen about nine pawns (the king is not given a value as it cannot be captured). Thus under this scheme, it would be considered a fair exchange to trade a rook for a bishop and two pawns, or a queen for two bishops and a knight.

This simple numerical scheme provides a crude evaluation of how 'good' (i.e. likely to lead to victory) a given position is. Such a scheme (or others very similar) provides the backbone for the evaluation function of almost all chess-playing computer

programs', including IBM's Deep Blue. For high performance play the basic scheme is augmented with numerous, often very elaborate, additional scoring terms.

The material-only scheme forms one of the simplest examples of evaluation functions described in section 3.4.1. The experiments presented in this Chapter are concerned with finding suitable weights for each of the five piece types: pawn, knight, bishop, rook and queen.

5.2 Temporal Difference Learning in Chess

The TD learning process is driven by the differences between successive predictions of the probability of winning during each game. Throughout the course of a single game, a record is kept of the value returned by the search after each move, and the corresponding principal position (see section 3.5.2).

In principle, TD(λ) weight adjustments can be made after each move or at any arbitrary interval, but for game-playing tasks the end of every game is the most convenient point to actually alter the evaluation weights. The update rule for applying weight adjustments at the end of each game is:

$$w \leftarrow w + \sum_{t=1}^{T} \Delta w_t \qquad (5.1)$$

where T is the time index at the end of the game.

As described in section 3.4, a sigmoid squashing function is used to convert the evaluation score of the principal position (i.e. the leaf of the principal variation) into a prediction of the final outcome of the game. We sometimes refer to these predicted outcomes as *prediction probabilities* as they are in the range 0-1.

The use of this squashing function ensures that weights having little effect on the prediction are adjusted less than weights to which the prediction is more sensitive.

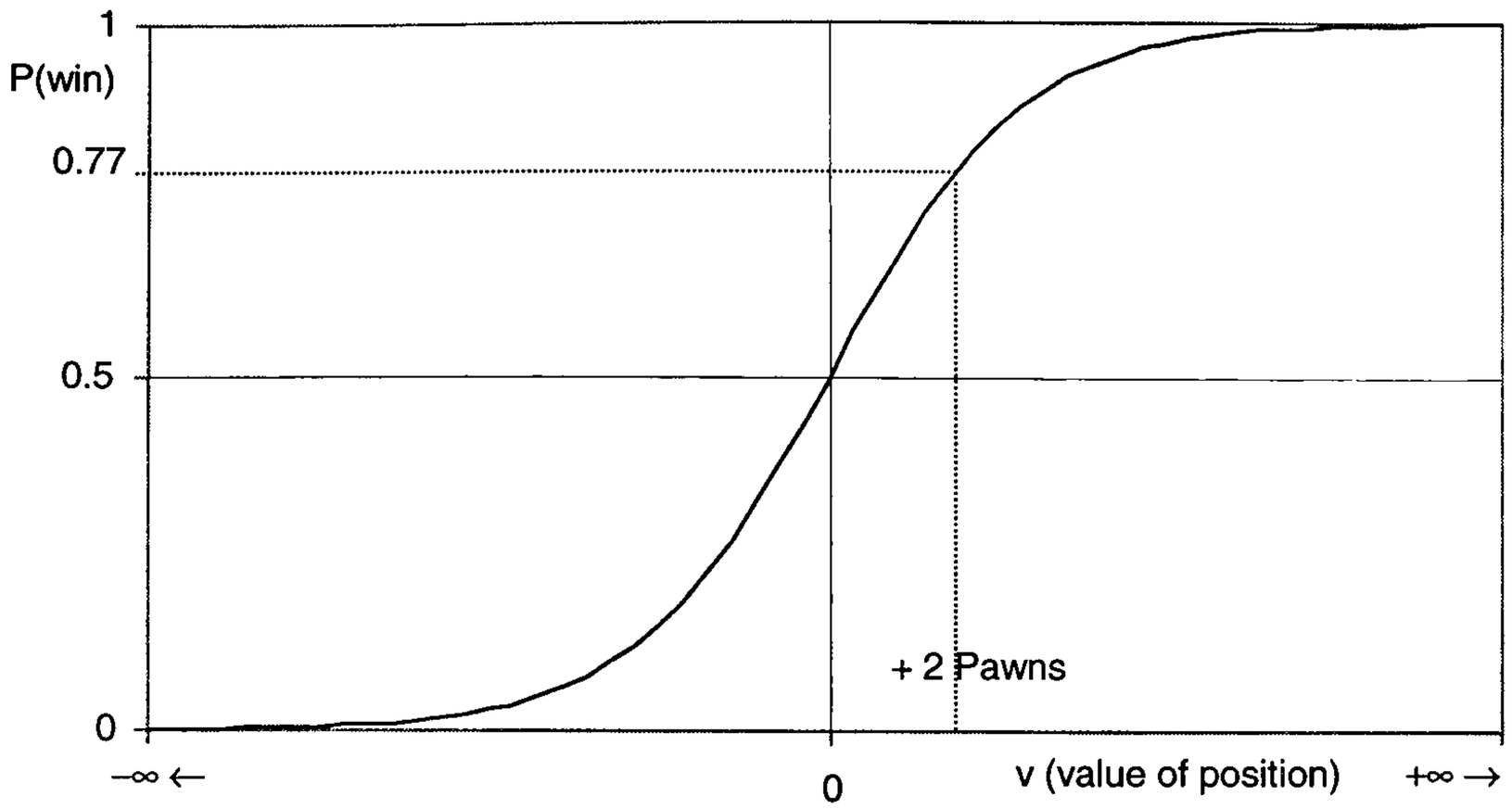

Figure 5.1: Graph showing the conversion of position value into prediction probabilities (including an example using piece values learnt in Run A from section 5.3).

Figure 5.1 shows the conversion of position value into prediction probability. The example score of 2 pawns (using pawn = 0.60 from Run A in section 5.4) is converted into a probability of winning of 0.77. Of course, the resulting probabilities for any given material advantage will vary according to the piece values that have been learnt. For example, using the values from Run B, 2 pawns advantage converts into a probability of winning of 0.74.

5.3 The Basic Learning Experiment

As outlined in section 3.5, the piece values being learnt are used to evaluate the leaves of a minimax search tree, and temporal difference learning is used to adjust these values over the course of a series of games.

We attempted to learn suitable values for five adjustable weights (pawn, knight, bishop, rook and queen). To demonstrate that the values learnt are reasonable, we played matches between two programs that differed only by one using the piece values learnt during the experiments and the other using values given in many elementary chess books of pawn=1, knight and bishop=3, rook=5 and queen=9. The programs using the learnt values consistently score well over fifty percent in matches of this sort.

5.3.1 The search engine

The search engine used in the experiments was described in section 4.1. The piece values being learnt (those for pawn, knight, bishop, rook and queen) were used as the evaluation function. The *material balance* for a position was calculated as the sum of all the values of the side to move's remaining pieces, minus the sum of all the values of the opponent's remaining pieces.

5.3.2 Experimental details

The games reported on in this section were played using a four-ply fixed-depth search. At the start of each learning experiment the piece values were all initialised to 1. For these experiments we used TD(λ) control parameters of $\alpha = 0.05$ and $\lambda = 0.95$.

To prevent the same games from being repeated, the move lists were randomised. This has the effect of selecting at random from all tactically equal moves, and has the added benefit of ensuring that a wide range of different types of position are encountered. It should also be noted that these experiments have learnt values within a material-only evaluation function. We would expect the material values learnt to be at least slightly different if the evaluation function included positional scoring terms (see Chapter 7).

5.3.3 Basic learning results

We conducted five runs (A-E) of 10,000 games each. Each run differed only in choice of the initial seed for the random number generator, ensuring that each consisted of entirely different game sequences.

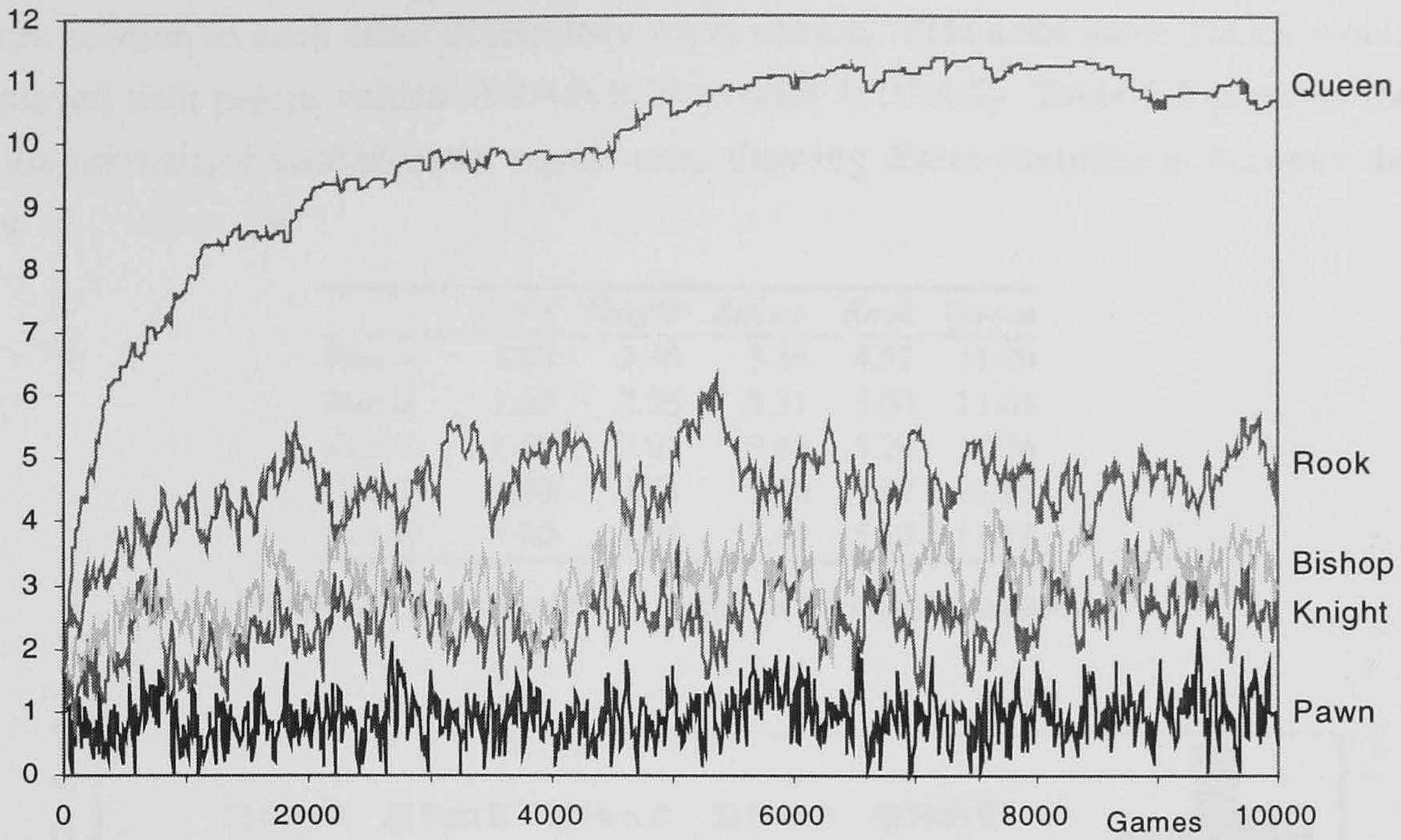

Figure 5.2: Graph of learnt values from a typical single trial (Run A). The absolute values have been normalised so that the average value of a pawn over the last 2,000 games is fixed at 1.

Figure 5.2 shows the evolution of the piece values for a single run over 10,000 games. From the graph it can be seen that the value of the queen is quickly established to be greater than that of the other pieces, and that after a few hundred games the relative ordering of the pieces has been established. There are a few minor fluctuations, where the value for knight is briefly above that of bishop, but these are quickly remedied. The four other runs (B-E) differed from this only in the choice of the random number seed, and they all produced learning traces very similar to those shown here.

Table 5.1 shows the piece values learnt. To avoid fluctuations in the weights due to noise from the stochastic component of the search engine, these values were calculated by averaging over the last 20% of games in each of the five runs.

	Pawn	*Knight*	*Bishop*	*Rook*	*Queen*
Run A	0.60	1.66	2.02	2.75	6.61
Run B	0.58	1.49	1.93	2.92	6.43
Run C	0.53	1.60	1.93	2.81	6.79
Run D	0.58	1.56	2.02	2.81	6.64
Run E	0.57	1.47	1.78	2.92	6.36

Table 5.1: Learnt values for each trial, averaged over the final 20% of the runs.

The absolute values of the weights could conceivably drift to be all very large, or all very small. What matters most, however, are their values relative to one another, because when the learnt piece values are used by a playing program, the value of the

pieces relative to each other determines move choice. (Thus the same games would be played with pieces values of 2:4:6:8:10 as with 1:2:3:4:5) Table 5.2 presents the results normalised so that pawn equals one, allowing direct comparison between the runs.

	Pawn	*Knight*	*Bishop*	*Rook*	*Queen*
Run A	1.00	2.76	3.36	4.57	11.00
Run B	1.00	2.56	3.31	5.00	11.01
Run C	1.00	3.01	3.62	5.29	12.76
Run D	1.00	2.71	3.50	4.87	11.51
Run E	1.00	2.58	3.13	5.13	11.17

Table 5.2: Learnt values for each run, normalised to pawn=1.

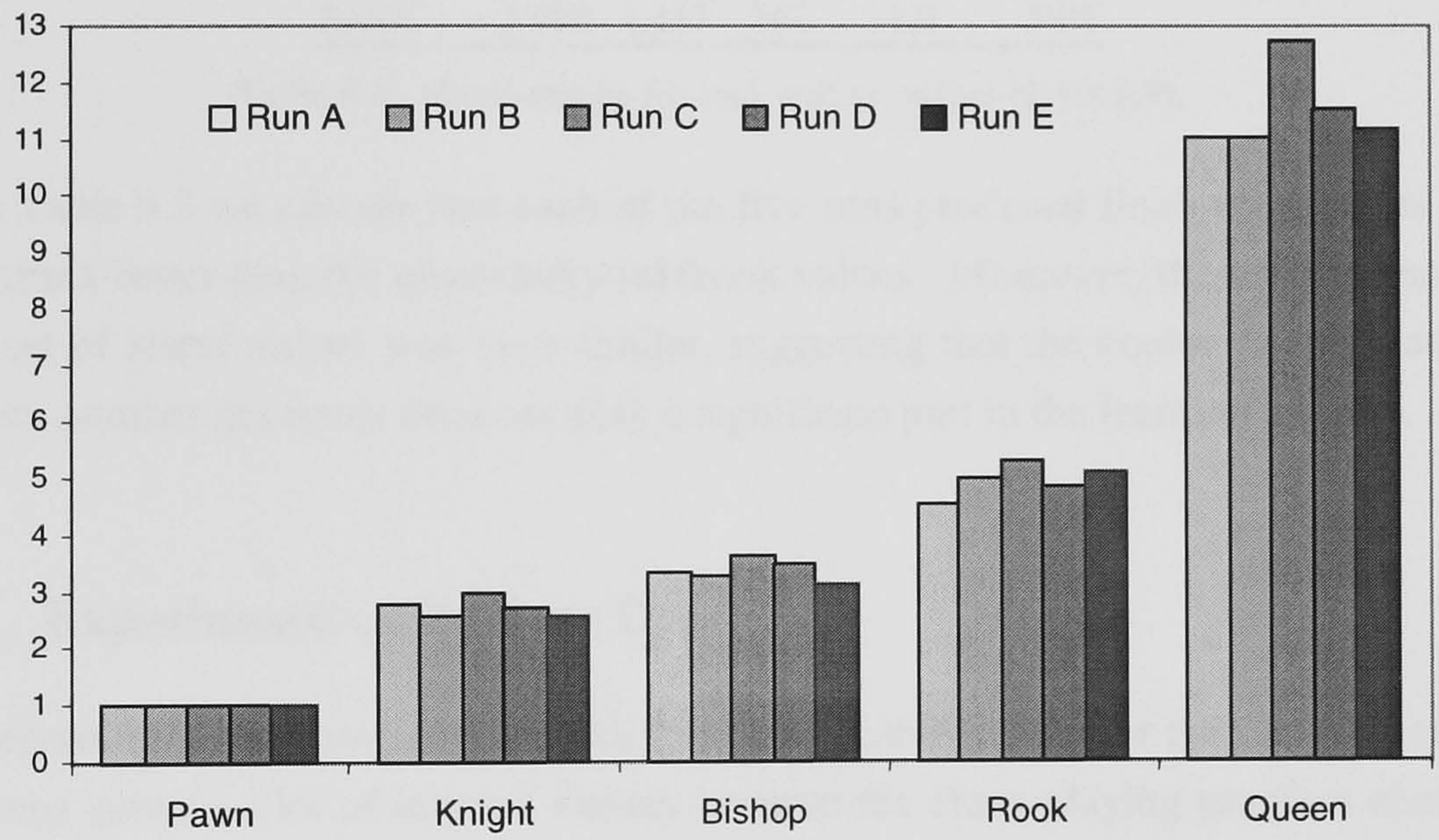

Figure 5.3: Normalised learnt piece values from 5 runs at search depth 4.

From Figure 5.3 we can see that similar piece values were learnt by each of the five runs, and that the relative values of the pieces in each case are similar to the widely quoted values taught to human beginners. By 'similar' we mean similar in the context of piece values being learnt by self-play from zero initial knowledge. In this context, the play is dominated by relative orderings of pieces and piece combinations, rather than numerical totals.

5.3.4 Basic results from matches using learnt values

To verify that the final values were reasonable, five matches were played between two identical search engines, one using the values (1:3:3:5:9) and the other using the newly learnt weights from one of the five runs.

A match consisted of 2,000 games, alternating black and white. Games that ended in mate were scored as 1 point for the winning side. Games that ended in a draw according to the laws of chess (stalemate, repetition, insufficient material) were scored as 1/2 point for both sides. Games that were unfinished after 400 ply (200 moves each) were scored as win for one side only if both programs' evaluation functions agreed that side was ahead on material, otherwise the game was scored as a draw.

	Played	*Won*	*Lost*	*Drawn*	*Percent.*
Run A	2,000	1,116	782	102	58%
Run B	2,000	1,117	766	117	59%
Run C	2,000	1,114	777	109	58%
Run D	2,000	1,112	781	107	58%
Run E	2,000	1,117	762	121	59%

Table 5.3: Match results for each trial vs. values (1:3:3:5:9).

From Table 5.3 we can see that each of the five runs produced final piece values that performed better than the elementary textbook values. Moreover, the performance of each set of learnt values was very similar, suggesting that the choice of seed for the random number generator does not play a significant part in the learning process.

5.4 Experiments at Various Depths

We repeated the basic experiment with five other search depths for the chess program. The runs contain a lot of internal variety because the chess playing program chooses randomly between materially-equal values, producing a variety of games for any given set of material values. This variety enables the learning process to achieve values that reflect values averaged over all positions. Nevertheless, to confirm that the results were not critically dependent on the random choices, we again ran each depth experiment with five different random-number seeds.

	Depth 1		*Depth 3*		*Depth 5*	
Pawn	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)
Knight	1.86	(0.09)	1.75	(0.10)	2.14	(0.12)
Bishop	2.11	(0.15)	2.31	(0.08)	2.49	(0.13)
Rook	3.34	(0.20)	3.88	(0.20)	4.06	(0.32)
Queen	7.26	(0.64)	8.08	(0.19)	8.11	(0.35)

Table 5.4: Learnt piece values from depths 1,3,5

	Depth 2		*Depth 4*		*Depth 6*	
Pawn	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)
Knight	2.75	(0.22)	2.72	(0.17)	2.91	(0.08)
Bishop	3.52	(0.33)	3.38	(0.19)	3.49	(0.09)
Rook	4.84	(0.35)	4.97	(0.27)	5.08	(0.09)
Queen	11.04	(0.55)	11.49	(0.74)	10.68	(0.28)

Table 5.5: Learnt piece values from depths 2,4,6

Tables 5.4 and 5.5 shows the average values learnt at depths 1,3,5 and 2,4,6 respectively. In both cases the values are normalised to pawn=1 for ease of comparison. The *absolute* values learnt in each run are presented in Appendix B.

It is a well known feature of minimax searches to fixed depths that there are often fluctuations of behaviour with the parity of ply (Beal and Smith 1995). This is probably caused by evaluations being biased either towards or against the player to move. The direction of bias, and its magnitude, depend on the nature of the evaluation function and the characteristics of the game. This makes comparison of values learnt at even depths of search with those learnt at odd depth problematic. For this reason, the results are presented in two sections: depths 1,3,5 and depths 2,4,6.

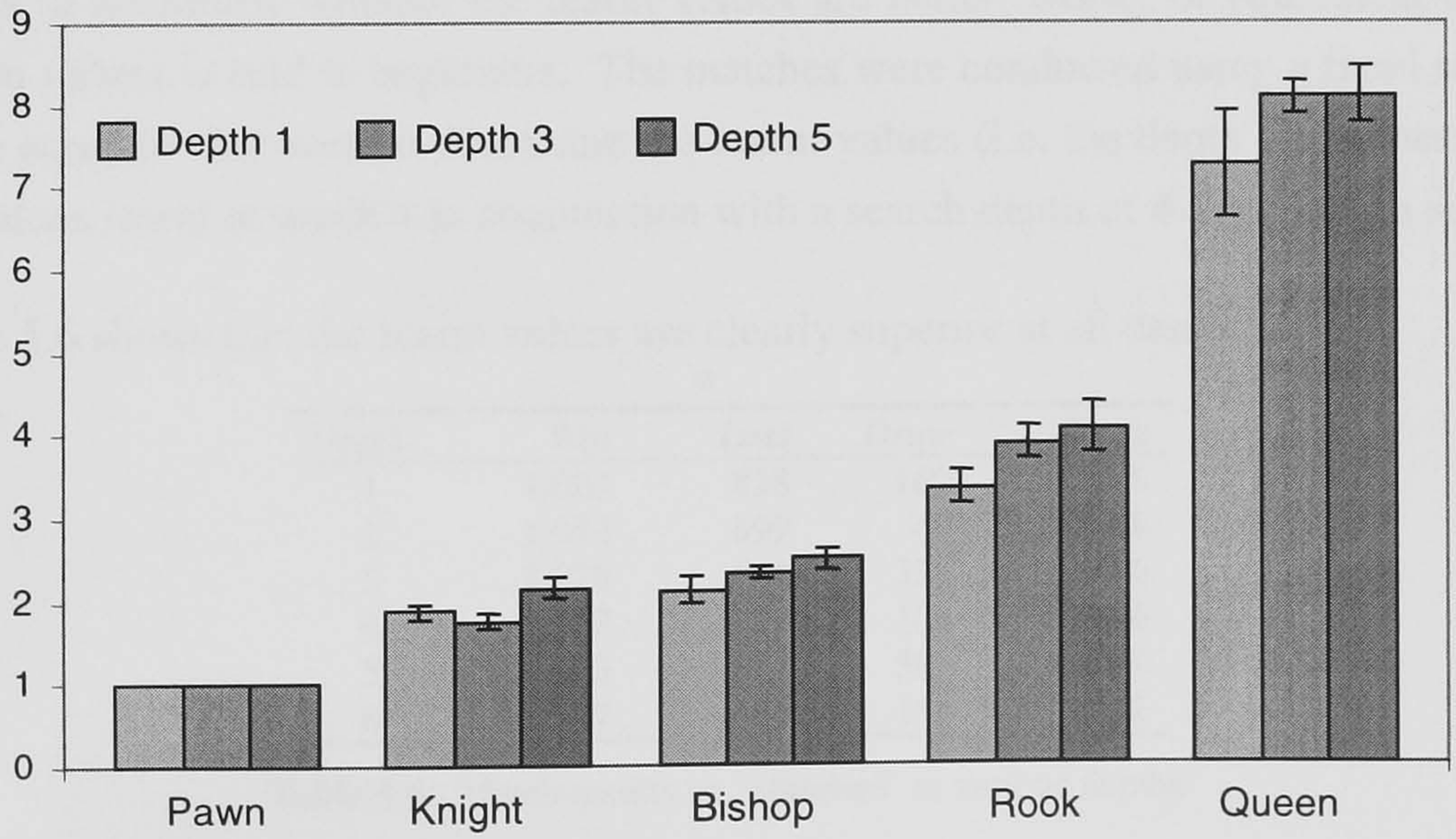

Figure 5.4: Learnt piece values from depths 1,3,5.

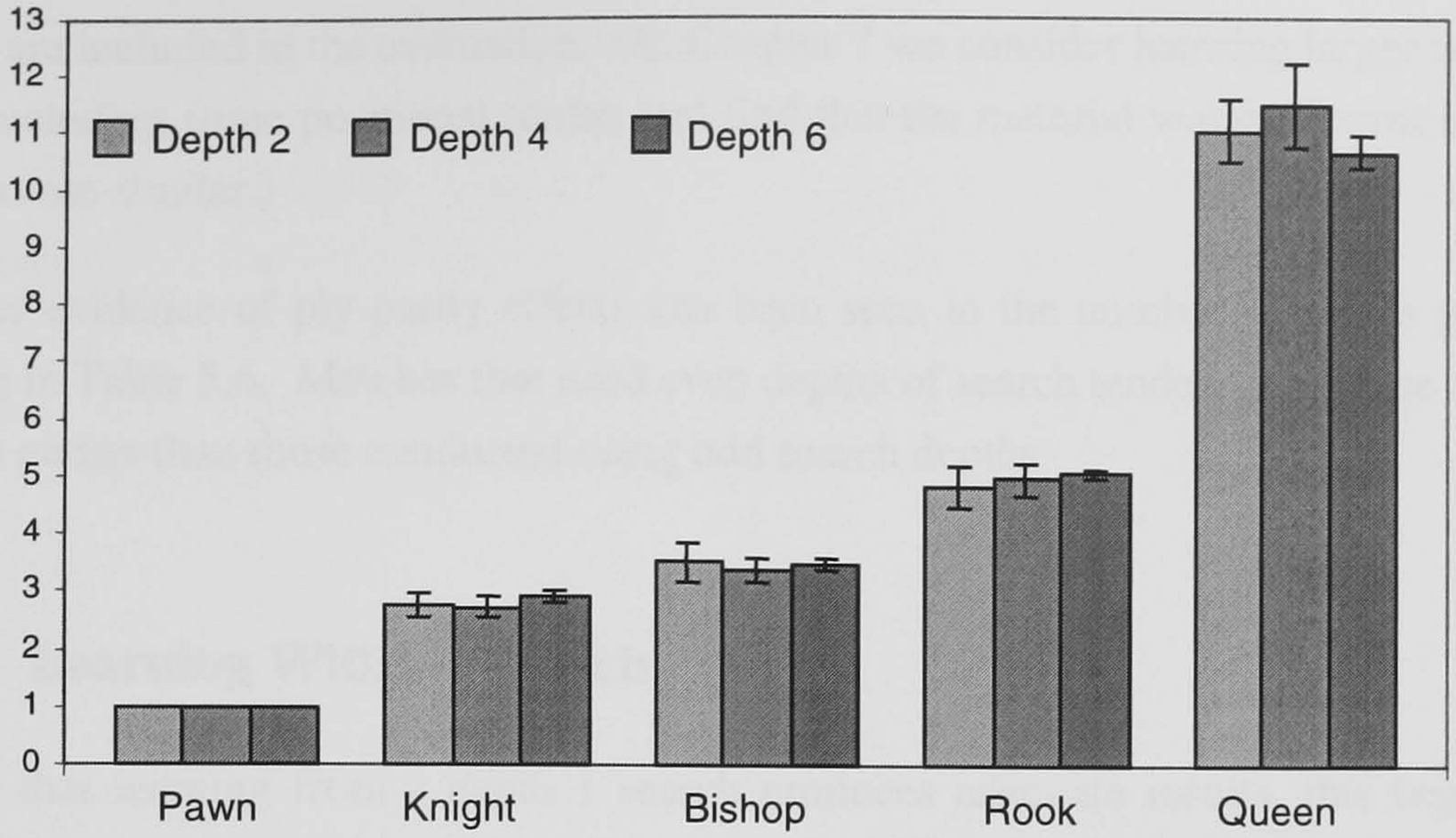

Figure 5.5: Learnt piece values from depths 2,4,6.

In Figures 5.4 and 5.5 the standard deviation from the different random seeds is shown as a vertical line embedded in the top of each bar. The Figures show that the learnt values vary a little from run to run, and with search depth, but that the average value for each type of piece is quite close to the standard values often told to beginners.

5.4.1 Matches at various depths

As in the basic experiment we ran matches between the learnt values and the 1:3:3:5:9 values to determine whether the learnt values are better, worse, or equivalent to the human values as told to beginners. The matches were conducted using a fixed search depth equal to that used to determine the learnt values (i.e. the depth 4 matches used the values learnt at depth 4 in conjunction with a search depth of 4-ply for both sides).

Table 5.6 shows that the learnt values are clearly superior at all depths.

Depth	*Win*	*Loss*	*Draw*	*Score*
1	1,003	828	169	54%
2	1,052	899	49	54%
3	1,028	644	328	60%
4	1,117	777	106	59%
5	1,053	552	395	63%
6	1,147	665	188	62%

Table 5.6: Match results vs. 'standard' at various depths.

This result has to be interpreted with the caveat that the superiority is only shown under the particular conditions of this experiment. The program evaluated positions using material alone, and the optimum material values might be different if positional

terms are included in the evaluation. (In Chapter 7 we consider learning larger weight sets, including some positional terms, and find that the material weights learnt in this context are similar.)

Further evidence of ply-parity effects can been seen in the number of drawn games shown in Table 5.6. Matches that used even depths of search tended to produce fewer drawn games than those conducted using odd search depths.

5.5 Learning Without Search

Given that learning from a depth 1 search produces adequate results, this begs the question as to whether search is needed at all. In order to demonstrate that some level of search is required for successful learning, we conducted experiments whereby the program attempted to learn from games played with no search at all, and with the moves chosen entirely at random. These runs failed to learn any useful values, even after 10,000 games. Given that there was no feedback of any kind from the piece weights into the move selection, this result is not surprising. Figure 5.6 presents the weight traces from one such run.

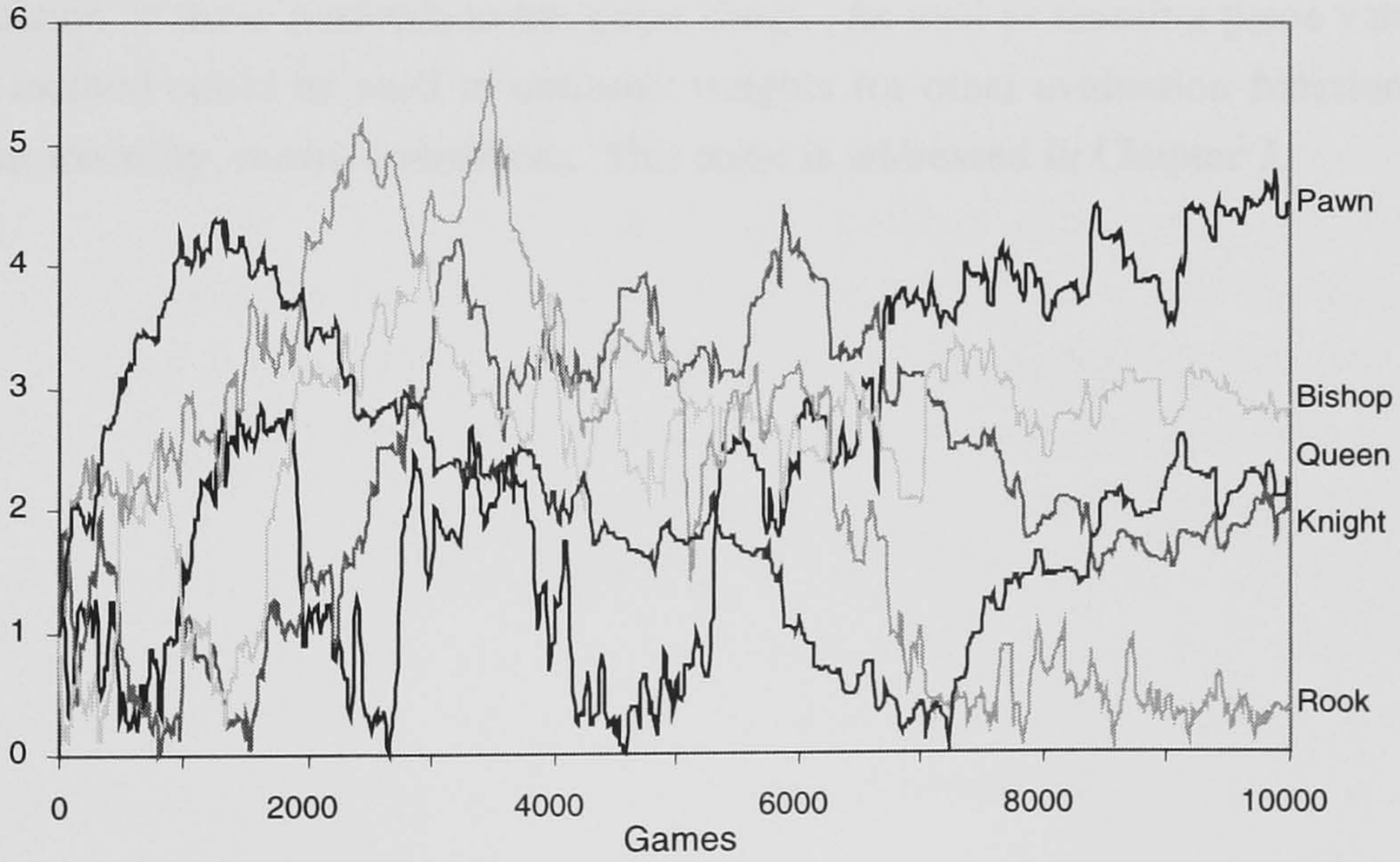

Figure 5.6: Failure to learn from entirely random play.

Beal and Smith (1994) observe that random *play* is very different from random *evaluations*, and demonstrate the counter intuitive result that a deep minimax search on random evaluations produces play vastly superior to the random selection of moves. A lightly edited version of this paper is contained in Appendix E.

We also investigated search regimes that invested less computational effort than the depth 1 plus quiescence search described above, namely a quiescence-only search and one that used 1 ply search without quiescence. Both the quiescence-only and 1-ply-no-quiescence runs learnt much more slowly and erratically than depth 1 plus quiescence, and had not approached stable values by the end of the runs. We interpret these results as being due to the quality of play being too poor to inform the learning. When an advantage was obtained by one side, the subsequent play was not good enough to consistently convert that advantage into a win. Inspection of a few sample games lent support to this interpretation.

5.6 Discussion

The experiments presented in this Chapter demonstrated the use of temporal differences to learn relative piece values that are at least as good (under our test conditions) as the widely-quoted values in elementary text books. These values were successfully learnt without any domain-specific knowledge being supplied.

The same method could be applied directly to many other two-person, perfect information games, e.g. checkers and Chinese chess. Chapter 6 describes the application of these methods to the game shogi. As well as learning piece values, the same method could be used to optimise weights for other evaluation function terms, such as mobility, centre control etc. This topic is addressed in Chapter 7.

6 LEARNING IN SHOGI

The previous Chapter presented results from the application of Temporal Difference learning in the chess domain. To widen our testing beyond chess alone, we also investigated shogi - another complex game that has some similarities to chess, but also major differences that make it harder to create programs that play it. This Chapter reports on experiments to determine whether sensible values for shogi pieces can be obtained in the same manner as for chess pieces. As was the case with the chess experiments presented in Chapter 5, the learning is obtained entirely from randomised self-play, without access to any form of expert knowledge. The piece values are used in a simple search program that chooses shogi moves from a shallow lookahead, using pieces values to evaluate the leaves, with a random tie-break at the top level. Temporal difference learning is used to adjust the piece values over the course of a series of games. The resulting learnt piece values were tested in matches against hand-crafted values, including a set of values used by the 1997 World Computer Shogi Champion.

6.1 Shogi: One Step Beyond

Shogi is a traditional Japanese board game, and considered by many researchers to be the next challenge beyond chess for computer game-playing (Matsubara, Iida and Grimbergen, 1996). In Japan the game has a very high profile, with top shogi professionals being regarded as national celebrities. Shogi belongs to the same family of games as western chess and Chinese-chess (Xiangqi), the most noticeable differences being that in shogi captured pieces are not eliminated from the game, but kept *in hand* by the capturing player, and may later be returned (*dropped*) on almost any vacant square. A further significant difference from western chess is that all pieces apart from the king and gold are eligible for promotion once they reach the *promotion zone* (the last three ranks of the board). Pawns, lances, knights and silvers may all promote to golds upon entering the promotion zone, whereas rooks and bishops promote to more powerful pieces. An introduction to the rules of shogi and some elementary strategic advice is given by Fairbairn (1989)

The re-introduction of captured pieces in shogi means that there is no loss of material as the game progresses. This makes the division of the game into stages (e.g. opening, middlegame and endgame) less feasible, and also means that it is extremely rare for a game not to end with a win for one side. The introduction of piece drops

also causes the game tree to have a much larger branching factor than chess, making the game much less amenable to full-width searching techniques.

The final frontier for computer game playing programs is likely to be *Go* (also known as *Wei chi* in Chinese and *Badduk* in Korean). With its average branching factor in excess of 200, and games typically taking over 200 moves for completion, the size of its search space is significantly greater than either chess or shogi. But what makes Go such a challenge is the lack of a natural evaluation function, such as material. Despite considerable effort in the field (Müller 1999), the best Go playing program has only reached the strength of a weak human amateur.

6.2 The Relative Value of Shogi Pieces

Sensible values for chess pieces are fairly widely known. However, the choice of suitable values for shogi pieces is a problem for game programmers, because shogi experts prefer not to allocate values to the pieces. Unlike the situation in chess, there is no generally-agreed standardised set of values for shogi pieces that is given as advice for beginners. Hence shogi programmers have more need for machine learning to generate material values for use in evaluation functions.

Note that in Shogi, unlike chess, the value of a piece is not the same as the change in the material balance when a piece is captured. For example, when capturing an opponent's promoted rook the change in material balance needs to take into account both the loss of the promoted rook to the opponent, and also the gaining of a rook in hand for the capturing side. The adjustable weight associated with rooks represents the value of a rook, and does not represent the effect of a rook capture, which would change the material balance by twice that value.

The focus of this work is on learning from self-play alone, with no knowledge input. This is of greater potential value for problems where existing expertise is not available, or where the computer program may be able to go beyond the level of existing knowledge.

The experiments presented in this Chapter were designed to discover whether the same TD technique that had performed well in the chess domain would perform as satisfactorily in the more demanding domain of shogi, and whether it would yield sensible values for shogi pieces.

6.3 The Shogi-Playing Search Engine

The shogi experiments used a search engine derived from the chess platform described in Chapter 4. This included a fixed-depth, iteratively-deepened full-width search, with a captures-and-promotions-only quiescence search at the full-width leaves. To prevent undue search effort being expended in the quiescence search, it was limited to a depth of eight plies. (We performed some test runs with an unlimited quiescence search, and obtained essentially identical results, but at much greater computational cost.) As in the chess engine described in Chapter 4, null-move pruning was used in the main search to reduce the size of the search tree, and the search was once again made more efficient by the use of a transposition table. A similar evaluation function to that used in the basic chess experiments described in Chapter 5 was used, and consisted of the material score only. To ensure variations in the games the move choice at the root was made randomly from the best of the materially-equal moves.

The thirteen piece values being learnt (seven main piece types and six promoted types) were used by the evaluation function. The *material balance* for a position was calculated as the sum of all the values of the side to move's pieces (including pieces in hand), minus the sum of all the values of the opponent's pieces. It would be possible to learn separate values for pieces on the board and those in hand, and further discrimination would be possible depending on the quantity of each piece type held in hand. In order to simplify the experiments a single value was used for each piece type both on the board and in hand.

6.4 Applying Temporal Difference Learning to Shogi

The experiments were designed to test the TD methods described in Chapter 3 to the task of learning suitable piece values for a shogi-playing program. Many learning runs were performed to explore the behaviour of the TD(λ) method using a variety of search depths. Suitable values for the learning rate, and values for λ, were determined by some preliminary test runs. All games were played using the shogi engine described in section 6.3, with a main search varying in depth between one and four plies. To prevent the same games from being repeated, the move lists were randomised, resulting in a random choice being made from all tactically equal moves. This has the added benefit of ensuring a wide range of different types of position are encountered.

During each game a record was kept of the value returned by the search after each move, and the corresponding principal position. These values are converted into prediction probabilities by the squashing function given in equation (3.4), and then equations (3.1) and (5.1) are used to determine adjustments to the weights at the end of each game.

Once again the experiments learn values for the pieces entirely from randomised self-play. This method has the advantage that it requires no play against well informed opponents, nor is there any need for games played by experts to be supplied. The piece weights are learnt 'from scratch', and do not need to initialised to sensible values. The only shogi-specific knowledge provided is the rules of the game. Whilst each learning run consists of several thousand games, this represents a relatively short amount of machine time, and the entire run can be completed without any external interaction.

In the experiments reported here we used a value for λ of 0.95, and a variable value of α that decreased during each learning run, from 0.05 to 0.002. At the start of each run all weights were initialised to 1, so that no game-specific knowledge was being provided via the initial weights.

6.5 Results from Learning

We present results from five separate learning runs of 6,000 games each. The learning runs were identical except that a different random number seed was used in each one, ensuring that completely different games were played in each. We shall refer to these learning runs as *Run A* through *Run E.*

6.5.1 Weight traces

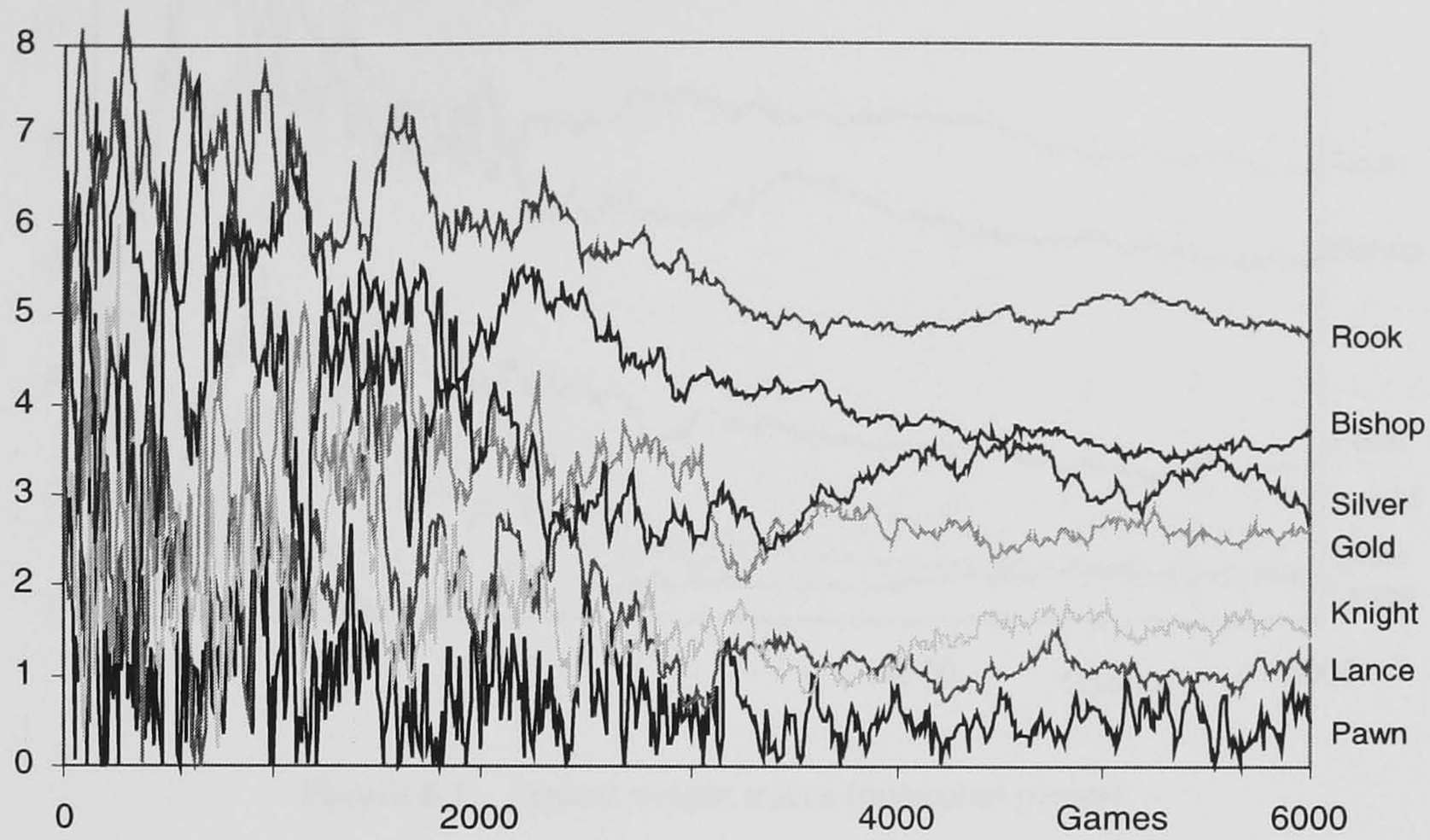

Figure 6.1: Typical weight traces (main pieces).

Figure 6.1 shows the weight traces for un-promoted pieces for a typical learning run (Run C) of 6,000 games. A decaying learning rate was used for the first half of the run, decreasing from 0.05 to 0.002. Once the learning rate reached 0.002, it remained constant for the remainder of the run. Very similar results were achieved using a fixed learning rate of 0.002, but the runs required more games to achieve stable values.

From Figure 6.1 we can see that the relative ordering of the main pieces has been decided after about 4,000 games, and that pieces remain in that relative order for the remainder of the run. During the last 2,000 games there is still considerable drift in the values. Some random drift is to be expected as a result of the random component included in the move choice. We averaged the values over the last 2,000 games in order to obtain values for testing against other weight sets.

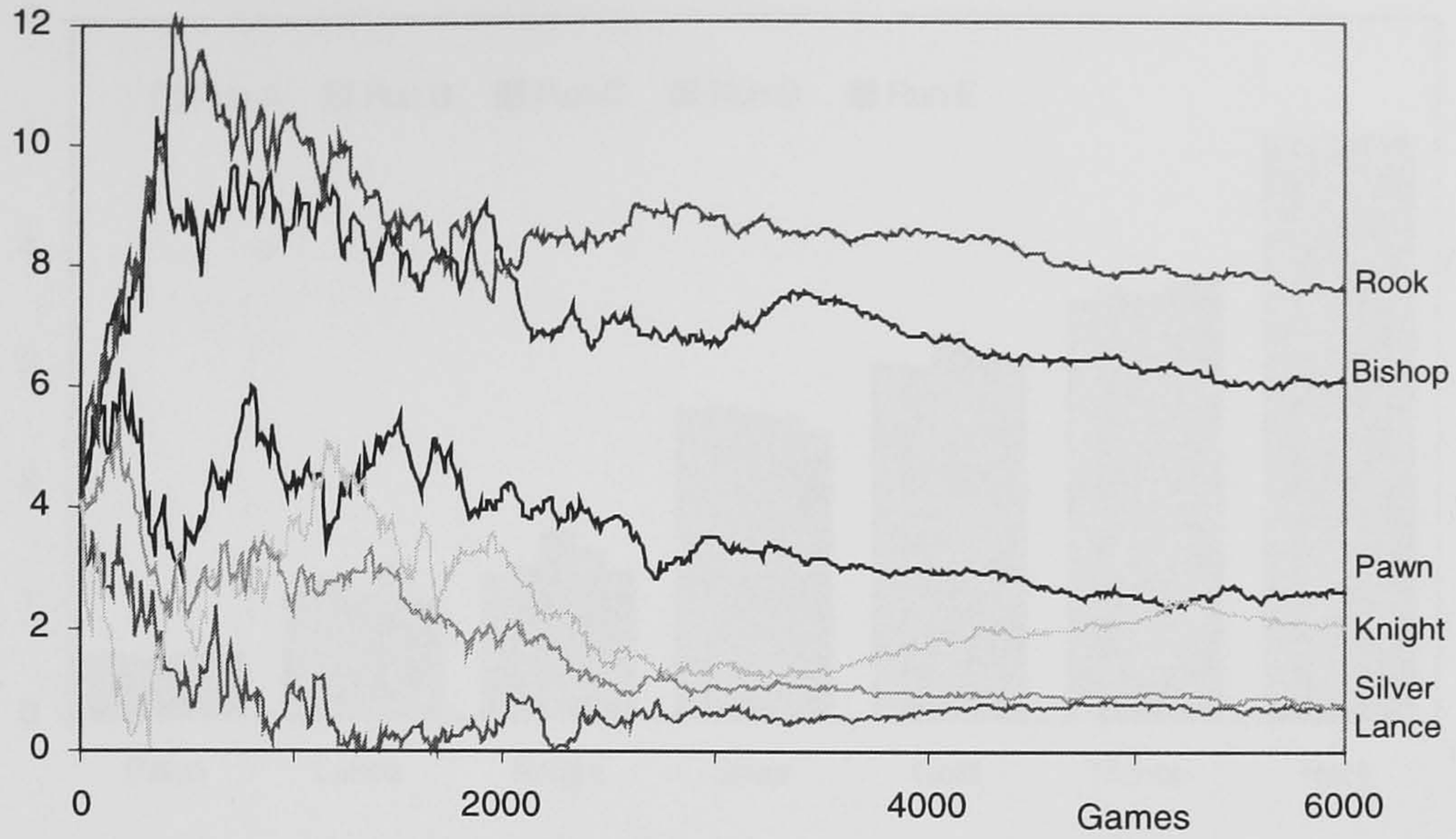

Figure 6.2: Typical weight traces (promoted pieces).

Figure 6.2 shows the weight traces for promoted pieces from Run C. Comparing Figures 6.1 and 6.2 we can see that the promoted piece traces appear more stable than the main piece traces. This is because adjustments to the promoted piece types occur less frequently during the course of a game. Indeed, some games may not contain a single instance of a given promoted piece type. There is no trace for gold, because they do not promote.

6.5.2 Main piece values

Figure 6.3 shows relative values for the seven main piece types, from each of the five learning runs. To avoid fluctuations in the weights due to noise from the stochastic nature of the game-playing process, these values represent the average over the last 2,000 games in each of the five runs.

It is the *relative* values of the pieces that governs move selection, not the *absolute* values. Normalising the values so that rook=5 enables us to readily compare the values from the five runs[1].

---

[1] In chess, one often refers to the values of pieces in terms of pawns, e.g. "A knight is worth three pawns". In shogi, there is no such commonly used metric. However, in certain rare situations (Fairbairn 1989) the rules of shogi state that rooks are to be scored as five points each, and all other pieces as one point each. We chose the five-point rook score as our reference value for normalising.

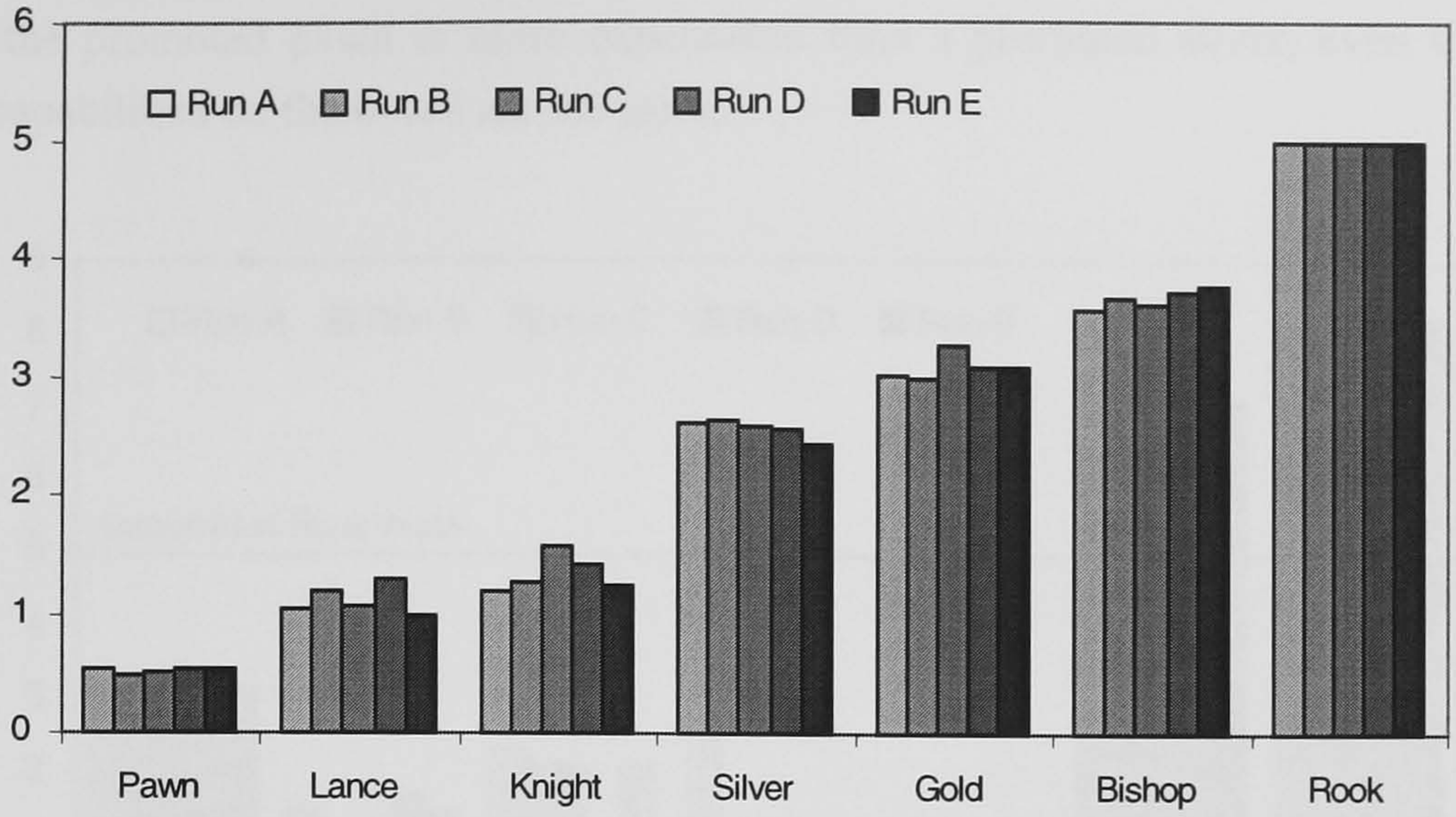

Figure 6.3: Normalised learnt values for 5 runs (main pieces).

From Figure 6.3 we can see that each of the five runs has learnt the same ordering of the pieces (pawn, lance, knight, silver, gold, bishop, rook). In addition, the relative magnitude of the learnt values is fairly consistent across the five runs.

6.5.3 Promoted piece values

Figure 6.4 shows the normalised relative values for the six promoted piece types (golds do not promote). The values for promoted bishops and rooks are substantially more that for their un-promoted counterparts.

When promoted, pawns, lances, knights and silvers all promote to piece types that move in exactly the same way as a gold. Despite this it can be seen from Figure 5 that the learnt values for these promoted types differ considerably. This might be due to in part to low numbers of promotions in the games, which leads to higher run-to-run variance and to end-of-run values which are not yet settled.

In particular, the value learnt for promoted pawns is consistently greater than those learnt for promoted lances, knights, or silvers. This is probably partly because the act of promoting a pawn has the additional benefit of making all empty squares in that file available for the subsequent dropping of a pawn in hand. (The rules of shogi prohibit dropping a pawn into a file that already contains a friendly un-promoted pawn.)

Another issue that might affect values of promoted pieces is the value to the opponent if they are captured. For example, a promoted pawn gives the opponent only a lowly pawn in hand, whereas a captured promoted silver gives the opponent a silver in hand.

Thus the promoted pawn is more expendable than a promoted silver, even though their capabilities on the board are the same.

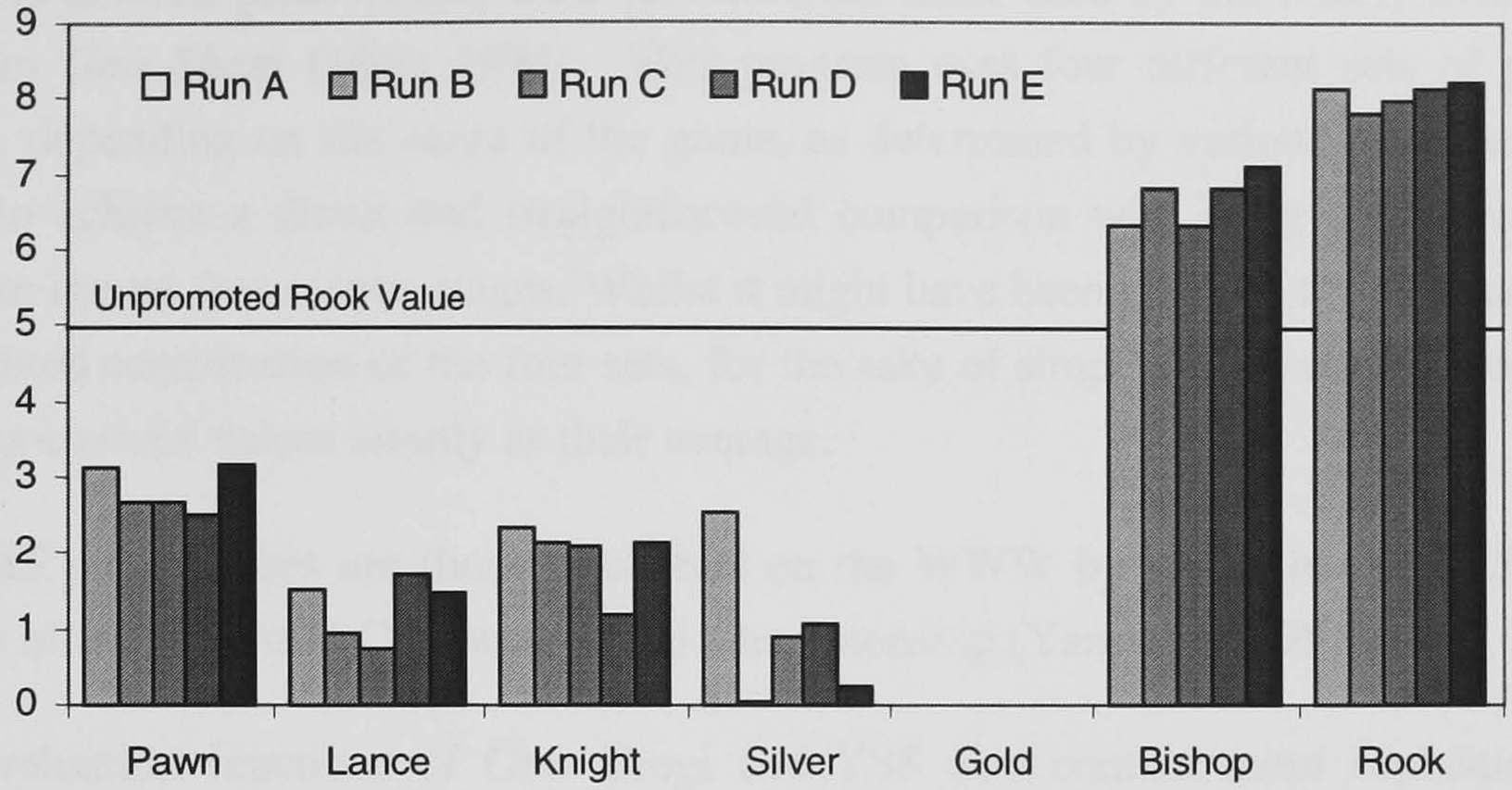

Figure 6.4: Normalised learnt values for 5 runs (promoted pieces).

6.6 Testing Learnt Values in Match Play

To test the effectiveness of the learnt values in our domain, a number of matches were played between identical search engines using various different piece values. The search engines were the same as those used for the learning experiments, but the piece weights were fixed to a given set of piece values and were not adjusted during the match.

Each match consisted of 2,000 games, alternating Black and White (*Sente* and *Gote*). Games that ended in mate were scored as 1 point for the winning side. Games that were unfinished after 600 ply (300 moves each) were scored as ½ point for each side.

We ran two set of matches. The first was effectively a mini-tournament to compare the average values from all five learning runs with values obtained from other sources. The second set compared each of the five sets of learnt values with the best of the values from other sources.

Since there is no generally-agreed set of values in shogi for comparison, the shogi learnt value set was tested in match play against three other value sets: *Beginner*; *Gnu-derived*; and *YSS*. The values used in each of these sets are presented in table form in Appendix C.

The *Beginner* piece values were decided by a shogi beginner (but experienced game programmer), guided by advice from Leggett (1966).

The *Gnu-derived* piece values were derived from those used by the widely available program *Gnu Shogi* (Mutz 1994). This program uses four different sets of piece values, depending on the stage of the game, as determined by various heuristics. In order to achieve a direct and straightforward comparison with other value sets we chose to ignore these game stages. Whilst it might have been possible for us to devise a weighted combination of the four sets, for the sake of simplicity we chose to define the *Gnu-derived* values simply as their average.

The *YSS* piece values are those published on the WWW by the author of *YSS 7.0*, winner of the 7th World Computer Shogi Championship (Yamashita 1997).

The evaluation functions of Gnu Shogi and YSS also contain more sophisticated positional terms, e.g. king safety. In both programs, piece values are fundamental and typically the largest component of the overall evaluation score for a position. (Positional factors can also reward material possession indirectly. We ignored this secondary effect for these value sets.) It is possible that optimum material values for a search using positional terms are different from those for a material-only search. However, we found in experiments in chess not yet published that the material values learnt in conjunction with positional scores for piece-square combinations were similar to those learned for material-only. We believe the optimum values for a program using sophisticated positional scores in addition to the material scores would be fairly close to our learnt values. The aim of the matches was primarily to demonstrate the adequacy of the method, rather than claim superiority of our values in all contexts.

The *Learnt* piece values are the average of the values presented in Figures 6.3 and 6.4.

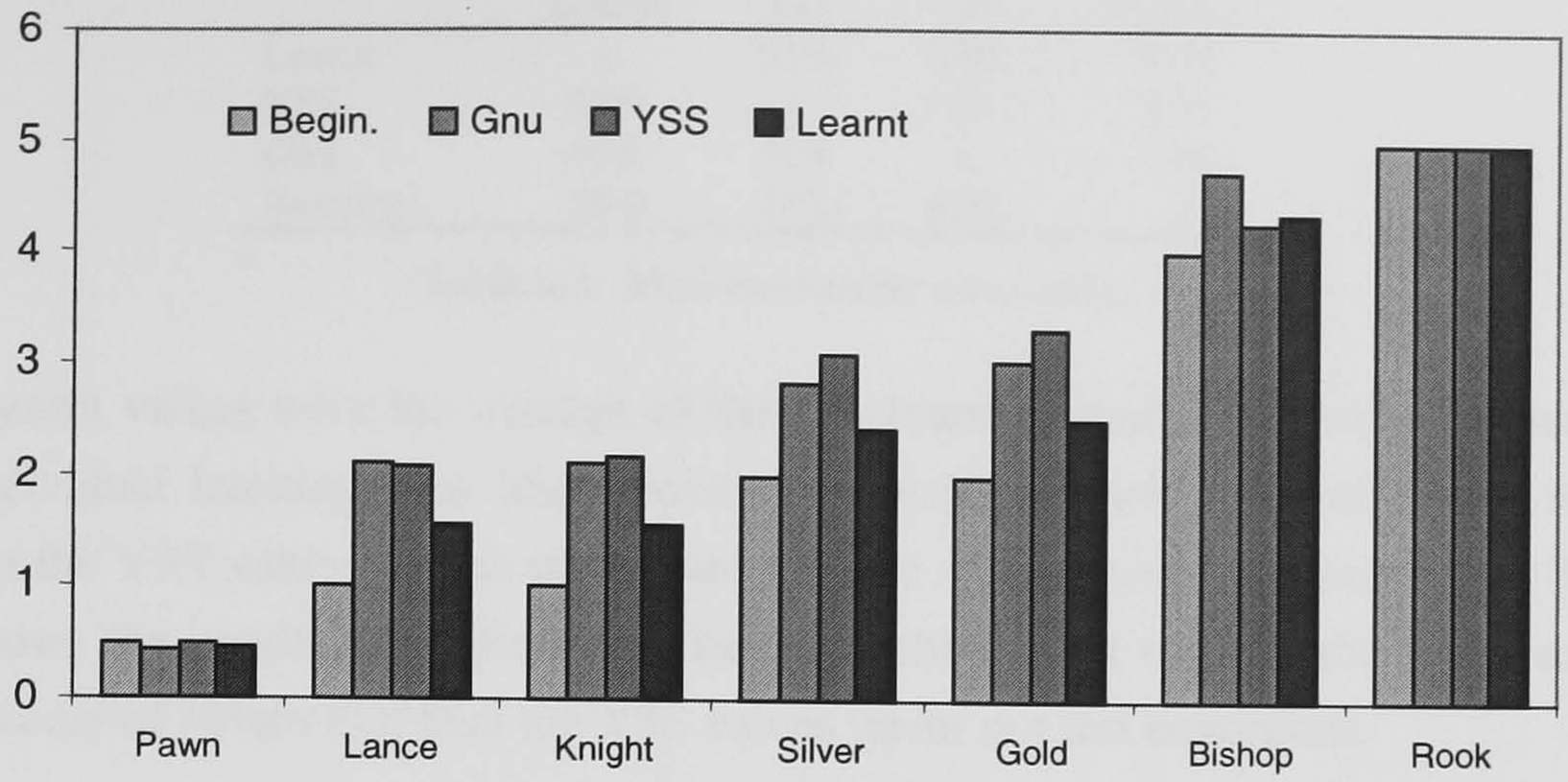

Figure 6.5: Value sets tested in match play (main pieces).

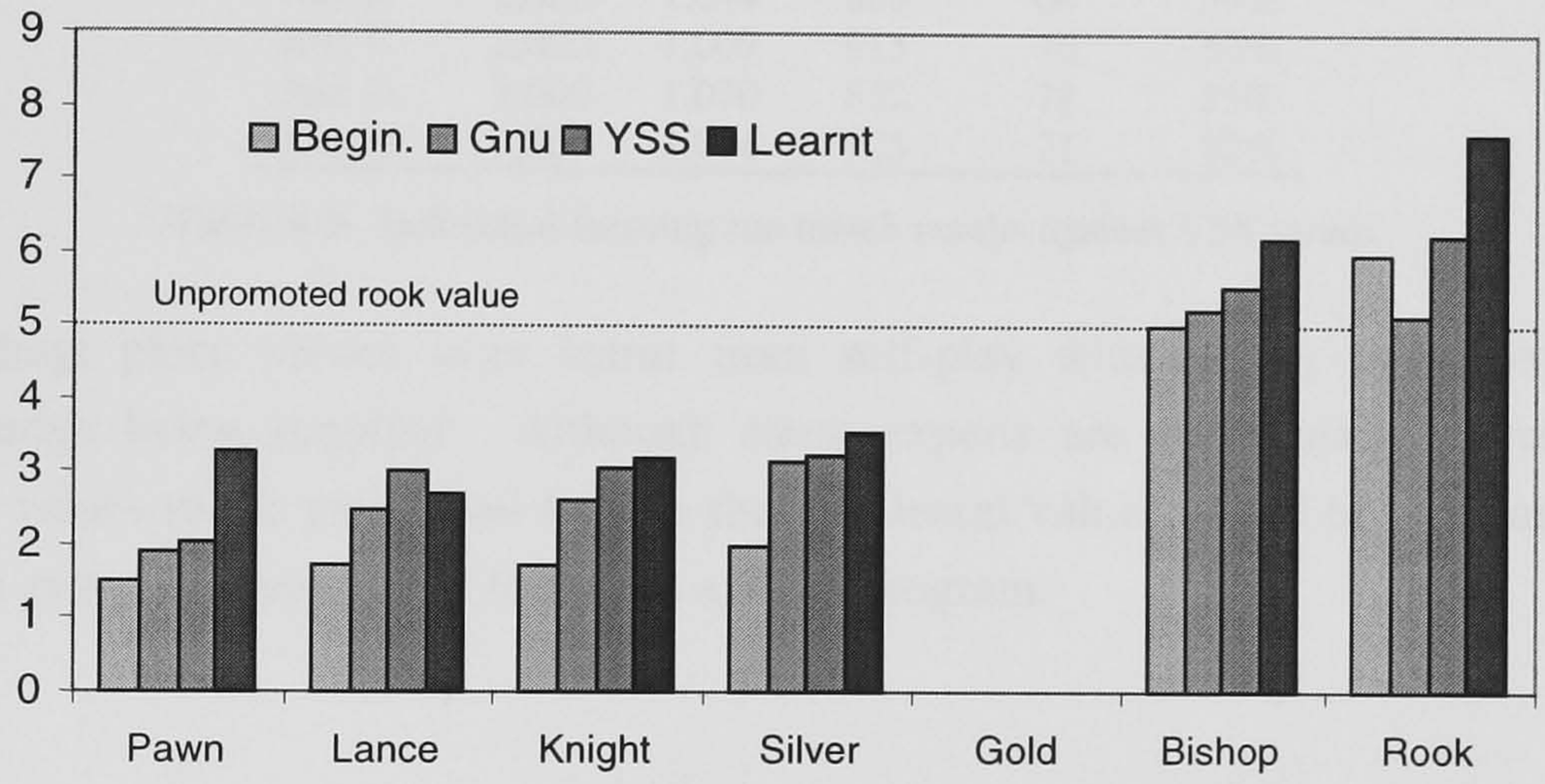

Figure 6.6: Values sets tested in match play (promoted pieces).

Figures 6.5 and 6.6 show the pieces values used in the matches, again normalised to rook=5.

Side 1		*Side 2*	*Games*	*Win*	*Loss*	*Draw*	*%*
Learnt	vs.	Beginner	2,000	1,206	718	76	62%
Learnt	vs.	Gnu	2,000	1,170	766	64	60%
Learnt	vs.	YSS	2,000	1,071	871	58	55%
YSS	vs.	Beginner	2,000	1,113	835	52	57%
YSS	vs.	Gnu	2,000	1,146	784	70	59%
Gnu	vs.	Beginner	2,000	1,018	911	71	53%

Table 6.1: Shogi match results.

Table 6.1 gives details of the matches played in the mini-tournament, and Table 6.2 shows the cross-table of results. The Learnt values performed better than any of the other value sets under our test conditions, scoring 55%, 60% and 62% against the *YSS, Gnu-derived*, and *Beginner* value sets respectively.

	Learnt	*YSS*	*Gnu*	*Beginner*
Learnt	x	55%	60%	62%
YSS	45%	x	59%	57%
Gnu	40%	41%	x	53%
Beginner	38%	43%	47%	x

Table 6.2: Mini-tournament cross-table.

The Learnt values were the average of the five learning runs. To verify that each of the individual learning runs learnt reasonable weights, each was pitted in a match against the YSS values, which performed the best of the three non-learnt sets. Table 6.3 shows the results from these matches, and shows that each of the five learning runs produced values that beat the YSS values under out test conditions.

	Games	*Win*	*Loss*	*Draw*	*Percent*
Run A	2,000	1,062	871	67	55%
Run B	2,000	1,044	888	68	54%
Run C	2,000	1,009	915	76	52%
Run D	2,000	1,070	852	78	55%
Run E	2,000	1,004	925	71	52%

Table 6.3: Individual learning run match results against YSS values.

The shogi piece values were learnt from self-play without any domain-specific knowledge being supplied. Although shogi experts are traditionally reluctant to assign values to the pieces, we believe that our learnt values would be recognised by human experts as reasonable for use in a shogi program.

6.7 Variation of Learnt Values with Search Depth

Figure 6.7 shows the average piece values learnt at each of the four depths, normalised so that rook=5. The standard deviation from the five different random seeds is shown as a vertical line embedded in the top of each bar. This Figure shows that the values learnt for the main piece types are fairly consistent across search depths, with the relative ordering of the pieces being the same in every case. The raw results used to construct this Figure are given in Appendix C.

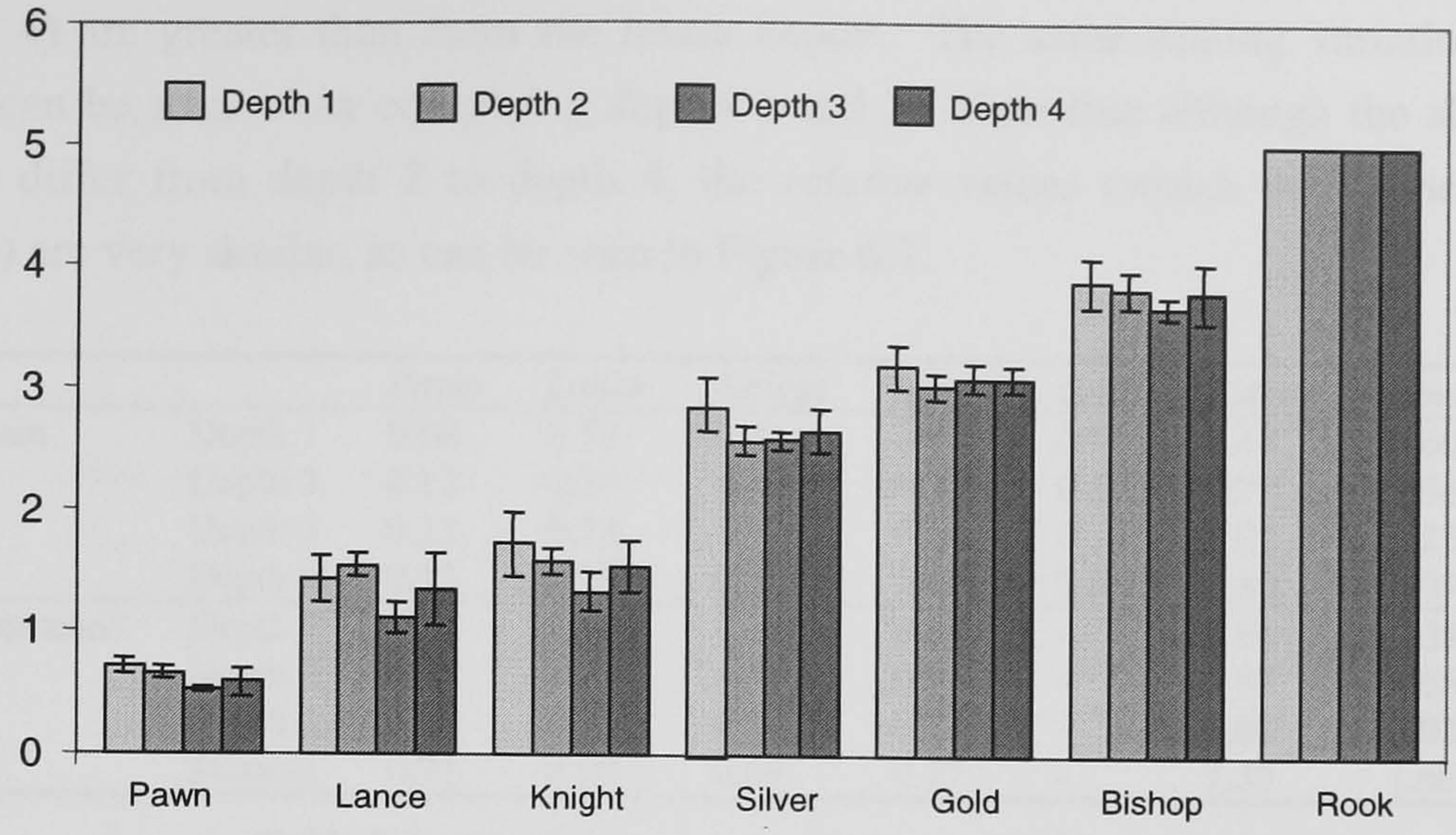

Figure 6.7: Main piece values learnt at depths 1-4, normalised to rook=5.

The search depths used for learning runs refer to the main search prior to the quiescence search. Thus 'depth 1' means 1 ply of main search followed by the quiescence search. The results show that the values are fairly consistent over search depths from depth 1 upwards.

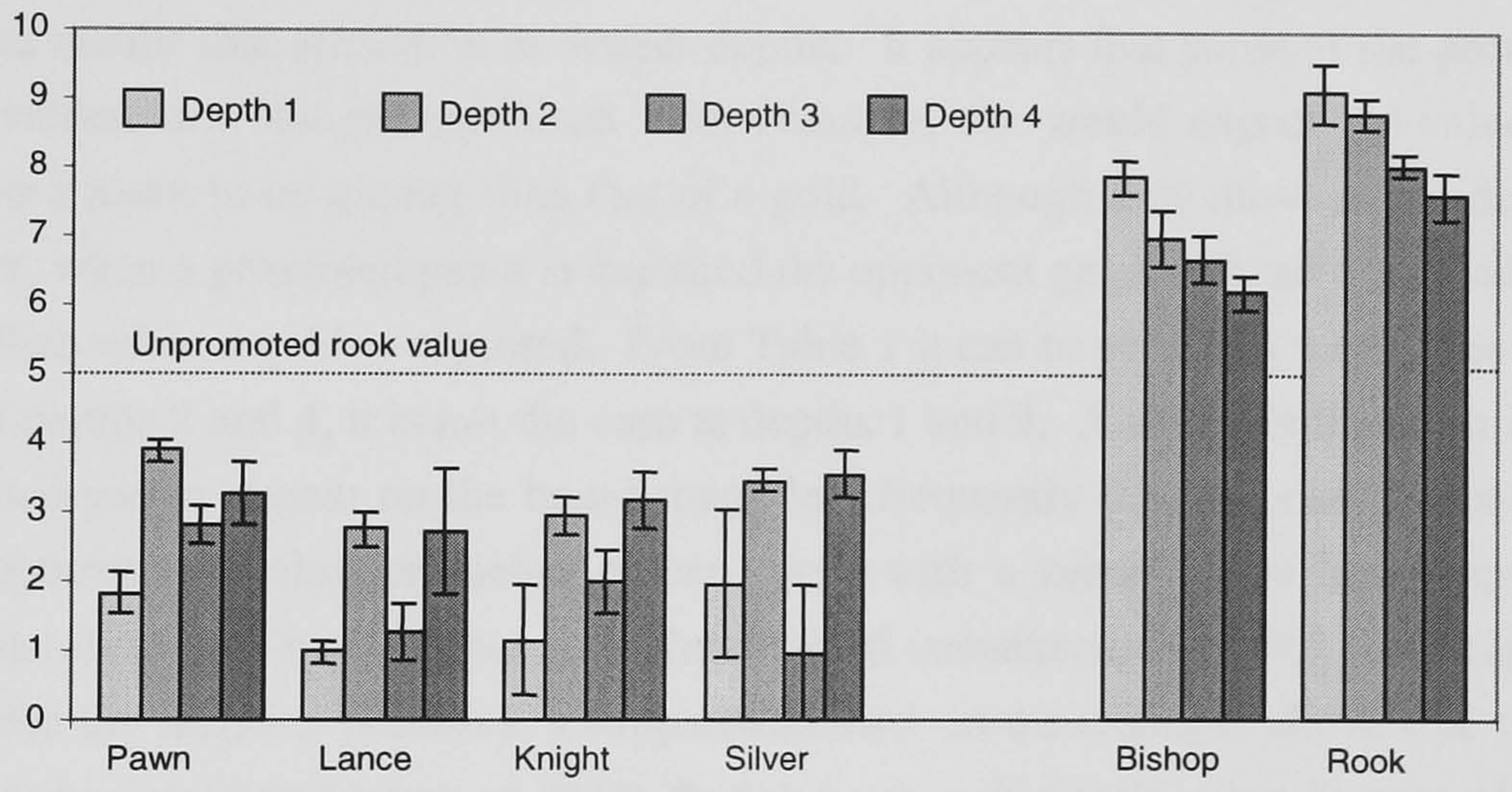

Figure 6.8: Promoted piece values learnt at depths 1-4, normalised to rook=5.

6.7.1 Scaling variation with depth

Table 6.4 gives the average piece values (before normalisation) for each of the four depths, for both the main and promoted piece types. Golds never promote, and so have no entry in the promoted section. Comparing the depth 2 values with those from depth 4, it can be seen that the *absolute* values obtained using better quality of play

(depth 4) are greater than from the lesser depth[2]. The same scaling variation with depth can be seen when comparing depths 1 and 3. Note that although the absolute values differ from depth 2 to depth 4, the *relative* values (which determine move choice) are very similar, as can be seen in Figure 6.7.

		Pawn	*Lance*	*Knight*	*Silver*	*Gold*	*Bishop*	*Rook*
Main	Depth 1	0.08	0.17	0.21	0.34	0.38	0.47	0.60
	Depth 2	0.12	0.29	0.30	0.48	0.57	0.72	0.94
	Depth 3	0.13	0.28	0.33	0.64	0.77	0.92	1.25
	Depth 4	0.13	0.29	0.34	0.57	0.67	0.82	1.09
Promoted	Depth 1	0.22	0.12	0.14	0.24	-	0.94	1.08
	Depth 2	0.73	0.52	0.55	0.65	-	1.32	1.66
	Depth 3	0.70	0.32	0.50	0.23	-	1.68	2.02
	Depth 4	0.71	0.60	0.69	0.77	-	1.35	1.66

Table 6.4: Average piece values (before normalisation).

The fact that the relative values vary little with an increase in search depth is encouraging for the use of this method by competitive shogi programs, which typically operate with a search depth greater than four plies. The computational cost of such searches makes learning runs of thousands of games at those depths infeasible, but these experiments show that shallower searches may well be able to produce results that are useful at deeper depths. It appears that some of the promoted piece values have not yet stabilised. For example, one would expect the value of a promoted pawn to be greater than that of a gold. Although they move in an identical manner, when a promoted pawn is captured the opponent gains in hand a less valuable piece than when a gold is captured. From Table 1 it can be seen that whilst this is the case at depths 2 and 4, it is not the case at depths 1 and 3. A likely explanation is that promoted pieces appear on the board much less frequently than the main piece types and captures involving promoted pieces occur with a much lower frequency than those involving the main pieces (it is the material imbalances resulting from captures that drive the learning process). Comparisons such as the example above (the choice of capturing promoted pawn or gold) do not occur sufficiently often in runs of 6000 games for the learning process to come to an informed decision. It is precisely because such positions occur so infrequently that the slight inconsistencies in values of promoted have only a minor effect on the match results presented below, in which the values of the main piece types play a dominant role.

---

[2] We do not compare values from odd and even depths as it is well known that search evaluations oscillate with odd and even depths and this effect interacts with the scaling variation.

6.7.2 Match results at various depths

The value sets *Beginner*, *Gnu* and *YSS* and the format of the matches were as described in section 6.6.

The matches were conducted using a fixed search depth equal to that used to determine the learnt values (i.e. the depth 4 matches used the values learnt at depth 4 in conjunction with a search depth of 4-ply plus quiescence for both sides).

	YSS	*Gnu*	*Beginner*
Depth 1	51%	52%	55%
Depth 2	54%	59%	62%
Depth 3	58%	64%	76%
Depth 4	55%	60%	66%

Table 6.5: Match results from depths 1-4

The results of the matches played are given in Table 6.5. Full details of the matches can be found in Appendix C. The learnt values from depths 1-4 consistently performed better than any of the other value sets under our test conditions, scoring 51%, 54% and 58% and 55% against the *YSS* value set, which was the best of the opponent value sets. The *YSS* and *Gnu* value sets were selected by their authors for use in their particular programs, yet are being tested in the context of our search regime. Thus these results indicate the adequacy of the learnt values, rather than superiority over the other values under all conditions.

6.8 Learning Without Search

Shogi is a game domain, like chess, for which it appears that a significant amount of computational effort must be invested in tactical search in order to achieve high levels of play (Rollason 1999). Despite the fact that deep searches are required for highly-skilled play, our experiments show that relatively shallow searches, even as shallow as one-ply full-width plus quiescence, are sufficient for learning good material values. As in the chess domain, this invites the question as to whether or not search is required at all for successful learning, the implication being that if search is not required, the computational cost of the learning sequences would be greatly reduced.

In Chapter 5.5 we showed that in the chess domain there was a minimum level of search (in that case one-ply plus quiescence) that was required for effective learning. Similar experiments were conducted in shogi. We performed learning runs that used no search at all, a quiescence-only search, and one that used 1 ply search without

quiescence. As with chess, we found that random move selection failed to produce any sensible learning, producing weight traces similar in nature to those shown in Figure 5.4.. The quiescence-only and 1-ply-no-quiescence runs learnt slowly and erratically, and had not approached stable values by the end of the runs. We conclude from these experiments that for shogi, as for chess, there is a minimum quality of play that is required to inform the learning process

6.9 Discussion

This Chapter described the application to shogi of the TD learning for minimax searches described in Chapter 3. The shogi piece values were learnt from self-play without any domain-specific knowledge being supplied. Although shogi experts are traditionally reluctant to assign values to the pieces, we believe that our learnt values would be recognised by human experts as reasonable for use in a shogi program. The values learnt using various depths of search all performed well in matches under our test conditions, and the consistency of the relative values across the search depths indicates that a one-ply plus quiescence search is already sufficient to learn reasonable values. This is encouraging for the potential application of this method to the learning of weights for use by deep-searching, high-performance programs. It indicates that the learning process can use much shallower, faster searches than the playing program, and thus obtain values from large numbers of training games in a reasonable time.

It should be noted that these experiments have learnt material values within a material-only evaluation function. We would expect the material values learnt to be somewhat different if the evaluation function included positional scoring terms. Also, our results were obtained using a specific set of search parameters (selectivity, quiescence details, etc). These could influence the optimum values, although we would expect changes to search parameters to have less effect on learnt values than additional evaluation terms. The method could be applied to any other set of search parameters, and other search engines. It is also applicable to learning an appropriate weight for positional evaluation terms, and we expect it to be useful in learning weights for more sophisticated evaluation functions in both chess and shogi.

Significantly, the shogi experiment was in a domain where human expertise was unable to provide the knowledge required. In such domains, computer methods that learn from their own experiences are highly desirable. The values learnt in the shogi

experiment are already of some interest to commercial shogi programmers (Rollason 1999).

7 LEARNING MORE COMPLEX WEIGHT SETS

In this Chapter we consider the application of our methods to more complex weight sets, including one used by a high-performance competitive program. Chapter 5 reported results from programs that use material piece counts, plus a random tie-breaker, to choose moves. This achieves a low level of play compared with expert humans. (Nevertheless, as can be seen from Figure 5.3, the learnt chess values correspond fairly well with expert human judgements.) Performance programs typically have far more evaluation terms, incorporating a variety of positional terms, each requiring an appropriate weight to be determined.

7.1 Weights for Piece-Square Tables

In many games the value of a piece can vary according its location on the board. In games such as chess and shogi, control of the centre of the board is important, whereas in games like go and othello, the corners of the board have increased value. A standard component of typical chess programs is piece-square tables. In the piece-square tables a separate adjustment weight can be given for each square that a piece might be on. In order to test TD learning on larger weight-sets, we ran experiments to learn piece-square weights.

Of the many possible 'positional' evaluation features, we chose piece-square tables as being one of the most domain-independent. Other positional evaluation features for chess, such as pawn structure or king-safety, are less easily applicable to other game domains. Piece-square tables have the additional attraction of allowing the relatively simple introduction of a large number of new weights to the learning process, allowing us to verify that our learning methods are able to cope with significant numbers of weights without being swamped.

Learning a full set of piece-square values results in the addition of 64 new weights for each of knight, bishop, rook and queen, and 48 new weights for pawns (which never occupy ranks 1 or 8). This increases the number of adjustable weights to be learnt from 5 to 309.

As intermediate steps between learning a small weight set (piece values alone) and sets as large as 309 weights (piece-square tables), we tried sets of 27 weights (pawn ranks plus piece centrality 'rings', i.e. radial distance from the centre), and 157

weights ('half-boards'). Half-board weight sets were created by reflecting the board so that squares in files *a-d* shared a weight with the mirrored square in files *e-h*, resulting in only 32 weights per piece type. TD learning produced successful weight values for all these weight sets. As might be expected, the half-board and full-board weight sets produced the best playing performance.

7.2 Weights for Pawn Advancement and Piece Centrality

7.2.1 Weights for pawn ranks

Many games such as chess, draughts and shogi include rules allowing for the promotion of pieces to more powerful piece types once they have advanced sufficiently far up the board. In such games, advancing pieces eligible for promotion towards their 'promotion zone' can be a powerful strategy. As an initial step beyond material only learning, we learnt values rewarding (or punishing) pawn advancement in chess. In this simple experiment we ignored files, and adjusted a single weight for every rank it was possible for a pawn to be on (ranks 2-7). This resulted in 6 pawn-rank weights being learnt, in addition to the 5 material weights, for a total of 11 adjustable weights.

7	7	7	7	7	7	7	7
6	6	6	6	6	6	6	6
5	5	5	5	5	5	5	5
4	4	4	4	4	4	4	4
3	3	3	3	3	3	3	3
2	2	2	2	2	2	2	2

Figure 7.1: Indexing for the pawn weights.

Figure 7.1 shows the indexing for the pawn advancement weights, using the in the range 2-7 (ranks 1 and 8 never contain pawns). As might be expected, the program quickly learnt that advancing pawns up the board was advantageous. Of course, the primary reason that pawn advancement is beneficial is the possibility of promotion to a highly valued queen once the final rank is reached.

7.2.2 Weights for piece centrality

As a simple measure of centrality, we divided the board into four 'rings' as shown in Figure 7.2. Four separate weights representing centrality were maintained for each of the piece types: knight, bishop, rook and queen, resulting in 16 additional weights. This results in a total of 27 weights, (5 piece values, 6 pawn ranks, 16 piece centrality weights)[1].

1	1	1	1	1	1	1	1
1	2	2	2	2	2	2	1
1	2	3	3	3	3	2	1
1	2	3	4	4	3	2	1
1	2	3	4	4	3	2	1
1	2	3	3	3	3	2	1
1	2	2	2	2	3	2	1
1	1	1	1	1	1	1	1

Figure 7.2: Indexing for the piece centrality weights.

We conducted 6 learning runs at search depth 4, using different random number seeds. Each run consisted of 10,000 games. The search engine was that used for the experiments in Chapter 5, except for inclusion of the additional terms in the evaluation function.

	Pawn	*Knight*	*Bishop*	*Rook*	*Queen*	
Piece values	0.83	1.58	1.72	2.43	4.52	
Pawn ranks	*Rank 2*	*Rank 3*	*Rank 4*	*Rank 5*	*Rank 6*	*Rank 7*
Pawn	-0.33	-0.31	-0.25	-0.17	0.25	0.63
Piece centrality	*Ring 1*	*Ring 2*	*Ring 3*	*Ring 4*		
Knight	-0.17	0.15	0.21	0.40		
Bishop	-0.04	0.25	0.25	0.27		
Rook	0.15	0.40	0.46	0.47		
Queen	0.58	0.75	0.83	0.78		

Table 7.1: The average weights learnt in the pawn ranks + piece centrality runs.

Table 7.1 shows the average weights learnt over the six runs. As they stand, these raw weights are hard to compare with other weight sets, especially as one needs to combine the weights for value of a piece and its location. One way to simplify this is to add the piece value weights into the positional weights, resulting in a single term

---

[1] It is a well known feature of chess that king centrality is detrimental in the opening and middlegame, but beneficial in the endgame. The evaluation function used in these experiments does not distinguish between stages of the game, and so no attempt was made to learn a weight for king centrality.

for each piece location. Table 7.2 and Figure 7.3 present such values, normalised so that the value of a pawn on the 2nd rank equals 1.

Pawn ranks	*Rank 2*	*Rank 3*	*Rank 4*	*Rank 5*	*Rank 6*	*Rank 7*
Pawn	1.00	1.06	1.17	1.33	2.18	2.96
Piece centrality	*Ring 1*	*Ring 2*	*Ring 3*	*Ring 4*		
Knight	2.86	3.51	3.62	4.01		
Bishop	3.39	3.98	3.98	4.02		
Rook	5.21	5.72	5.84	5.86		
Queen	10.31	10.67	10.82	10.72		

Table 7.2: Composite values for piece locations, normalised so that a pawn on rank 2 = 1.

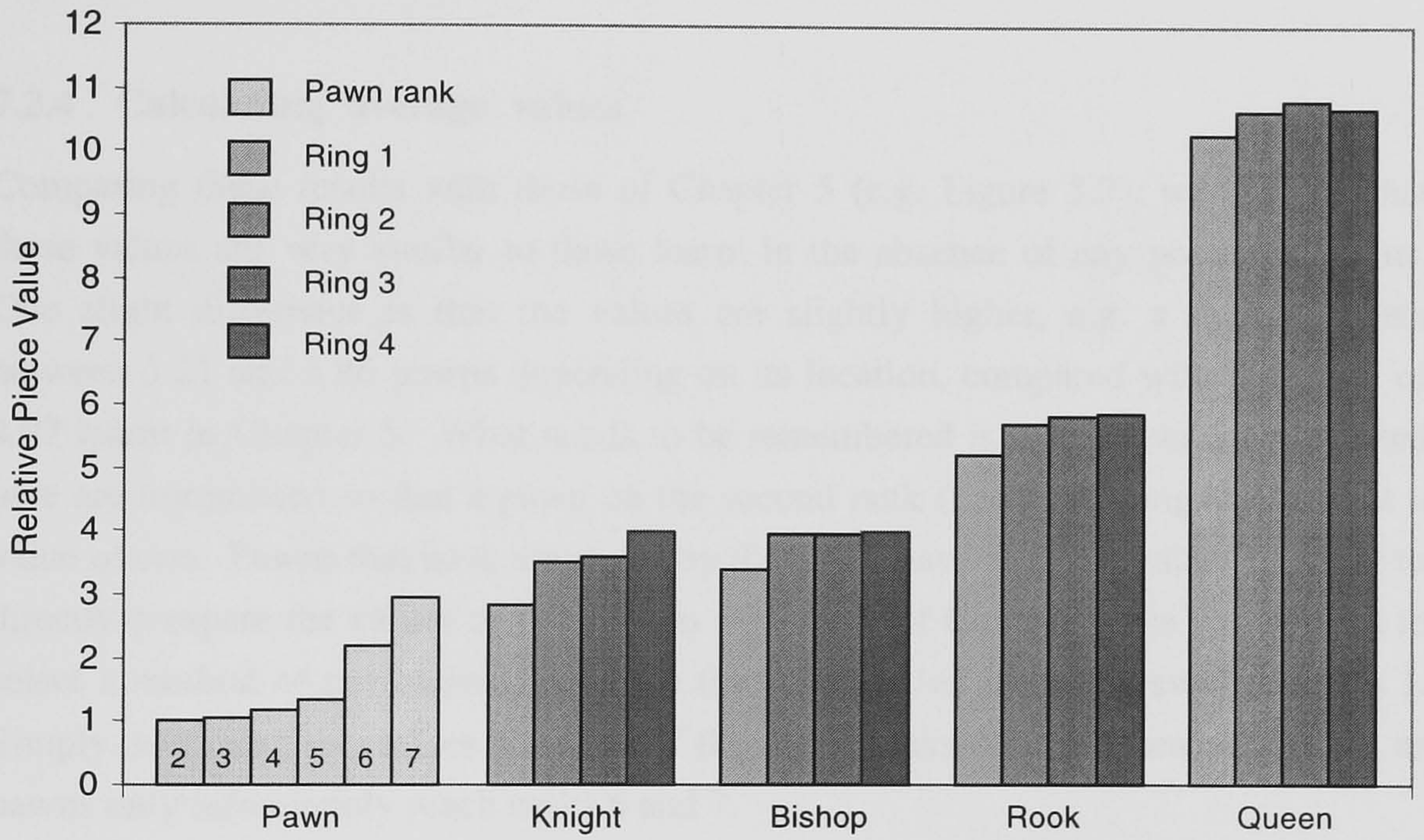

Figure 7.3: Composite pawn rank and piece centrality values, normalised so that a pawn on rank 2 = 1.

Figure 7.3 shows the composite pawn rank and piece centrality values learnt using search depth 4. Similar results were obtained at other search depths. This Figure shows clearly how pawn advancement is valued. A pawn on the seventh rank (one square away from promotion) is valued at approximately three times that of a pawn on its starting square, and more than a knight on the edge of the board. From the Figure we also can see that pieces generally perform better when stationed in the centre of the board. This is especially true for knights, whose mobility can be maximised only in the central two rings. It is interesting to note from Table 7.2 that a knight on one of the four central squares is valued slightly higher than a bishop that is not in the centre of the board.

7.2.3 Match results

The learnt weights were tested in match play against two separate opponents, *None* and *Central*. *None* consisted of the piece weights for depth 4 as presented in Chapter 5, with no additional positional terms. *Central* consisted of *None* augmented by pawn rank and piece centrality values suggested by a computer chess expert (see Appendix D).

Match	*Win*	*Loss*	*Draw*	*Score*
Learnt vs. None	1,847	106	29	94.0%
Learnt vs. Central	1,190	716	94	61.9%
Central vs. None	1,846	113	40	93.3%

Table 7.3: Match results using pawn rank and piece centrality values.

7.2.4 Calculating 'average' values

Comparing these results with those of Chapter 5 (e.g. Figure 5.3), we can see that these values are very similar to those learnt in the absence of any positional terms. One slight difference is that the values are slightly higher, e.g. a rook is worth between 5.21 and 5.86 pawns depending on its location, compared with the value of 4.97 learnt in Chapter 5. What needs to be remembered is that the values presented here are normalised so that a pawn on the second rank (i.e. its starting square) has a value of one. Pawns that have advanced up the board have a higher value. In order to directly compare the values of this section with those of Chapter 5, we would need to select a method of normalisation so that the value of 'an average pawn' is set to 1. Simply averaging the values for each of the pawn ranks would be unsatisfactory as pawns only infrequently reach ranks 6 and 7.

One way to attempt such a calculation, in order to make these results more comprehensible for human chess players, is to count how often each piece type appears in each of its specified locations during the course of the learning and compute an average value weighted by 'frequency of occurrence'.

Pawn ranks	*Rank 2*	*Rank 3*	*Rank 4*	*Rank 5*	*Rank 6*	*Rank 7*
Pawn	40.7%	27.2%	23.6%	6.5%	1.6%	0.5%
Piece centrality	*Ring 1*	*Ring 2*	*Ring 3*	*Ring 4*		
Knight	22.6%	23.8%	36.9%	16.7%		
Bishop	37.4%	30.5%	22.8%	9.3%		
Rook	65.1%	15.8%	14.7%	4.4%		
Queen	38.9%	31.3%	25.5%	4.3%		

Table 7.4: Percentage 'frequency of occurrence' for each piece location during the learning runs.

Table 7.4 presents location counts, expressed as percentages[2], for the learning runs described above. These values enable us to calculate a weighted average of the piece values in Table 7.2, resulting in 'typical' values for each piece type. Figure 7.4 presents these new piece values.

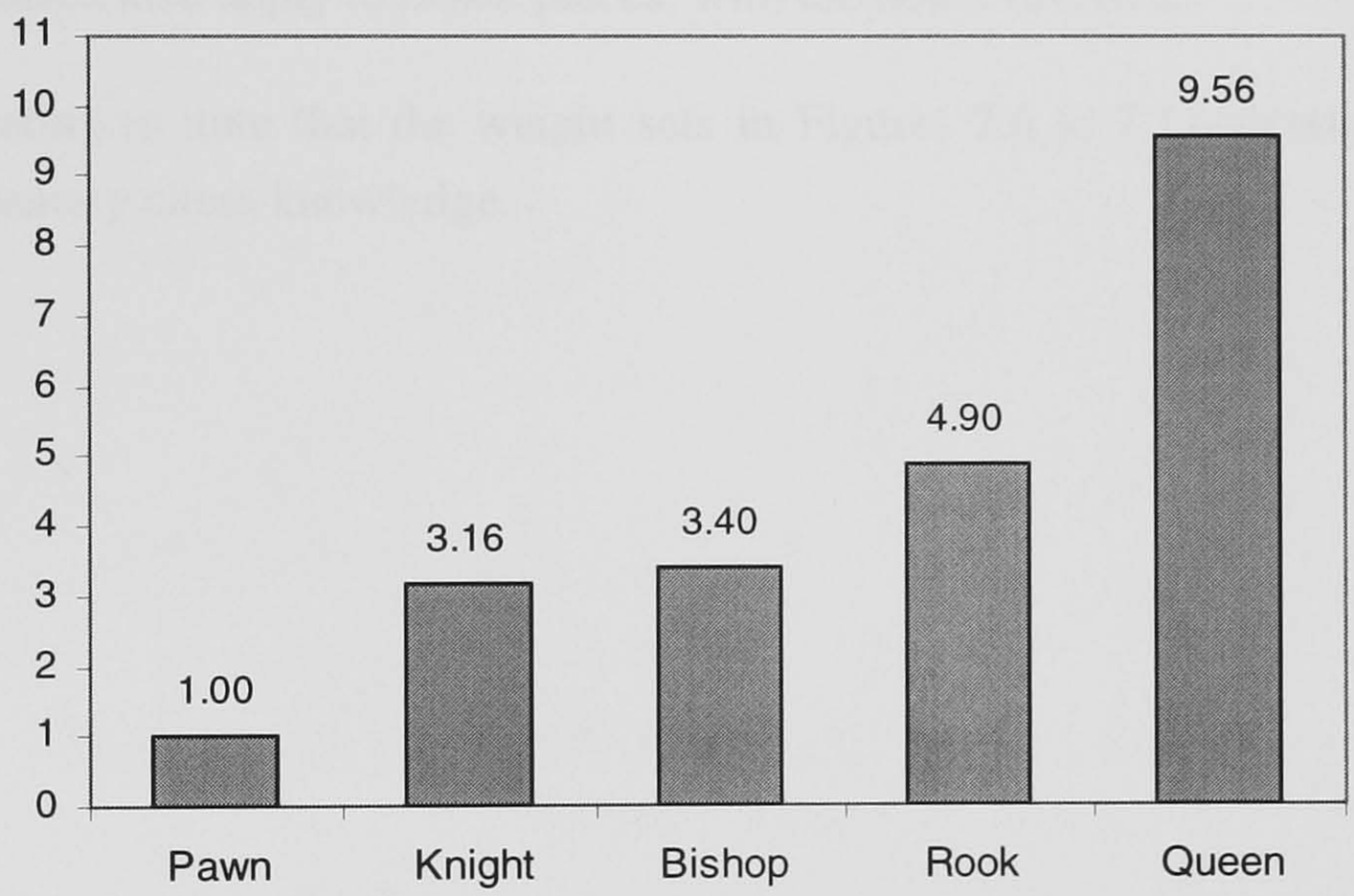

Figure 7.4: 'Typical' piece values calculated from Tables 7.2 and 7.4.

7.4 Weights for Half-board and Full-board Sets

The half-board set of weights produced similar weight patterns to the full-board weight-set, except for the queen weights, which were significantly asymmetric. To illustrate typical weight values, we present here results from the half-board weight sets. For comparison, we also show the queen weights from the full-board weight-set. The values presented are the average of 20 different runs, each consisting of 10,000 games.

---

[2] The percentage figures are rounded to one decimal place and might not sum to exactly 100%.

29	30	31	32	32	31	30	29
25	26	27	28	28	27	26	25
21	22	23	24	24	23	22	21
17	18	19	20	20	19	18	17
13	14	15	16	16	15	14	13
9	10	11	12	12	11	10	9
5	6	7	8	8	7	6	5
1	2	3	4	4	3	2	1

Figure 7.5: Indexing for the half-board weights

Figures 7.6 through 7.11 give a graphical representation of the piece-square values learnt for pawn, knight, bishop, rook and queen respectively. (The numerical details are given in Appendix D.) The Figures are presented from White's point of view, but the same values also apply to Black pieces, with the board inverted.

It is interesting to note that the weight sets in Figures 7.6 to 7.11 clearly represent some elementary chess knowledge.

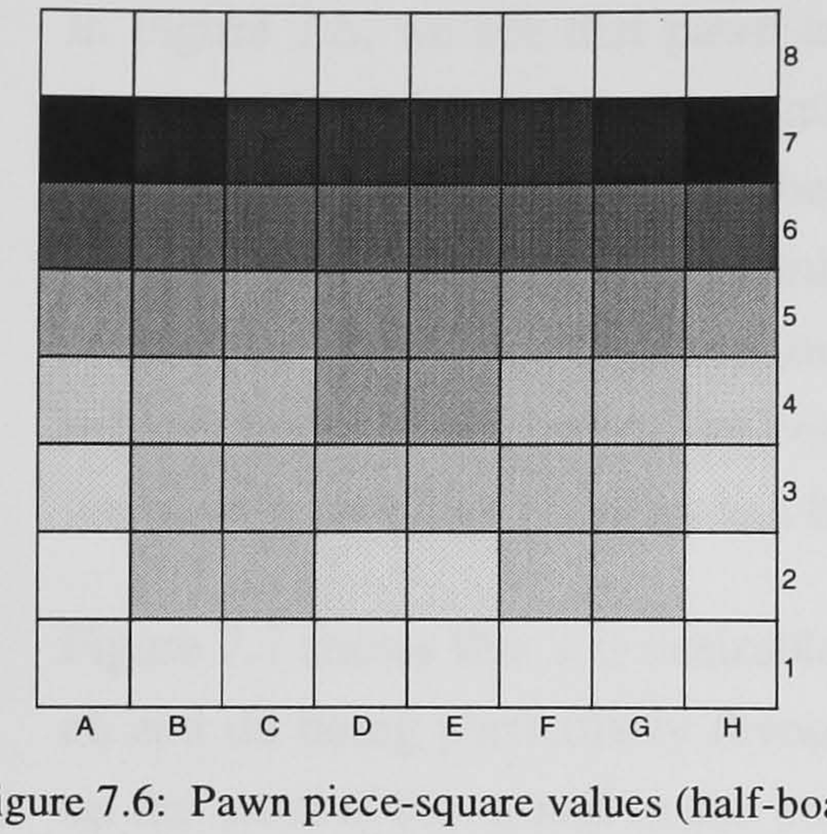

Figure 7.6: Pawn piece-square values (half-board)

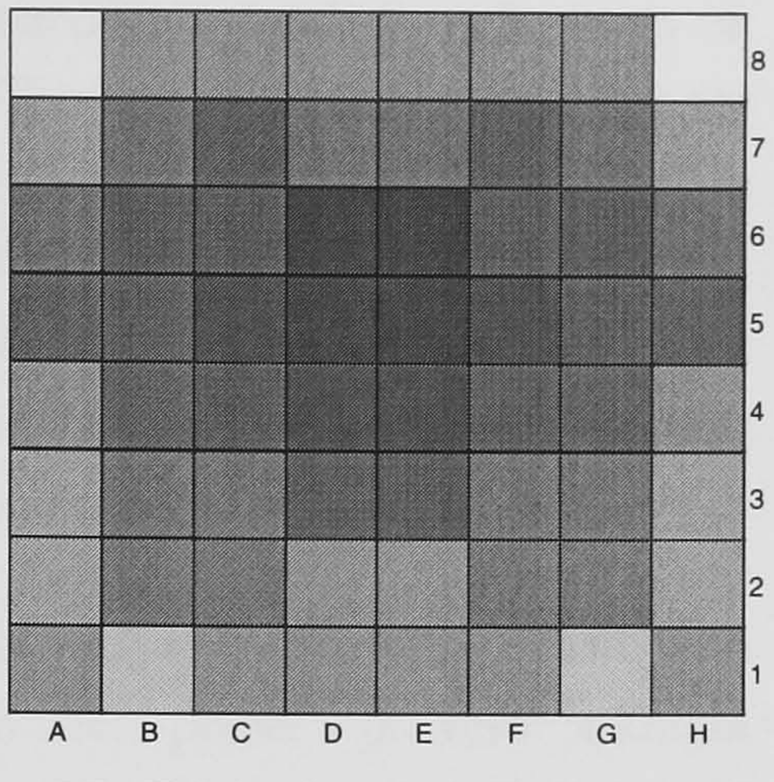

Figure 7.7: Knight piece-square values (half-board)

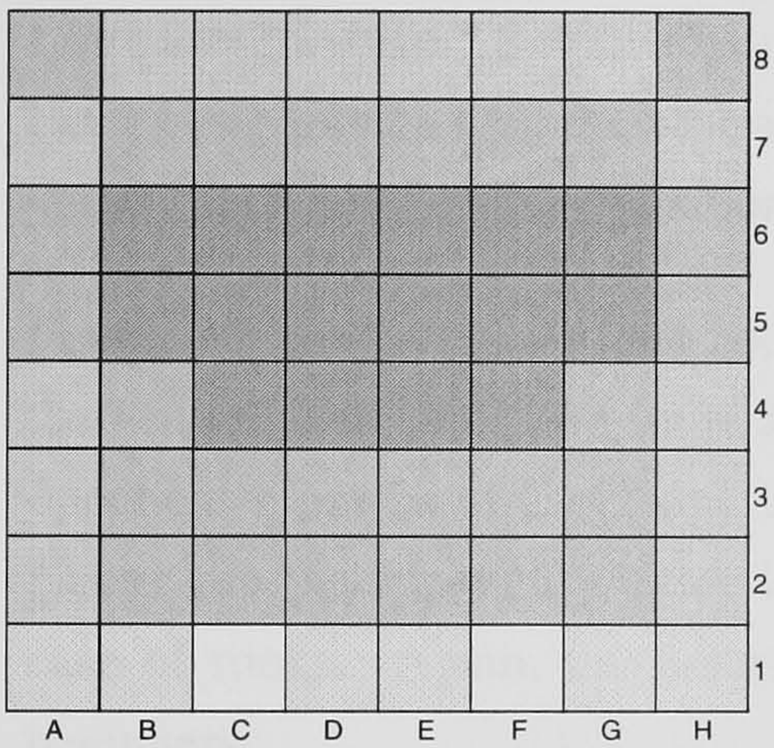

Figure 7.8: Bishop piece-square values (half-board)

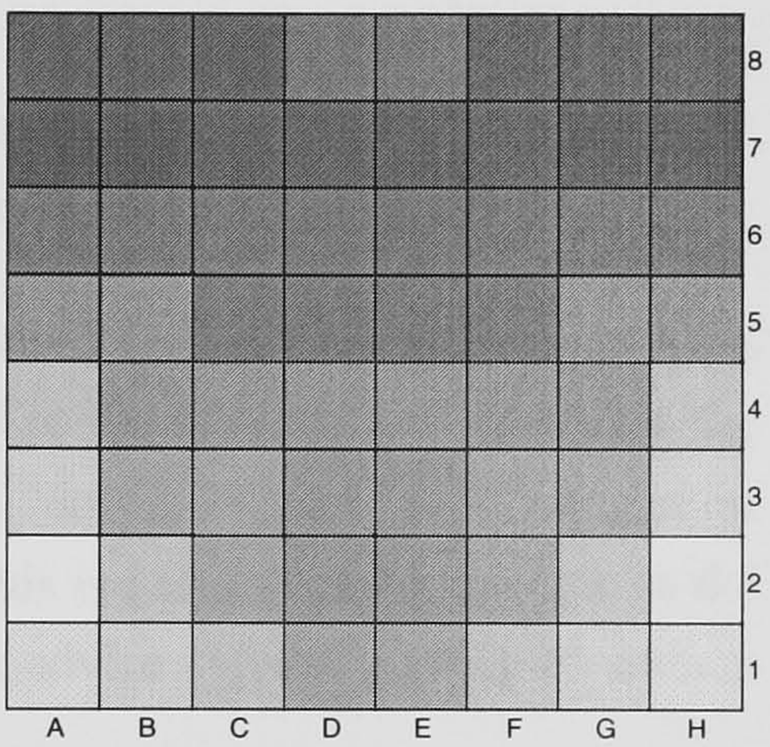

Figure 7.9: Rook piece-square values (half-board)

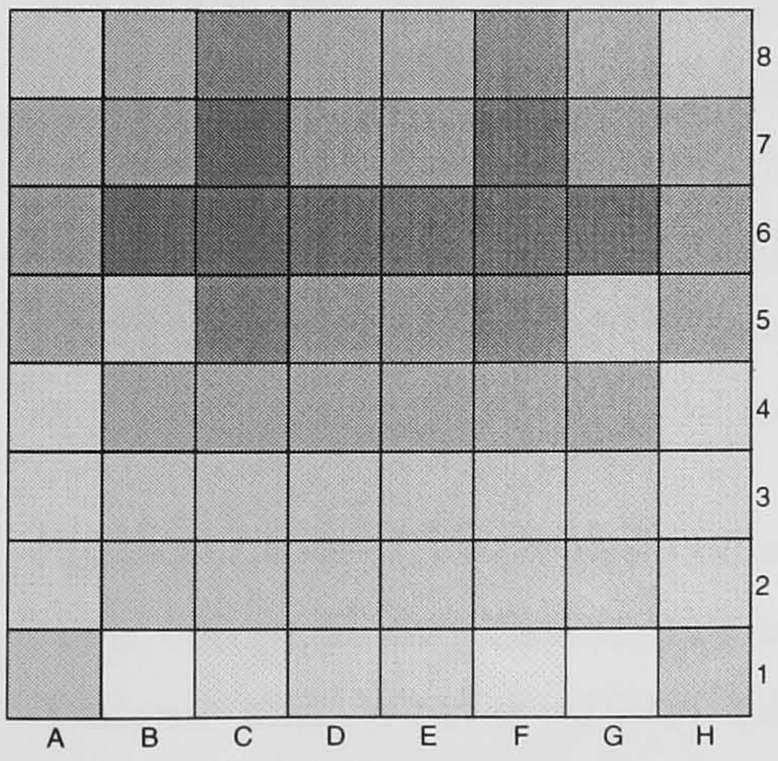

Figure 7.10: Queen piece-square values (half-board)

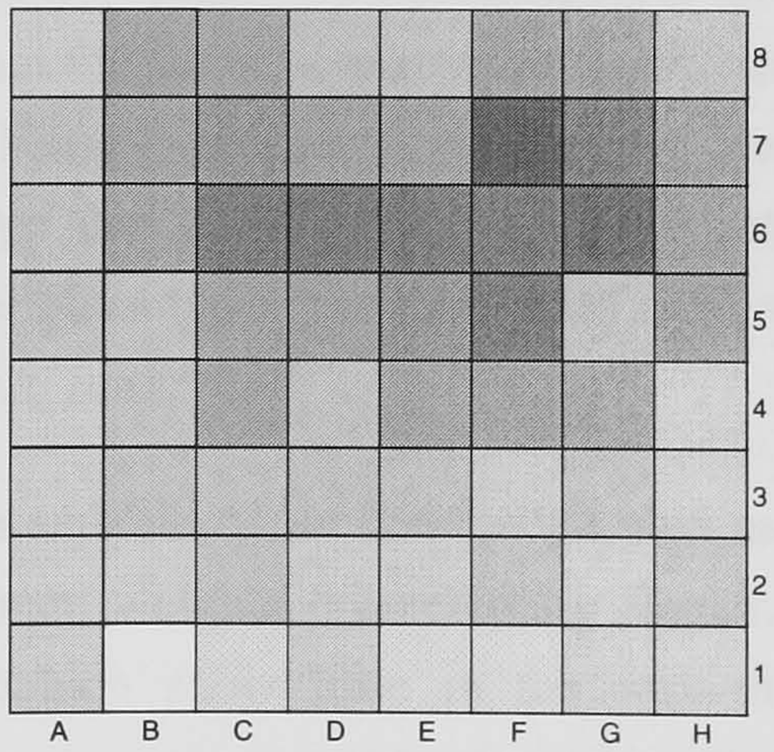

Figure 7.11: Queen piece-square values (full-board)

In Figure 7.6, we see that pawn advancement is generally rewarded, with pawns on the seventh rank receiving a significant benefit. The largest bonus is reserved for pawns on a7 and h7, perhaps because pawns on these squares are the hardest to prevent from queening. Of the initial pawn double-advance moves that are possible, we see that e2-e4 and d2-d4 are the most favoured, as is the case in human play. This reflects both the control of the centre that these moves entail, and also the resulting increase in mobility of queen and bishop, aiding development.

Figure 7.7 shows that it is desirable to position knights in the centre of the board, with e6 and d6 being particularly favourable. There is a chess-player's proverb "a knight on the sixth is like a nail in the knee" that describes the effect that a knight in such a position can have on the opponent's mobility.

Comparing Figure 7.8 with Figure 7.7, we can see that bishop centrality is desirable but less significant than the centrality of knights. In other words, the value of a knight varies much more with its position on the board than does the value of a bishop.

Figure 7.9 shows clearly that rooks can be considered well-placed when occupying the central files, and even more so when advanced to the seventh rank to harass the opponent's pawns and perhaps threaten the king. In all Figures, advancement of pieces into the opponent's camp is rewarded, and this is particularly noticeable in the case of rooks. Again, this information resembles advice typically given to human beginners.

Figure 7.10 shows the half-board piece-square values for the queen. Centrality, and advancing into the opponent's territory, are again rewarded. Comparison with the full-board values shown in Figure 7.11 indicates that the queen scores more highly on the opponent's king-side, with threats against f7 being especially important. In the full-board runs the queen was the only piece type that demonstrated a significant preference for one side of the board. With all other piece types, the half-board results were very similar to those produced by the full-board runs.

Figures 7.6 to 7.11 clearly contain human-understandable chess knowledge, and as such compare very favourably with the piece-square weights presented by Baxter, Tridgell and Weaver (1998) (see section 3.6) which show little in the way of recognisable chess knowledge. For example, their piece-square tables for rooks (the only piece type for which they present learnt piece square tables) show only a heavy penalty for rooks on their starting squares which they suggest is a way for their program to encourage castling.

It should be noted that piece-square tables respond to average positions of enemy pieces. This is also true of much elementary chess knowledge. For example, the desirability of moving rooks to the seventh rank is partly due to the opponent's pawns tending to be positioned there, and the opponent's king being stationed on the back rank. More complex evaluation terms that relate to enemy piece positions can be expected to produce stronger playing programs. Such terms are beyond the scope of this experiment, but suitable values for them could be learnt using methods presented here.

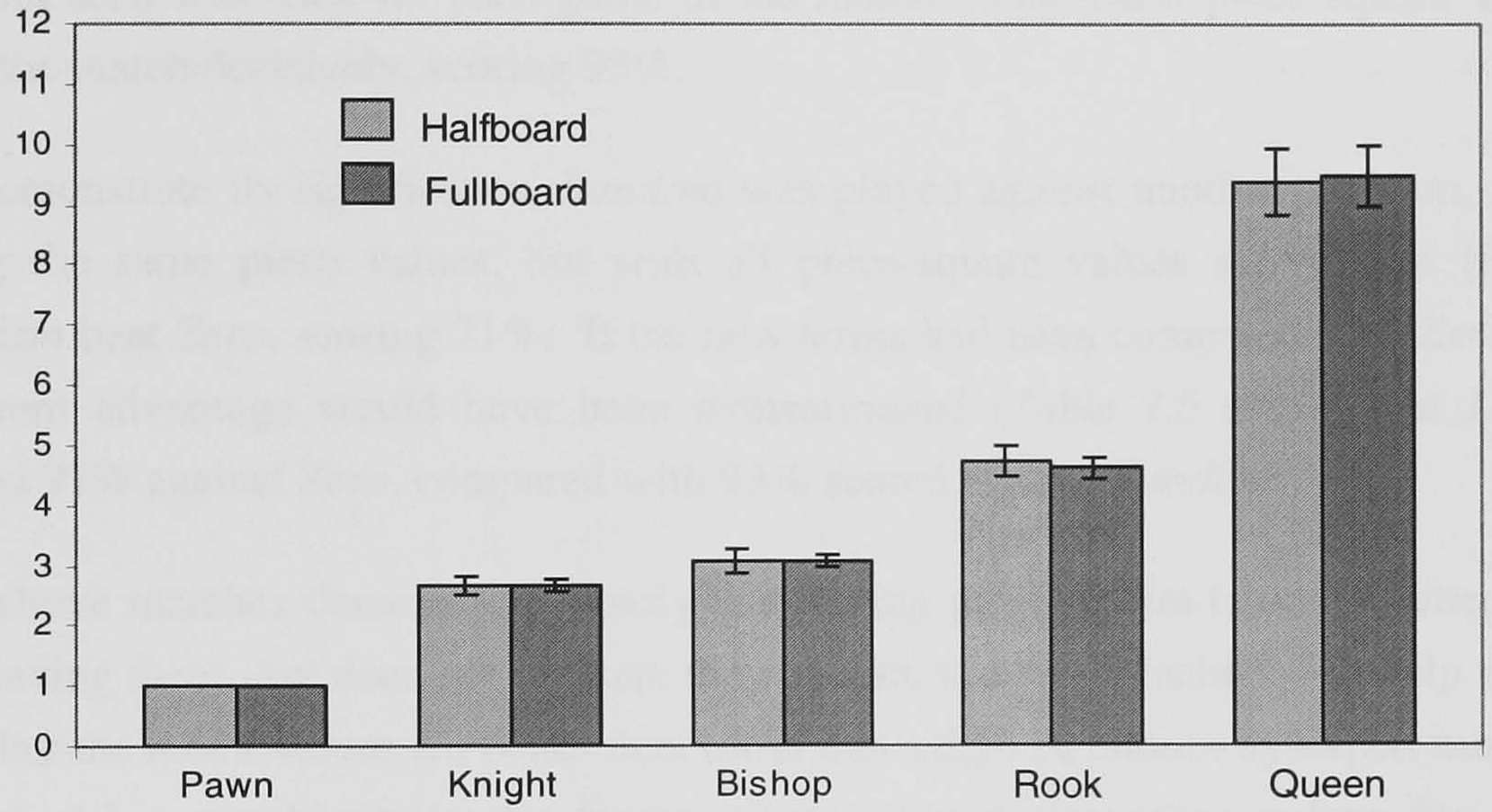

Figure 7.12: Average relative piece values from half-board and full-board runs.

Figure 7.12 shows the average piece values from the half-board and full-board runs, and includes the standard deviation over the 20 trials. From the Figure we can see that both sets of runs learned very similar piece values (full numerical details are provided in Appendix C).

7.4.1 Match results using piece-square values

The learnt piece-square values were tested by playing matches against other value sets. The match results are presented in Table 7.5.

When introducing a new term into an evaluation function used by minimax search, it is natural to try to assess the improvement the new term makes to the performance of the program by playing two versions of the program against each other. One side's evaluation function includes the new term, suitably weighted, and the other side's does not. Surprisingly, this does not necessarily reveal whether the new term is an asset. Even if the new term is entirely random, it might still improve the performance

of the program (see paper *Random Evaluations in Chess* in Appendix E). A better method is to play the new program against a version with a random term that is given the same weight as the new term. This will help to determine whether or not the new evaluation term is measuring anything worthwhile, or is no better than noise.

With the above in mind, the learnt half-board values were pitted in a match against a program that had random values allocated to its piece-square values. This program we called *Random*. Both programs used the same piece values. The random numbers were chosen in a range similar to those found in the learnt values, and a different random seed was used for each game in the match. The learnt piece-square values won the match decisively, scoring 93%.

To demonstrate its significance, *Random* was played against another program, again using the same piece values, but with all piece-square values set to zero (*Zero*). *Random* beat *Zero*, scoring 71%. If the new terms had been compared with *Zero*, the apparent advantage would have been overestimated (Table 7.5 shows that *Learnt* scored 97% against *Zero*, compared with 93% scored against *Random*).

The above matches demonstrate clearly that having piece-square tables is better than not having them, but does not indicate the absolute standard reached. To help assess whether the learnt values are better than those that might be chosen by expert humans, we played a match pitting the learnt values against alternative values for piece centrality and pawn-advancement (*Central*), suggested by a computer chess expert. In this match the learnt values again won decisively, scoring 74%.

Match	*Win*	*Loss*	*Draw*	*Score*
Learnt vs. Random	1,848	127	25	93%
Learnt vs. Zero	1,916	53	31	97%
Random vs. Zero	1,359	517	124	71%
Learnt vs. Central	1,449	475	76	74%
LCentral vs. Central	1,167	704	129	62%
Learnt vs. LCentral	1,114	691	195	61%

Table 7.5: Match results using half-board piece-square values.

The Central piece-square set contained 6 different pawn advancement values, one for each rank, and 4 different values for the other pieces, measuring distance from the centre. Using the methods described above, we learnt replacement weight values for this set, called *LCentral*. The LCentral set scored 62% in a match against Central, indicating that in this simplified environment also, learnt values perform better than our human-chosen values.

Against the LCentral set, the learnt values scored a convincing 61%, demonstrating that the half-board values contain more useful chess knowledge than just pawn advancement and piece centrality.

7.4.2 Ensuring variation in the matches

In order to ensure that each match consisted of 2,000 different games, the scores backed-up to the first ply of search (where the move choice is made) were randomly varied by a small amount (1/10th pawn). This ensured that a wide variety of games were played, and helped to reduce the danger that the self-play games were restricted to a small section of the possible game space. Other methods of variation were tried, including starting each game with a ballot of randomly-chosen moves, and these other methods produced similar match results to those presented above. The figure of 1/10th of a pawn was chosen arbitrarily. Other values were used and produced similar results.

The chess experiments seem to show that it is not necessary to perform deep searches during the learning phase in order to learn effective weights, and that the method is capable of handling large numbers of weights.

The piece-square weights learnt performed better than the suggestions of a human expert, which suggests TD learning is capable of outperforming human expertise, providing that suitable evaluation terms can be identified.

7.5 Learning Weights for other Evaluation Terms

The success of our methods in learning both piece values and piece-square values suggests that such methods may well be useful for setting the weights in high-performance competitive programs.

Don Dailey at MIT has applied our methods to his Grandmaster strength program *Cilkchess* (`http://supertech.lcs.mit`) which recently achieved a creditable fourth place in the 1999 World Computer Chess Championship (ICCA Journal, vol. 22, no. 3), only ½ point behind the winner. He reports the successful learning of a large number (over 200) of positional evaluation weights alongside piece values themselves, and that the performance of his program when using these weights (which have been learnt from scratch) compares well with its performance using carefully hand-tuned weights, scoring approximately 61% in a match between the two

programs. It was intended to use these learnt weights during the World Championship, but due to a programming error discovered at the last minute, it was necessary to revert to the inferior hand-tuned weights. With Don Dailey's permission we reproduce a piece weight trace from a learning run conducted over in excess of 3,000 games. Comparing these results with our own material-only results in Chapter 5 it can been seen that Figure 7.13 is remarkably similar to our own Figure 5.2, indicating that the introduction of over 200 additional positional terms has not had a severely detrimental effect upon the learning of the piece weights themselves.

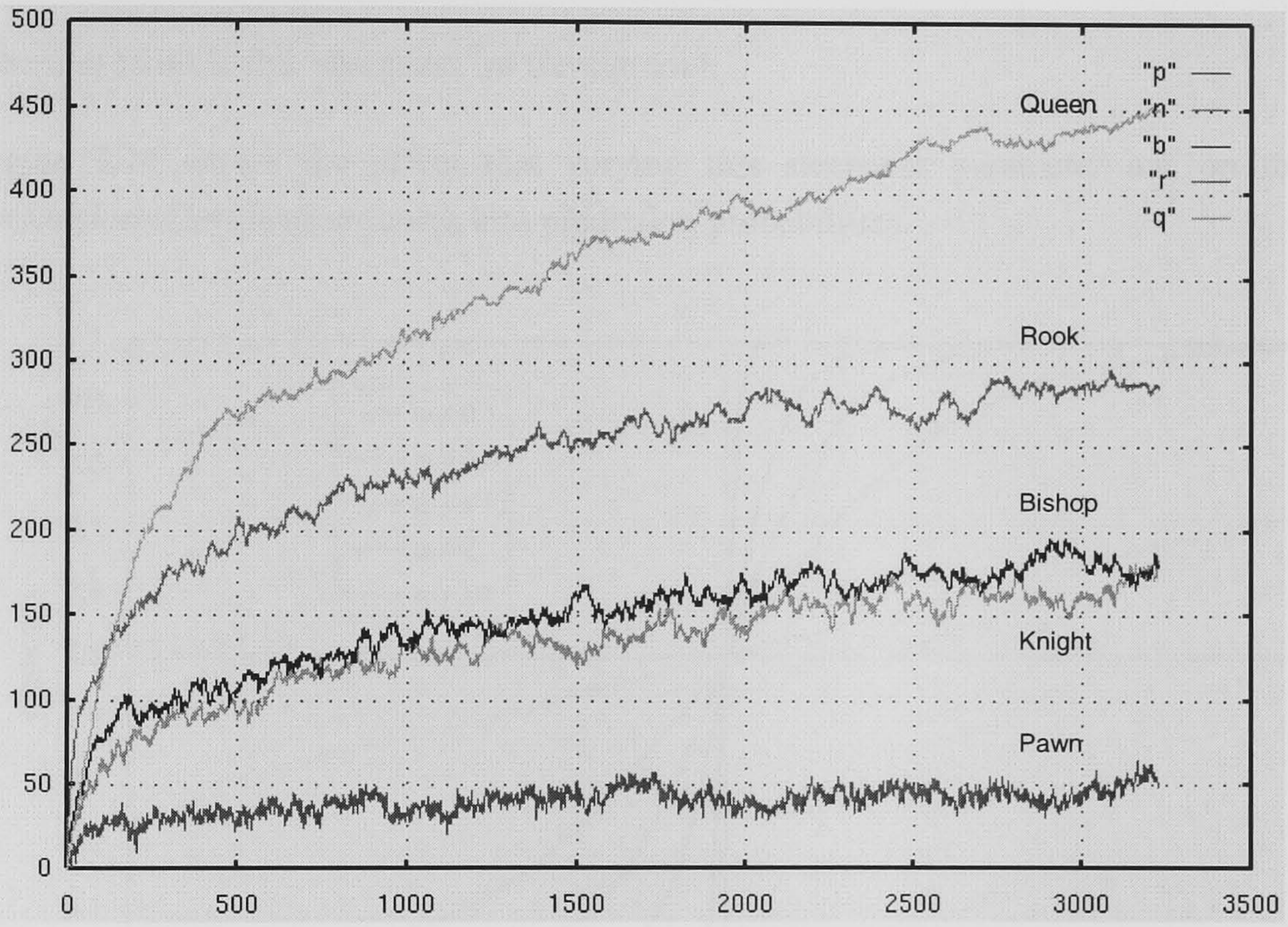

Figure 7.13: Piece weight traces from an experiment at MIT (reproduced with permission).

On a more general note, it is reassuring to find the weight traces produced by a different researcher, using an entirely different search engine, bear a striking resemblance to those we present in Chapters 5 and 8. Don Dailey's experiments were also conducted using selfplay, and the Grandmaster strength performance of his program Cilkchess suggests that online play against strong opponents is not required for successful high-level learning as some have suggested (Baxter, Trigell and Weaver 1988).

7.6 Learning the 'Steepness' of the Squashing Function

A possible modification of the squashing function described in Chapter 3 allows adjustment of the 'steepness' of the sigmoid. In this section we describe a small experiment to investigate how useful this might be. The sigmoid squashing function (3.4) can easily be modified to include an additional parameter:

$$S(v) = \frac{1}{1 + e^{-\omega v}} \tag{7.1}$$

where ω controls the 'steepness' of the sigmoid.

Figure 7.14 shows the effect that varying this steepness parameter has on the conversion of evaluation scores into prediction probabilities.

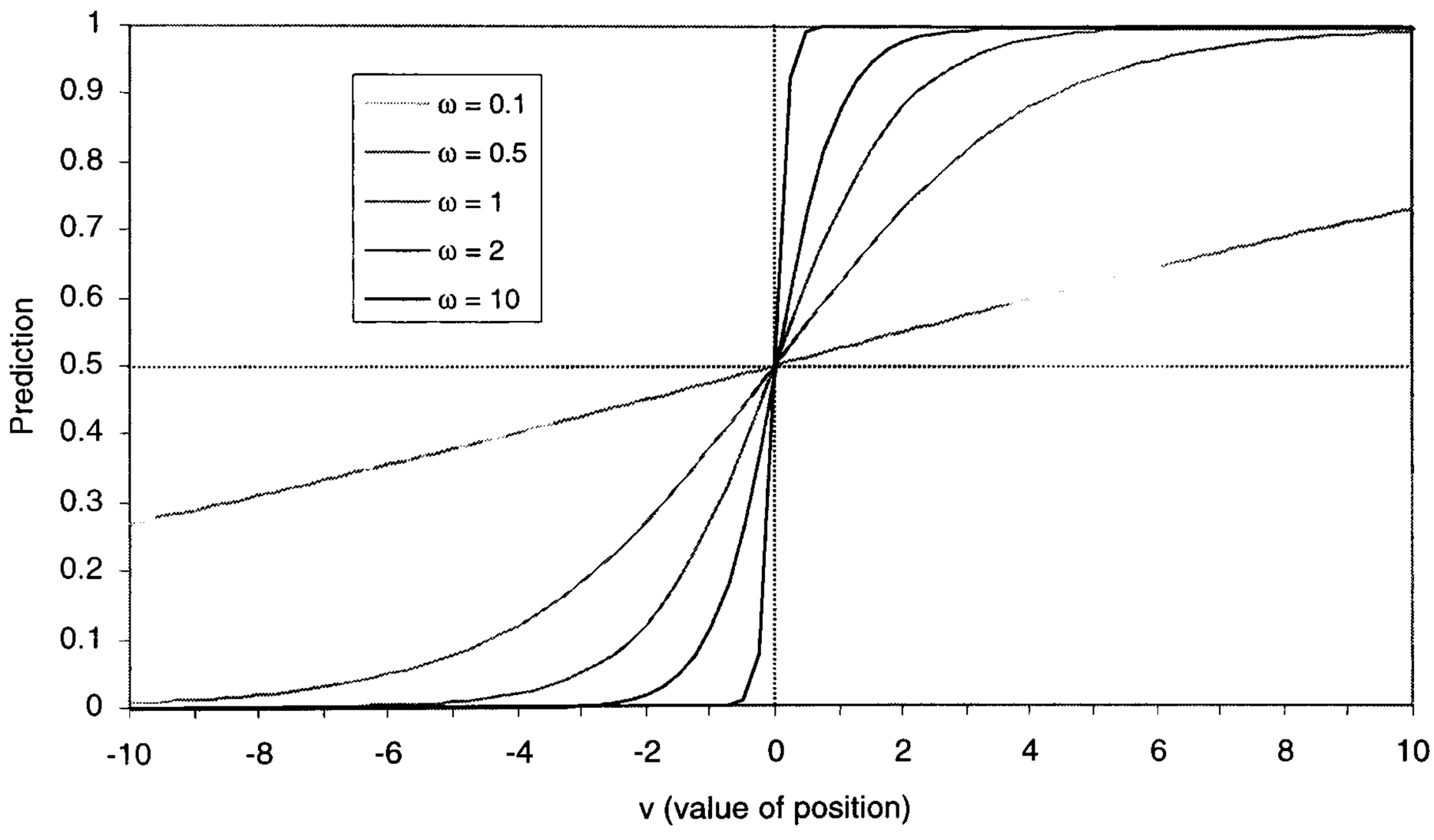

Figure 7.14: Various values of steepness and the resulting predictions.

The experiments of the preceding Chapters effectively used a steepness of 1, as the parameter ω was not used. Table 7.6 shows how the conversion of evaluation score v (using example values from Table 5.1) into predictions vary according to the value of the steepness parameter ω.

Advantage	*v*	$\omega = 0.1$	$\omega = 0.5$	$\omega = 1$	$\omega = 2$	$\omega = 10$
1 Pawn	0.60	0.515	0.574	0.646	0.769	0.998
2 Pawns	1.20	0.530	0.646	0.769	0.917	1.000
1 Knight	1.66	0.541	0.696	0.840	0.965	1.000
1 Bishop	2.02	0.550	0.733	0.883	0.983	1.000
1 Rook	2.75	0.568	0.798	0.940	0.996	1.000
1 Queen	6.61	0.659	0.965	0.999	1.000	1.000

Table 7.6: Examples of material advantages (from Run A in Table 5.1) and their corresponding predictions using various steepness values.

The introduction of a steepness parameter has the effect of scaling the absolute values of the pieces, but should have little effect on the normalised final values. We experimented with allowing this parameter to be learnt along with the piece weights, but found that the final normalised piece values were very similar. Figure 7.15 shows the trace of the steepness parameter over two runs, *A* and *B*. Both runs were conducted using the same random seeds, but in *A* the steepness parameter was initialised to 1 and in *B* it was initialised to 0.5

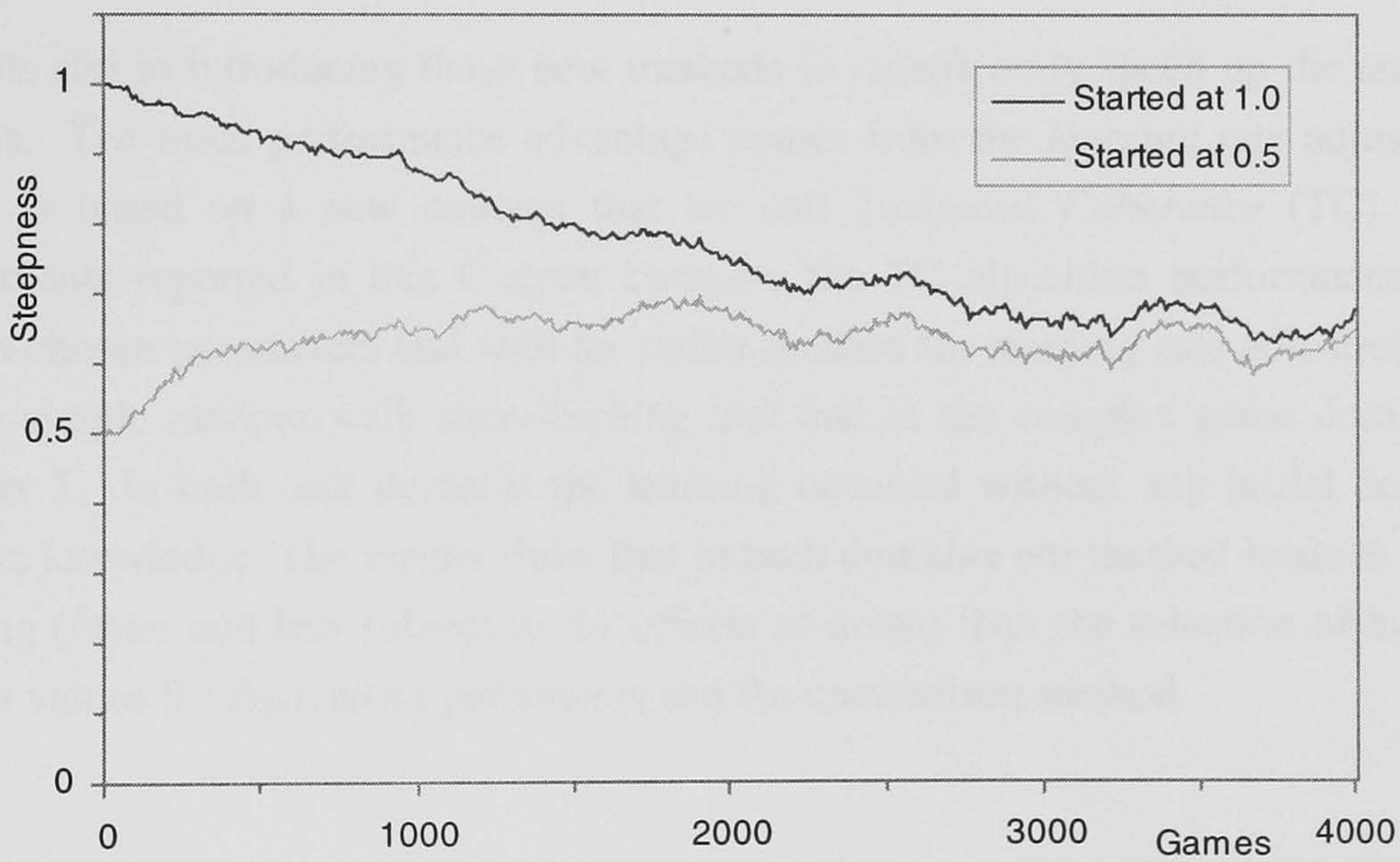

Figure 7.15: Steepness traces converging from different starting points.

From the Figure we can see that the first few hundred games played were very different, but that once the steepness values have almost converged, and correspondingly so have the piece values, then the games played were identical. This of course is to be expected once the relative piece values are the same, given that both runs used identical random number seeds. This experiment suggests that the selection of the steepness parameter is not an important factor in the learning runs, and that steepness $\neq 1$ is equivalent to scaling the weights.

8 TEMPORAL COHERENCE AND PREDICTION DECAY

This Chapter presents the most important contribution of this thesis. We describe an extension of the temporal difference learning method, designed to greatly improve the efficiency of the learning by reducing the number of trials required. The standard form of the TD(λ) method as described in Chapter 3 has the problem that two control parameters, learning rate and temporal discount, need to be chosen appropriately. These parameters can have a major effect on performance, particularly the learning rate, which affects the stability of the process as well as the number of observations required. Our novel extension to the TD(λ) algorithm automatically sets and subsequently adjusts these parameters. Most of this section was published (Beal and Smith 1999b) in the *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99).*

It is our aim in introducing these new methods to significantly speed up the learning process. The main performance advantage comes from the learning rate adjustment which is based on a new concept that we call *Temporal Coherence* (TC). The experiments reported in this Chapter compare the TC algorithm performance with human-chosen parameters and with an earlier method for learning rate adjustment, in both a simple random-walk state-learning task and in the complex game domain of Chapter 5. In both task domains the learning occurred without any initial domain-specific knowledge. The results show that in both domains our method leads to better learning (faster and less subject to the effects of noise) than the selection of human-chosen values for the control parameters and the comparison method.

8.1 Control Parameters for TD(λ)

Two major parameters that control the behaviour of the TD(λ) algorithm are the learning rate (or *step-size),* α, and the temporal discount parameter, λ (see equation 3.1 in Chapter 3).

The choice of these parameters can have a major effect on the efficacy of the learning algorithm. Selection of the learning rate parameter α is particularly hard to get right. It needs to be as high as possible for rapid learning, but high rates lead to high levels of erratic movements, even after optimum values may have been reached. In effect,

high learning rates lead to high levels of noise in the weight movements, and this means that the process does not produce stable values.

On the other hand, learning rates that are too low can lead to orders of magnitude more observations being required to reach optimum weight values. Experience with the TD(λ) method in practice has shown that very different values of α are required in different domains, as shown by the different rates used in, for example, Sutton (1988, 1992).

In practical problems, the control parameters are often determined somewhat arbitrarily or else by trying a number of values and 'seeing what works' (e.g. Tesauro 1992). This was the method we used for the experiments of Chapter 5. Another widely used method is to use a learning rate that decreases over time. This was the method we used in Chapter 6, in an attempt to reduce the computational cost of our experiments in shogi. However, such systems still require the selection of a suitable schedule, and in Chapter 6 we found it necessary to try a number of different schedules and choose the one that seemed to work best in that particular domain.

Sutton and Singh (1994) describe systems for setting both α and λ, within the framework of Markov-chain models. These methods assume relatively small numbers of distinct states, and acylic graphs, and so are not directly applicable to more complex real-world problems. Jacobs (1988) presented the 'delta-bar-delta' algorithm for adjusting α during the learning process. We compared the performance of delta-bar-delta with our algorithm on two sample domains. More recently, Almeida *et al.* (1998) and Schraudolph (1998) have presented other methods for α adaptation in stochastic domains and neural networks respectively.

We describe a new system which adjusts α and λ automatically. This system does not require *a priori* knowledge about suitable values for learning rate or temporal discount parameters for a given domain. It adjusts the learning rate and temporal discount parameters according to the learning experiences themselves. We present results to show that this method is effective. In our sample domains the new methods yielded better learning performance than our best attempt to find optimum choices of fixed α and λ, and better learning performance than delta-bar-delta.

8.2 Temporal Coherence: Adjustments to Learning Rates

Our system of self-adjusting learning rates is based on the concept that the learning rate should be higher when significant learning is taking place, and lower when changes to the weights are primarily due to noise. Random noise will tend to produce adjustments that cancel out as they accumulate. Adjustments making useful adaptations to the observed predictions will tend to reinforce as they accumulate. As weight values approach their optimum, prediction errors will become mainly random noise.

Motivated by these considerations, our Temporal Coherence (TC) method estimates the significance of the weight movements by the relative strength of reinforcing adjustments to total adjustments. The learning rate is set according to the proportion of reinforcing adjustments as a fraction of all adjustments. This method has the desirable property that the learning rate reduces as optimum values are approached, tending towards zero at optimum values. It has the equally desirable property of allowing the learning rate to increase if random adjustments are subsequently followed by a consistent trend.

Separate learning rates are maintained for each weight, so that weights that have become close to optimum do not fluctuate unnecessarily and thereby add to the noise affecting predictions. The use of a separate learning rate for each weight allows for the possibility that different weights might become stable at different times during the learning process. For example, if weight A has become fairly stable after 100 updates, but weight B is still consistently rising, then it is desirable for the learning rate for weight B to be higher than that for weight A. An additional potential advantage of separate learning rates is that individual weights can be independent when new weights are added to the learning process. If new terms or nodes are added to an existing predictor, independent rates make it possible for the new weights to adjust quickly, whilst existing weights only increase their learning rates in response to perceived need.

The TC learning rates are determined by the history of *recommended changes* to each weight. We use the term 'recommended change' to mean the temporal difference adjustment prior to multiplication by the learning rate. This detachment of the learning rate enables the TC algorithm to respond to the underlying adjustment impulses, unaffected by its own recent choice of learning rate. It has the additional advantage that if the learning rate should reach zero, future learning rates are still free to be non-zero, and the learning does not halt.

The *recommended change* for weight w_i at timestep t is defined as:

$$r_{i,t} = (P_{t+1} - P_t)\sum_{k=1}^{t} \lambda^{t-k} \nabla_{w_i} P_k \tag{8.1}$$

The actual change made to weight w_i after each game is:

$$\Delta w_i = c\ \alpha_i \sum_{t=1}^{end-1} r_{i,t} \tag{8.2}$$

where α_i is the individual learning rate for weight w_i, and c is the learning rate for the whole process

For each weight we are interested in two numbers: the accumulated *net change* (the sum of the individual recommended changes); and the accumulated *absolute cha*nge (the sum of the absolute individual recommended changes). The ratio of net change, *N*, to absolute change, *A*, allows us to measure whether the adjustments to a given weight are mainly in the same 'direction'. We take reinforcing adjustments as indicating an underlying trend, and cancelling adjustments as indicating noise from the stochastic nature of the domain (or limitations of the domain model that contains the weights). The individual learning rate, α_i for each weight w_i, is set to be the ratio of net recommended change to absolute recommended change:

$$\alpha_i = \frac{|N_i|}{A_i} \tag{8.3}$$

with the following definitions and update rules:

$$N_i \leftarrow N_i + \sum_{t=1}^{end-1} r_{i,t} \tag{8.4}$$

$$A_i \leftarrow A_i + \sum_{t=1}^{end-1} |r_{i,t}| \tag{8.5}$$

$r_{i,t}$ = recommended change for weight w_i at prediction t

$P_1..P_{end-1}$ are predictions, P_{end} is the final outcome

The operational order is that changes to w_i are made first, using the previous values of N_i, A_i and α_i ; then N_i, A_i and α_i are updated. The parameter c has to be chosen, but this does not demand a choice between fast learning and eventual stability, since it

can be set high initially, and the α_i then provide automatic adjustment during the learning process. All the α_i are initialised to 1 at the start of the learning process.

The foregoing formulae describe updating the weights and learning rates at the end of each sequence. The method can be amended easily to update more frequently (e.g. after each prediction), or less frequently (e.g. after a batch of sequences). For the experiments reported in this Chapter, update at the end of each sequence is natural and convenient to implement.

8.3 Prediction Decay: Determining λ

We determine a value for the temporal discount parameter, λ, by computing a quantity ψ we call *prediction decay*. Prediction decay is a function of observed prediction values, indexed by temporal distance between them, and described in more detail in section 8.3.1. An exponential curve is fitted to the observed data, and the exponential constant, ψ, from the fitted curve is the *prediction decay*. We set $\lambda = 1$ initially and $\lambda = \psi$ thereafter.

The use of $\lambda = \psi$ has the desirable characteristics that (i) a perfect predictor will result in $\psi = 1$, and TD(1) is an appropriate value for the limiting case as predictions approach perfection, and (ii) as the prediction reliability increases, ψ increases, and it is reasonable to choose higher values of λ for TD learning as the prediction reliability improves. We make no claim that setting $\lambda = \psi$ is optimum[1]. Our experience is that typically it performs better than human-guessed choices of fixed λ *a priori*.

The advantage of using prediction decay is that it enables TD learning to be applied effectively to domains without prior domain knowledge, and without prior experiments to determine an optimum λ. When combined with our method for adjusting learning rates, the resulting algorithm performs better than the comparison method, and better than using fixed rates, in both test domains.

8.3.1 Setting the temporal discount parameter using prediction decay

Prediction decay is the average deterioration in prediction quality per timestep. A *prediction quality* function measures the correspondence between a prediction and a

---

[1] By expending sufficient computation time to repeatedly re-run the experiments we were able to find somewhat better values for λ.

later prediction (or end-of-sequence outcome). The observed prediction qualities for each temporal distance are averaged. An exponential curve is then fitted to the average prediction qualities against distance (Figure 8.1 shows an example), and the exponential constant of that fitted curve is the prediction decay, ψ. We set the TD discount parameter λ to1 initially, and $\lambda = \psi$ thereafter. In the experiments reported, ψ (and hence λ) were updated at the end of each sequence.

The *prediction quality* measure, $Q_d(p, p')$ we used is defined below. It is constructed as a piece-wise linear function with the following properties:

(a) When the two predictions p and p', are identical, $Q_d = 1$. (The maximum Q_d is 1.)

(b) As the discrepancy between p and p' increases, Q_d decreases.

(c) When one prediction is 1 and the other is 0, then $Q_d = -1$. (The minimum Q_d is –1.)

(d) For any given p, the average value of Q_d for all possible values of p', such that $0 \le p' \le 1$, equals 0. (Thus random guessing yields a score of zero.) This property is achieved by the quadratic equations in the definition below.

We achieve these properties by defining:

$$Q_d(p, p') = \left\{ \begin{array}{lll} p \ge .5 & : & F(p, p') \\ p < .5 & : & F(1-p, 1-p') \end{array} \right\} \tag{8.6}$$

$$F(p, p') = \left\{ \begin{array}{lll} p \le s & : & \left\{ \begin{array}{lll} r \le x & : & 1 - r/x \\ r > x & : & -p(r-x)/(p-x) \end{array} \right\} \\ p > s & : & \left\{ \begin{array}{lll} r \le y & : & 1 - r/y \\ r > y & : & -p(r-y)/(p-y) \end{array} \right\} \end{array} \right\} \tag{8.7}$$

where:

$$r = |p - p'|$$

$$s = \text{ solution of } \quad 2s^2 - 5s + 1 = 0$$

$$x = \text{ solution of } \quad 2(1+p)x^2 - 4px + p = 0$$

$$y = \text{ solution of } \quad (1+p)y^2 + \left(2 - 2p - p^2\right)y - (1-p)^2 = 0$$

p is the current prediction, p' is an earlier prediction, and d refers to the temporal distance between p and p'. Predictions lie in the range [0, 1].

It is assumed that learning occurs over the course of many multi-step sequences, in which a prediction is made at each step; and that the sequences are independent. To form a prediction pair, both predictions must lie within the same sequence.

$\overline{Q}_d$ is the average prediction quality over all prediction pairs separated by distance d observed so far. For this purpose, the terminal outcome at the end of the sequence is treated as a prediction. At every prediction, values of $\overline{Q}_d$ are incrementally updated.

An example graph from our experimental results is given in Figure 8.1. This example is typical of the fit to the observed data in the game domain. The exponential curve is fitted to the averaged prediction qualities by minimising the mean squared error between the exponential curve and the observed $\overline{Q}_d$ values. ψ was fairly stable in the range 0.990 – 0.993 during the test runs.

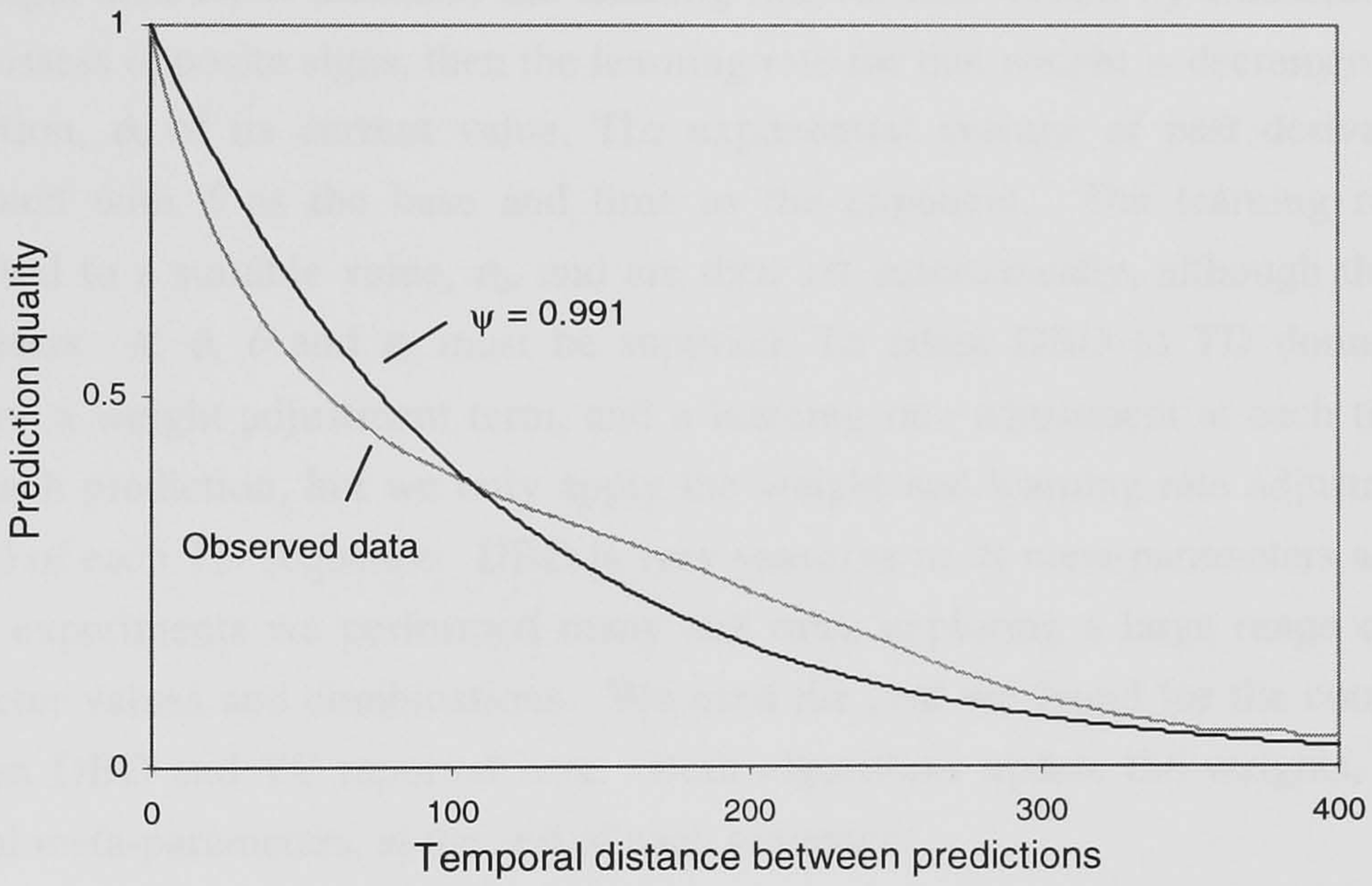

Figure 8.1: Fit of the prediction quality temporal decay to observed data from the game domain, at the end of a run of 2000 games.

To prevent rarely occurring distances from carrying undue weight in the overall error, the error term for each distance is weighted by the number of observed prediction pairs. Thus we seek a value of ψ which minimises:

$$\sum_{d=0}^{l} \left(\overline{Q}_d - \psi^d\right)^2 N_d \qquad (8.8)$$

where $\overline{Q}_d$ is the average prediction quality for distance d, and N_d is the number of prediction pairs separated by that distance, and l is the length of the longest sequence in the observations so far. In the experiments reported here the value for ψ was

obtained by simple iterative means, making small incremental changes to its value until a minimum was identified. Values for ψ (and hence λ) were updated at the end of each sequence.

8.4 Delta-bar-delta

The delta-bar-delta algorithm (DBD) for adapting learning rates is described by Jacobs (1988). Sutton (1992) later introduced Incremental DBD for linear tasks. The original DBD was directly applied to non-linear tasks, and hence more easily adapted to both our test domains. In common with our temporal coherence method, it maintains a separate learning rate for each weight. If the current derivative of a weight and the exponential average of the weight's previous derivatives possess the same sign, then DBD increases the learning rate for that weight by a constant, κ. If they possess opposite signs, then the learning rate for that weight is decremented by a proportion, ϕ, of its current value. The exponential average of past derivatives is calculated with θ as the base and time as the exponent. The learning rates are initialised to a suitable value, ε_0, and are then set automatically, although the meta-parameters κ, ϕ, θ and ε_0 must be supplied. To adapt DBD to TD domains, we compute a weight adjustment term, and a learning rate adjustment at each timestep, after each prediction, but we only apply the weight and learning rate adjustments at the end of each TD sequence. DBD is very sensitive to its meta-parameters and prior to our experiments we performed many test runs, exploring a large range of meta-parameter values and combinations. We used the best we found for the comparison between DBD and TC reported here. Both algorithms update the weights, and the internal meta-parameters, at the end of each sequence.

8.5 Test Domain One: A Bounded Random Walk

The methods described in this Chapter are designed to be domain independent, and should be applicable over a wide range of possible domains. We report first on a simple domain, that of a bounded random walk where the task is to learn the probability of terminating the walk at a particular state. We selected this domain because it has been used as a test domain before, for example by Sutton (1988) and Dayan (1992).

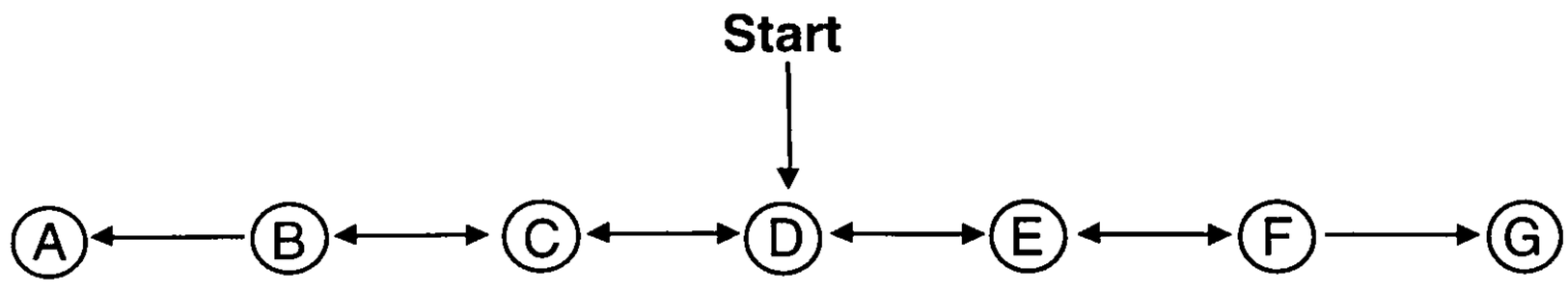

Figure 8.2: A bounded random walk.

All walks begin in state D. When in states B, C, D, E, and F, there is a 50% chance of moving to the adjacent left state and a 50% chance of moving to the adjacent right state. When either end state (A or G) is reached, then the walk terminates, with a final outcome defined as 0 in state A, and 1 in state G. This absorbing Markov process generated the random walks used in the experiments. Each sequence for the TD learning process is based on one walk. The learning task is to obtain five weights, one for each of the five internal states. These weights are estimates of the probabilities of terminating the walk at G, starting at the given internal state.

Sutton (1988) presents this task and shows that temporal difference learning is more effective here than the widely-used supervised learning method of Least Mean Square (Widrow and Hoff 1960), given an appropriate choice of control parameters. His experiments showed that the results achieved by TD(λ) in this domain were sensitive to the choice of both α and λ.

8.5.1 Results from the bounded random walk

For each experiment, a set of 1,000 random walks was generated and each of the learning procedures was then applied to the same 1,000 sequences. With this large number of sequences, the values towards the end of the run, averaged over recent sequences, are very close to the known theoretical value, and the weight movements are random noise. Nevertheless, to allow for variations due to different random sequences, each experiment was repeated 10 times using a different random number seed. The results from each of the 10 seeds were all very similar. Figures 8.3 through 8.6 are derived from one particular starting seed, and are typical. Figure 8.7 presents results averaged from all 10 seeds.

Figures 8.3 and 8.4 show the weight movements from typical runs, using TD with two fixed learning rates, chosen to cover the range we found to be best from many runs, and a fixed λ (0.3) chosen as the most suitable for this task from results presented by Sutton (1988). The five traces on each graph show the estimated values of the five unknown states, after each of 1,000 sequences. For this task, the true values are known, and are shown on the graphs as horizontal lines. The graphs illustrate that the

higher the learning rate, the faster the weights approach the target values initially, but also illustrate that as the learning rate rises, the less stable the weight values become. At high rates, it may become impracticable to extract stable weights.

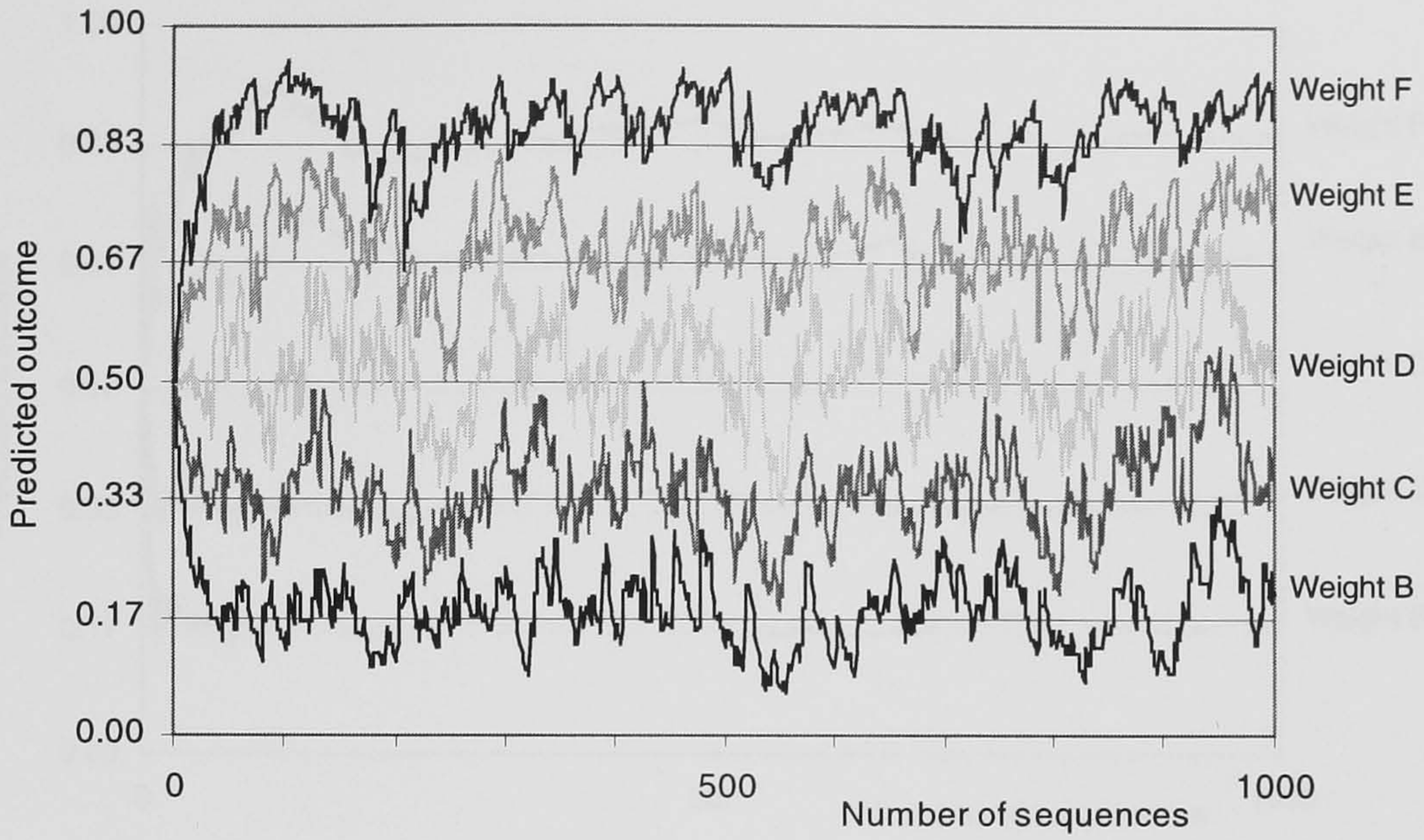

Figure 8.3: Weight movements from a typical run using a fixed α of 0.1.

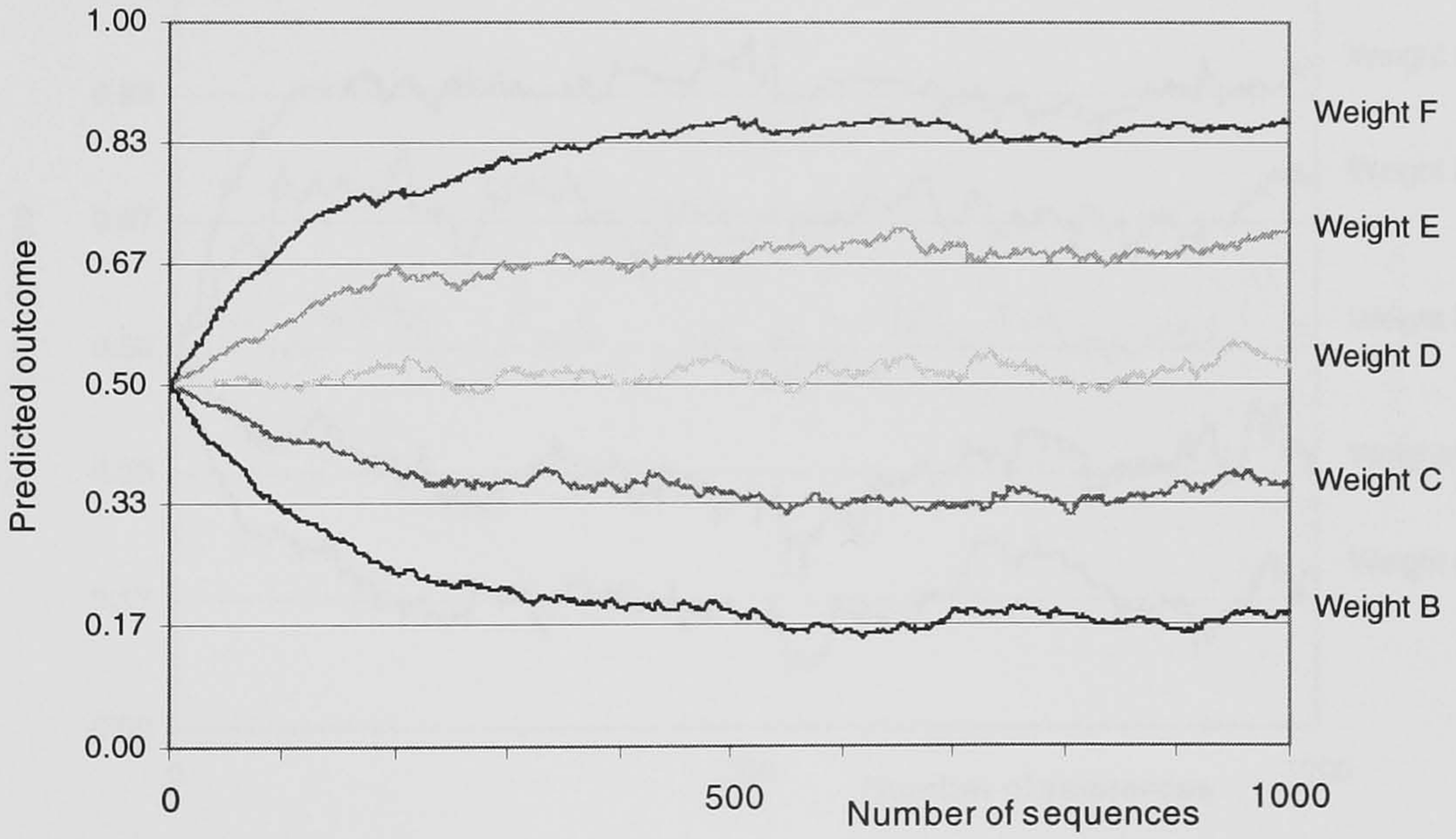

Figure 8.4: Weight movements from a typical run using a fixed α of 0.01.

From Figure 8.3 it can be seen that the weight adjustment made using a fixed α of 0.1 are seriously unstable, even towards the end of the run, when the average value is close to the desired value. This learning rate is too high for obtaining stable weights. On the other hand, Figure 8.4 shows that if the learning rate is lower, the final weights

are much more stable. However, the weights in Figure 8.4 take of the order of 500 sequences to approach the right values, whereas the high learning rate of Figure 8.3 only took around 50 sequences. This behaviour was repeated in each of the 10 runs.

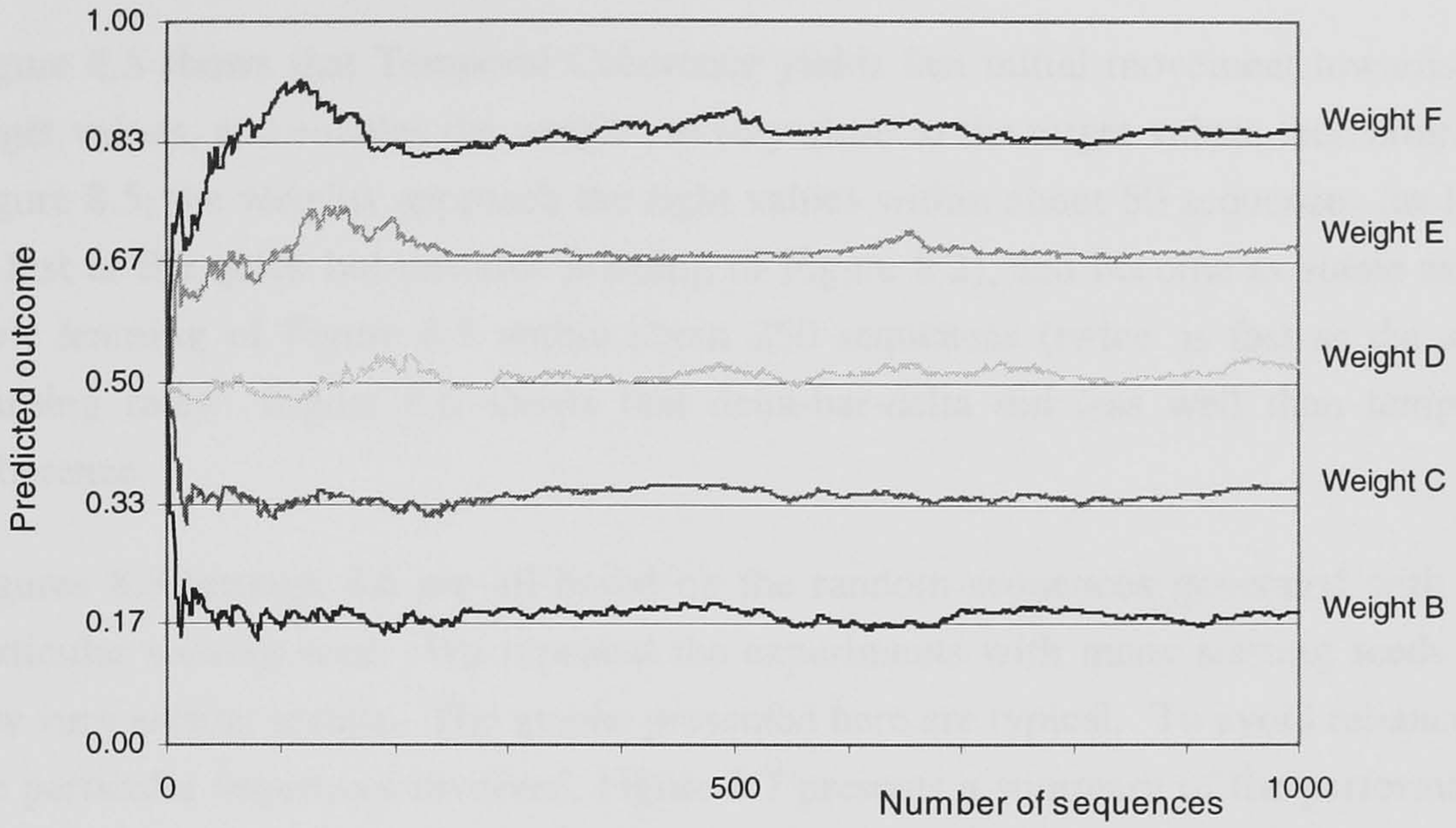

Figure 8.5: Weight movements from a typical run using temporal coherence.

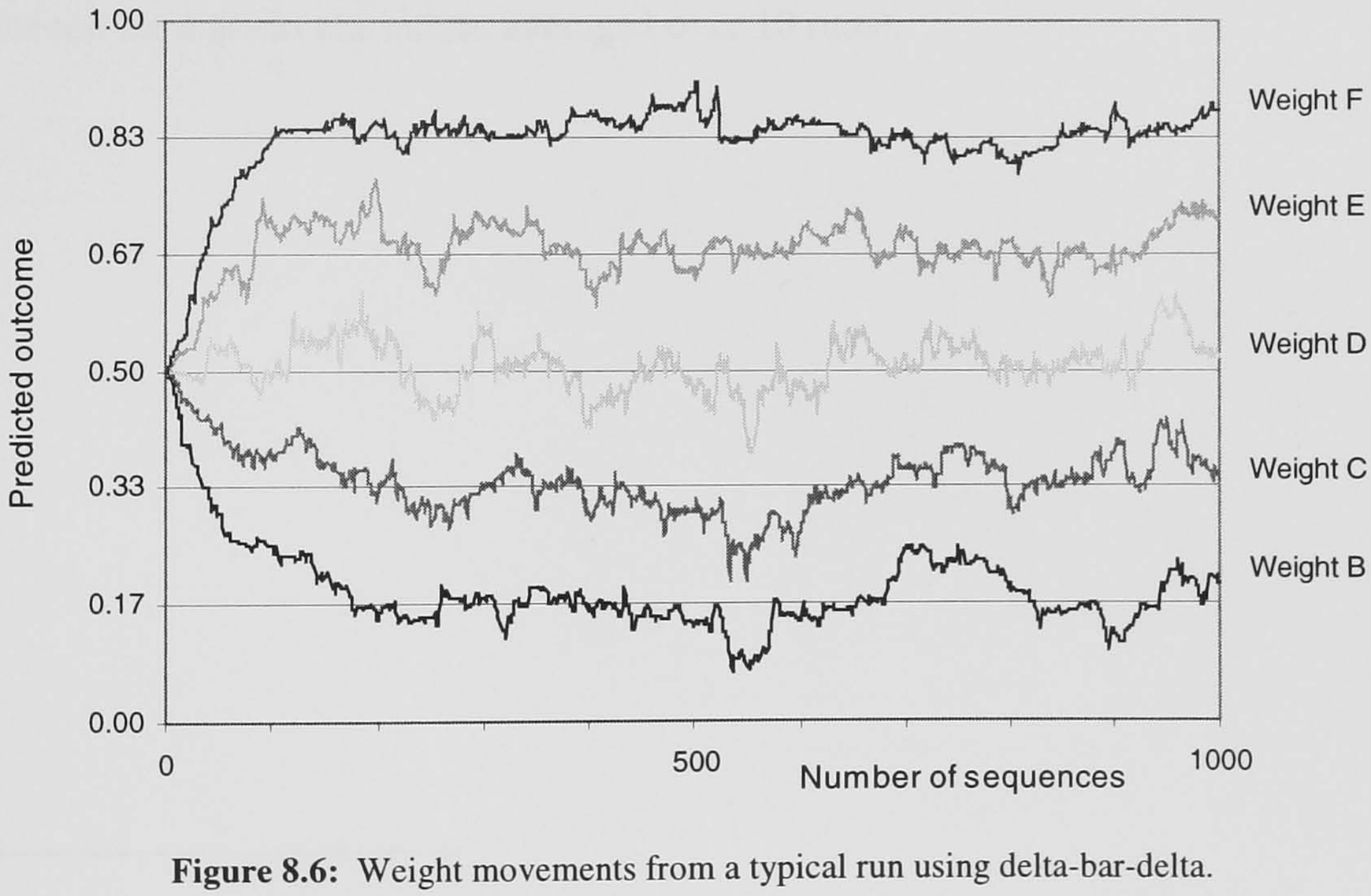

Figure 8.6: Weight movements from a typical run using delta-bar-delta.

Figures 8.5 and 8.6 show typical results from our temporal coherence algorithm, and delta-bar-delta, respectively. Both of these methods automatically adjust the learning rate. We found the delta-bar-delta algorithm to be highly sensitive to its meta-parameters (starting rate, adjustment step size and ratio, and exponential decay factor of weight changes). Figure 8.6 shows the best result from several runs performed

with different values of the meta-parameters, guided by the values used by Jacobs (1988). We used parameter settings of: $\kappa = 0.01$, $\phi = 0.333$, $\theta = 0.7$, $\varepsilon_0 = 0.03$ and λ=0.3. We tried a number of other parameter settings, none of which performed any better than the chosen set.

Figure 8.5 shows that Temporal Coherence yields fast initial movement towards the target values, and enables the weights to stay close to the target values thereafter. In Figure 8.5, the weights approach the right values within about 50 sequences (as least as fast as the quick but unstable learning of Figure 8.2), and become as stable as the slow learning of Figure 8.3 within about 250 sequences (twice as fast as the slow learning rate). Figure 8.6 shows that delta-bar-delta did less well than temporal coherence.

Figures 8.3 through 8.6 are all based on the random sequences generated with one particular starting seed. We repeated the experiments with many starting seeds and saw very similar results. The graphs presented here are typical. To avoid reliance on the particular sequences involved, Figure 8.7 presents a summary of the performance of the different algorithms, showing their progress towards the values sought. Each trace in Figure 8.7 represents the squared error, summed over all weights, after each sequence for a given algorithm, averaged over 10 runs[2].

---

[2] For presentation convenience, the sum-of-squared error is first normalised to the range [0,1] where 1 is the error at the start of the run, and 0 is no error (achieved when the weight equals the true value), then charted as g, the inverse of error:

$$g = 1 - \sum_{i=1}^{5} (w_i - v_i)^2 \Big/ \sum_{i=1}^{5} (0.5 - v_i)^2 \qquad (8.9)$$

w_i is the weight for state i, v_i is the true value for state i, 0.5 is the initial value for all weights at the start of the run.

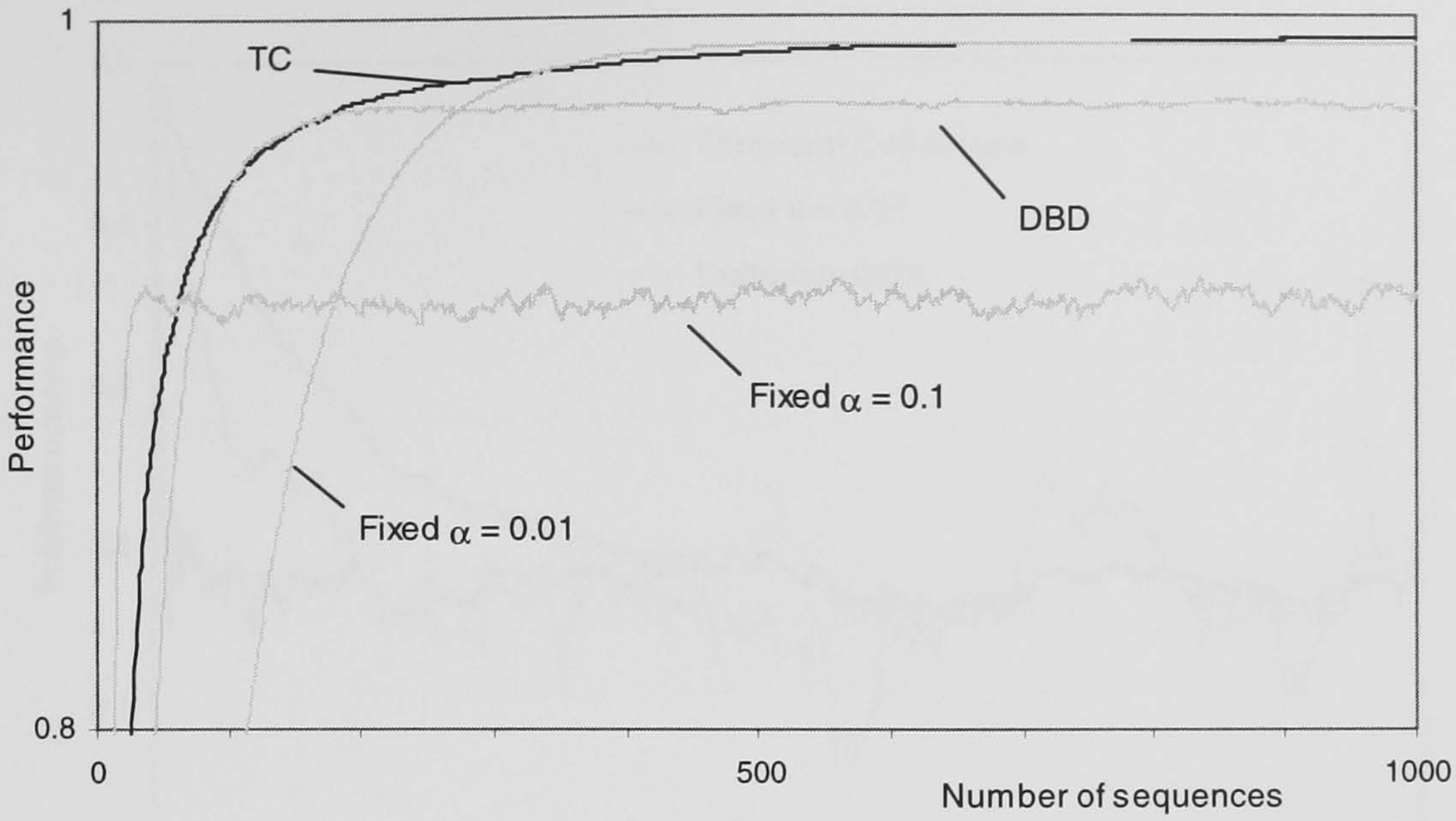

Figure 8.7: Performance averaged over 10 runs for various learning rate methods.

Figure 8.7 shows that the fast fixed learning rate has the best initial scores, but the performance trace for that option shows that it never reaches accurate values. It remains unstable and well below the other traces from about sequence 50 onwards. Figure 8.7 shows that temporal coherence has the best overall performance. It is almost as fast as the unstable learning rate initially, achieves a closer final approach at the end of the runs than any of the other methods tested, and is either closer to the right values or reaches them sooner than the other methods. These conclusions are emphasised by Figure 8.8, which compares the trace of weight B from Figures 8.4, 8.5 and 8.6.

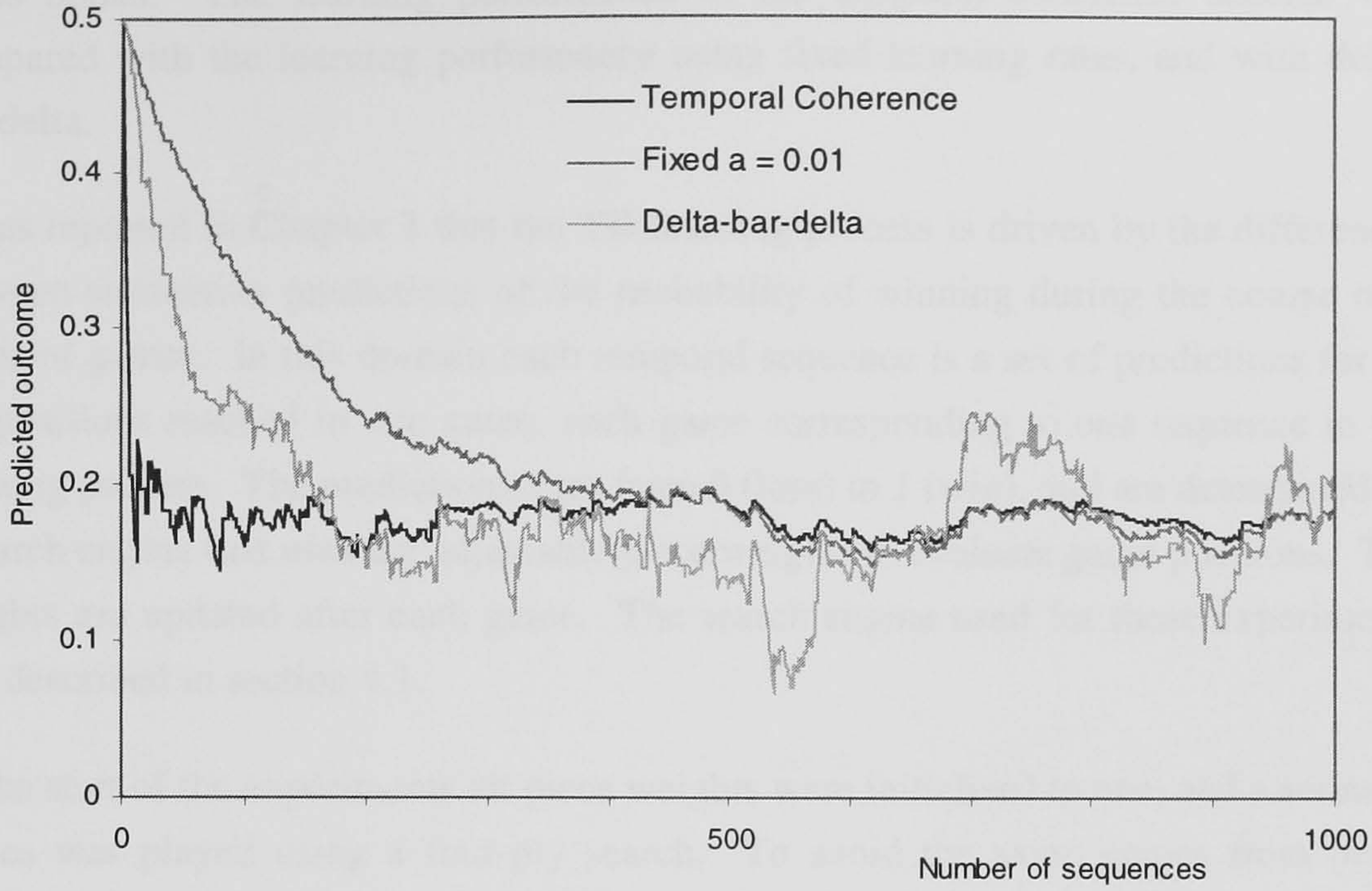

Figure 8.8: Weight trace B compared from three different learning rate methods.

In this domain, unlike the more complex domains of chess and shogi, the value of the weights does not have any effect on the sequences. For this reason, a given seed for the random number generator ensures that each of the learning methods will have exactly the same sequences to learn from. From Figure 8.8 it can be seen that, especially towards the ends of the runs, the peaks and troughs in the weight traces match up, and represent similar learning adjustments in response to the training sequence.

8.6 Test Domain Two: Learning the Values of Chess Pieces

In addition to the simple bounded walk problem, we also tested the new methods in the more complex domain of chess. As in Chapter 5, the chosen task was the learning of the values of chess pieces by a minimax search program in the absence of any chess-related initial knowledge other than the rules of the game.

The task was to learn suitable values for five adjustable weights (pawn, knight, bishop, rook and queen), via a series of randomised self-play games. Learning from self-play has the important advantage that no existing expertise (human or machine) is assumed, and thus the method is transferable to domains where no existing expertise is available. In Chapter 5 we showed that using this method it is possible to learn relative values of the pieces that perform at least as well as those quoted in elementary

chess books. The learning performance of the temporal coherence scheme was compared with the learning performance using fixed learning rates, and with delta-bar-delta.

It was reported in Chapter 3 that the TD learning process is driven by the differences between successive predictions of the probability of winning during the course of a series of games. In this domain each temporal sequence is a set of predictions for all the positions reached in one game, each game corresponding to one sequence in the learning process. The predictions vary from 0 (loss) to 1 (win), and are determined by a search engine that uses the adjustable piece weights to evaluate game positions. The weights are updated after each game. The search engine used for these experiments was described in section 4.1.

At the start of the experiments all piece weights were initialised to one, and a series of games was played using a four-ply search. To avoid the same games from being repeated, the move lists were randomised. This had the effect of selecting at random from all tactically equal moves with the added benefit of ensuring that a wide range of different types of position was encountered.

8.6.1 Results from single runs

To visualise the results obtained from the various methods for determining learning rates, we present graphs produced by plotting the weight movements for each of the five piece values over the course of runs each consisting of 2,000 game sequences. As the absolute values of the piece weights are unimportant compared with their *relative* values, the graphs are normalised so that the average value of the pawn weight over the last 10% of the game sequences is 1. This enables comparison with the widely quoted elementary values of 1:3:3:5:9. As in the bounded walk domain, the number of sequences in each run is large enough for the values to reach a quasi-stable state of random noise around a learnt value. To confirm that the apparent stability is not an artefact, each experiment was repeated 10 times, using different random number seeds.

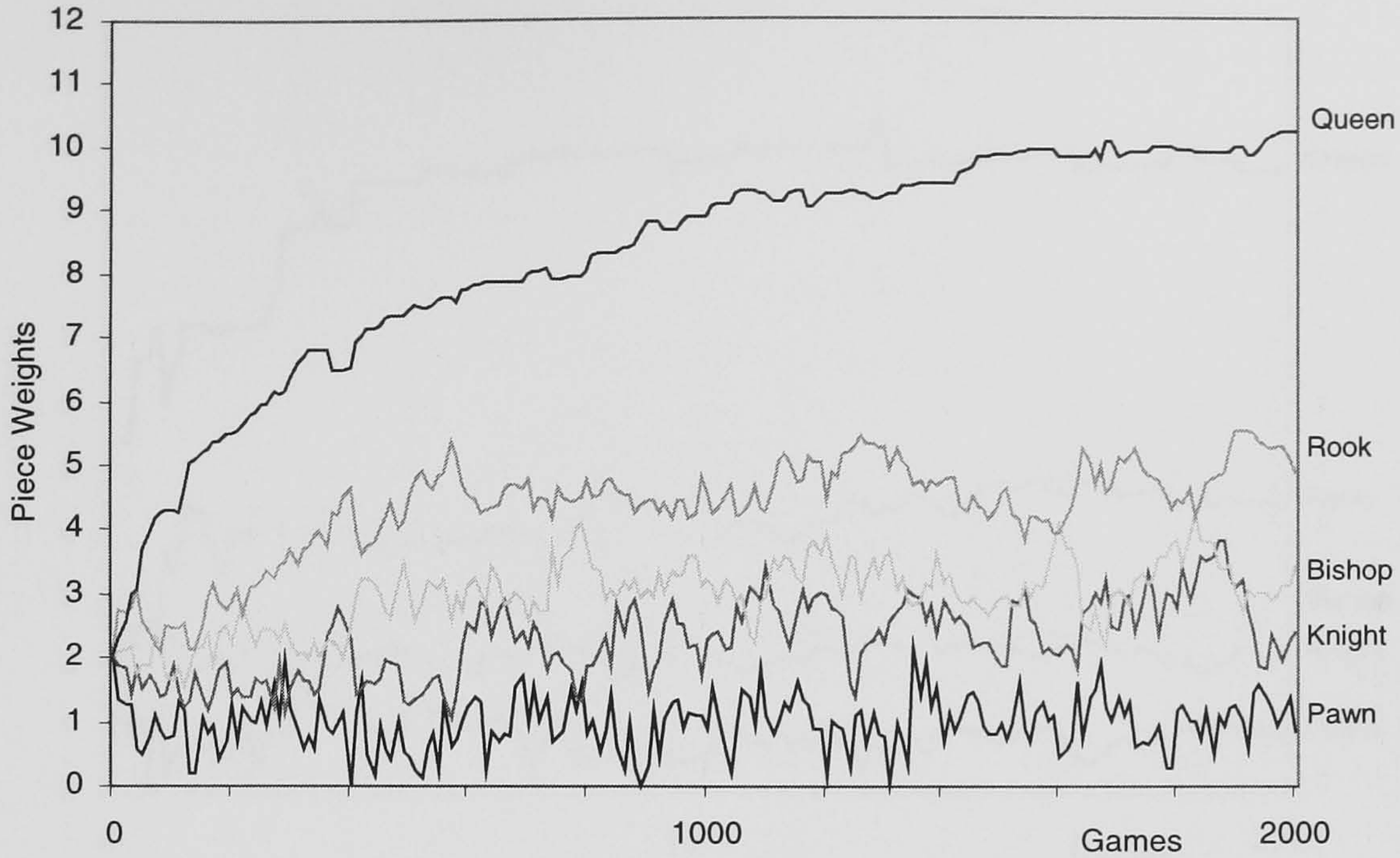

Figure 8.9: Weight movements from a typical single run using fixed α=0.05.

Figure 8.9 shows the weights achieved using fixed settings of α = 0.05 and λ=0.95 over a typical single run. In the experiments of Chapter 5 we found that these settings offered a good combination of learning rate and stability from the many fixed settings that were tried. A lower learning rate produced more stable values, but at the cost of further increasing the number of sequences needed to establish an accurate set of relative values. Raising the learning rate made the weights increasingly unstable.

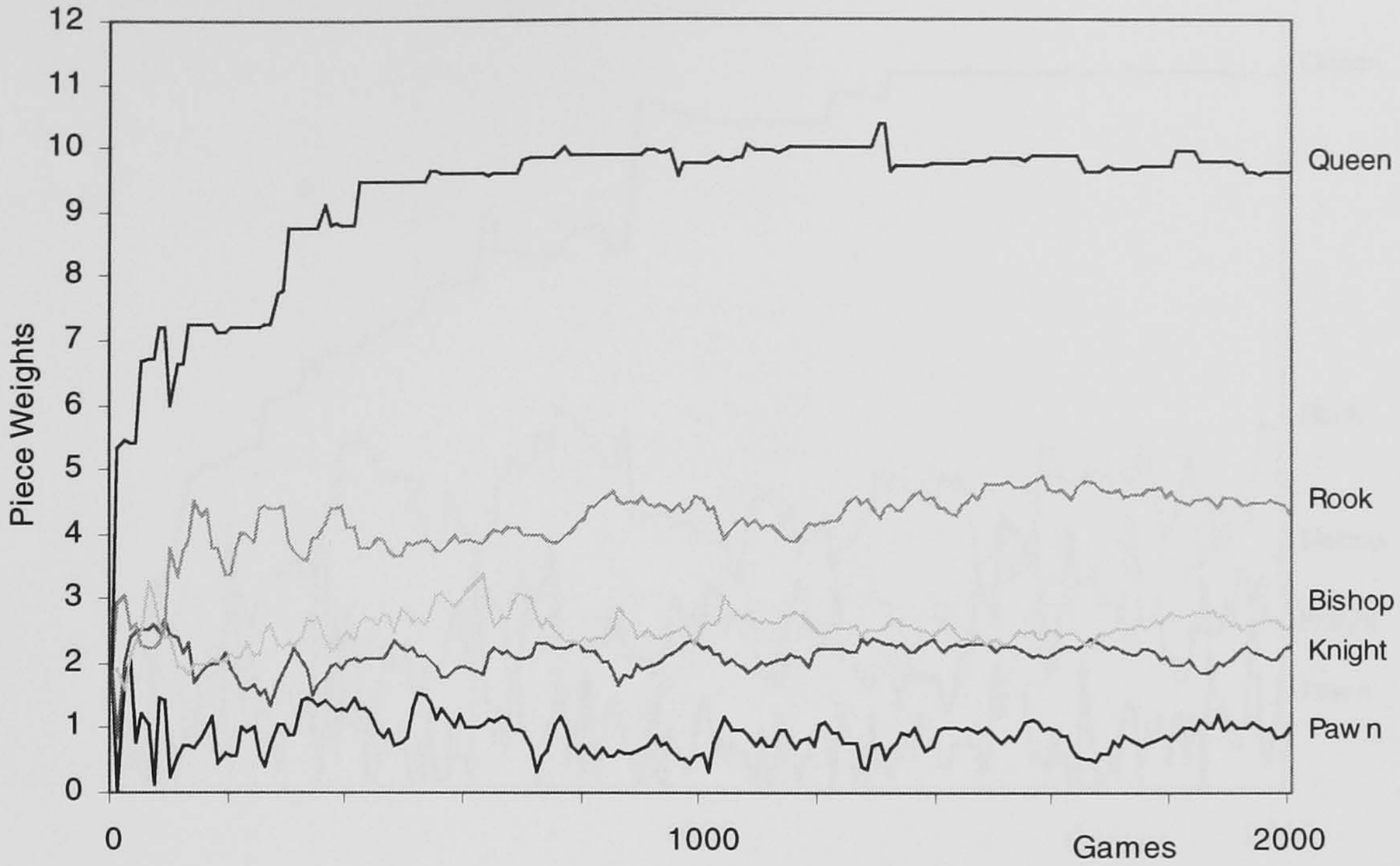

Figure 8.10: Weight movements from a typical single run using temporal coherence.

Figure 8.10 shows the weights learnt by the temporal coherence system over a typical single run. In this Figure we can see that the learning process is essentially complete after 500 sequences and that the weights remain fairly stable for the remainder of the sequences. This was typical of all runs, as shown later in Figure 8.11.

The pieces values are much more stable in Figure 8.10 than Figure 8.9, and the relative ordering of the pieces is consistent over the length of the run. In addition, the speed of learning is significantly faster using temporal coherence than in the fixed learning rate run where a significant amount of the learning occurring after 500 sequences.

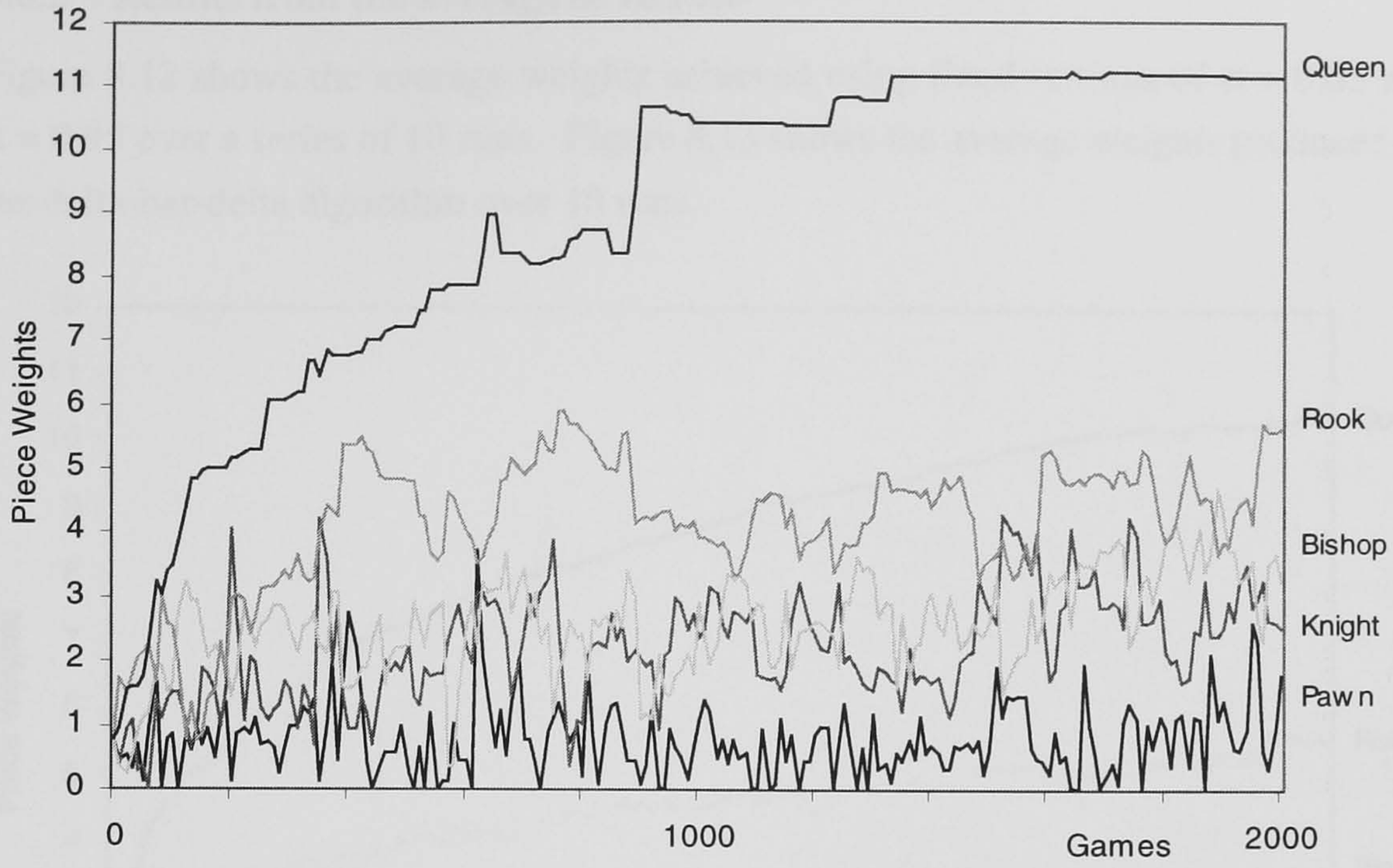

Figure 8.11: Weight movements from a typical single run using delta-bar-delta.

Figure 8.11 shows the results achieved from a typical run using delta-bar-delta. Comparing this Figure with Figure 8.10, we can see that the weights produced using DBD are much less stable than those produced by TC, and the relative ordering of the pieces is not consistent. For this domain we used meta-parameters of: $\kappa = 0.035$, $\phi = 0.333$, $\theta = 0.7$, and $\varepsilon_0 = 0.05$, guided by data presented by Jacobs (1988) and preliminary experiments in this domain. For λ, which DBD does not set, we used λ=0.95 derived from our experience with the fixed rate runs. We tried a number of other meta-parameter settings, none of which performed better. It is possible that a comprehensive search for a better set of meta-parameters might have improved the performance of the delta-bar-delta algorithm, but given the computational cost of a single run of 2,000 sequences, we were unable to attempt a systematic search of all the meta-parameter values. Other runs with different random seeds showed similar behaviour to Figure 8.11.

Figures 8.9 to 8.11 showed typical results obtained from single runs, determined by a random number seed. We repeated the experiment ten times, using ten different seeds. Figures 8.12 to 8.14 show averaged weight movements, to confirm that the characteristics seen in the single runs represent consistent behaviour.

8.6.2 Results from the average of 10 runs

Figure 8.12 shows the average weights achieved using fixed settings of $\alpha = 0.05$ and $\lambda = 0.95$ over a series of 10 runs. Figure 8.13 shows the average weights produced by the delta-bar-delta algorithm over 10 runs.

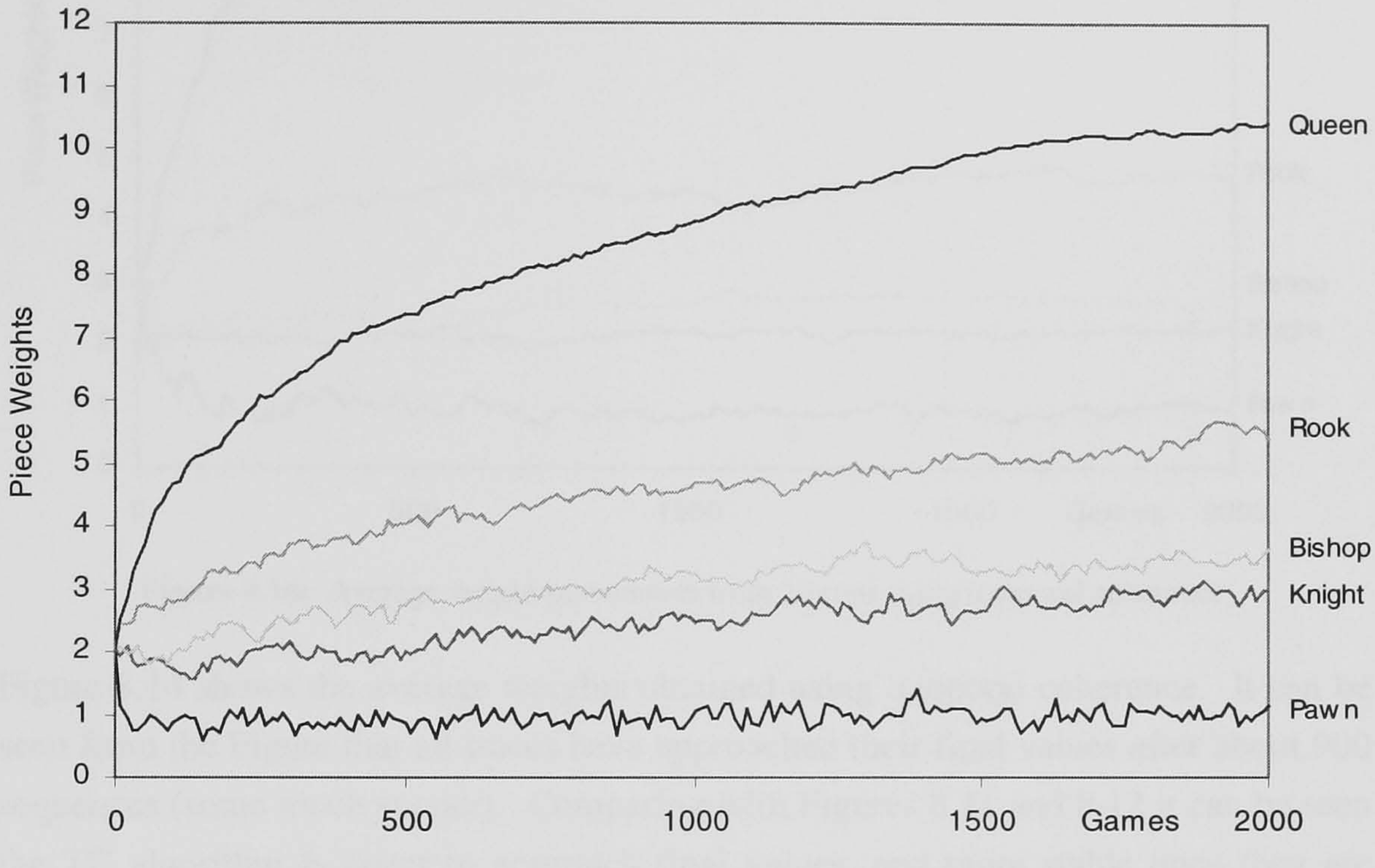

Figure 8.12: Average weight movements from 10 runs using a fixed α of 0.05.

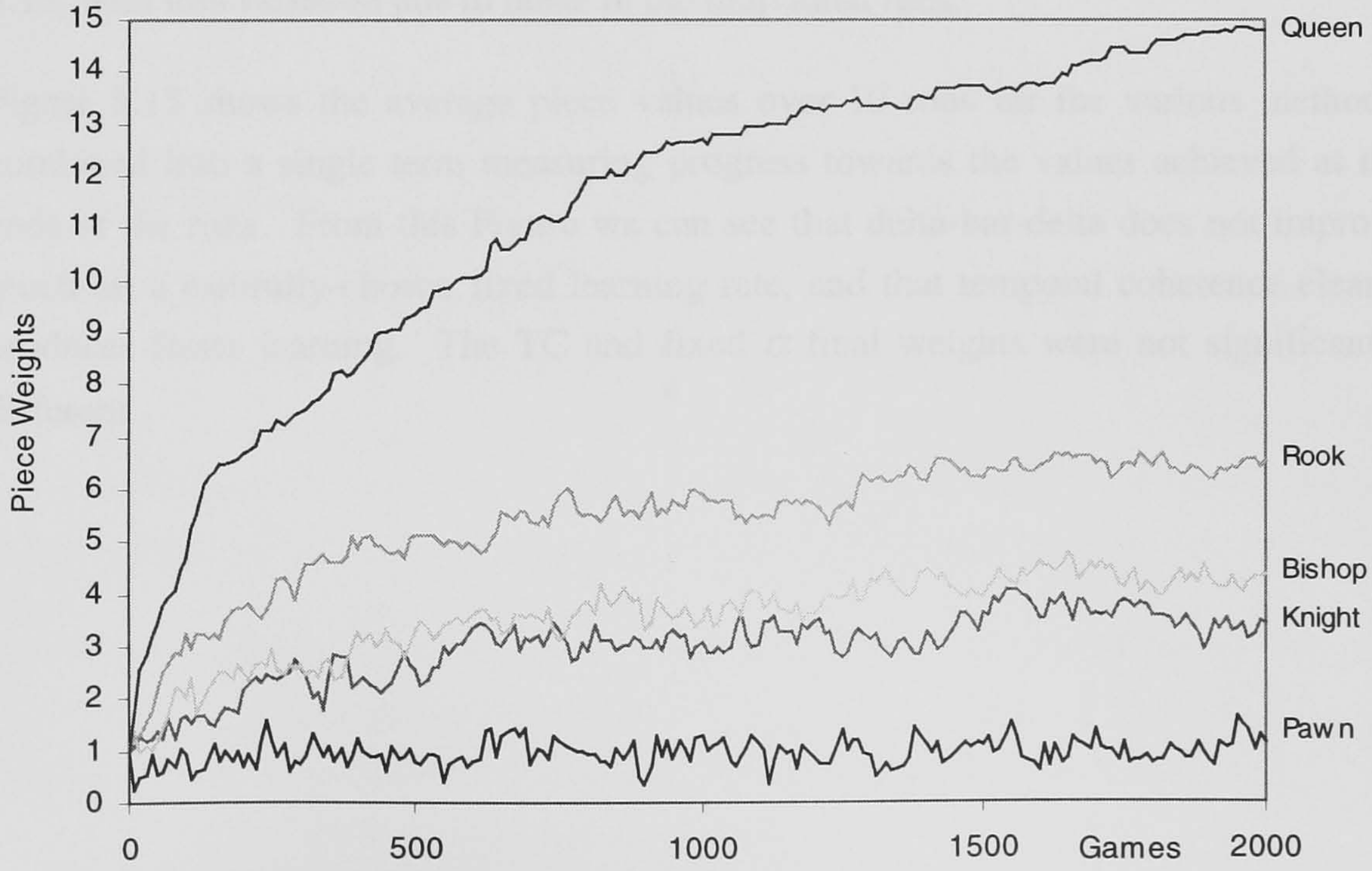

Figure 8.13: Average weight movements from 10 runs using delta-bar-delta.

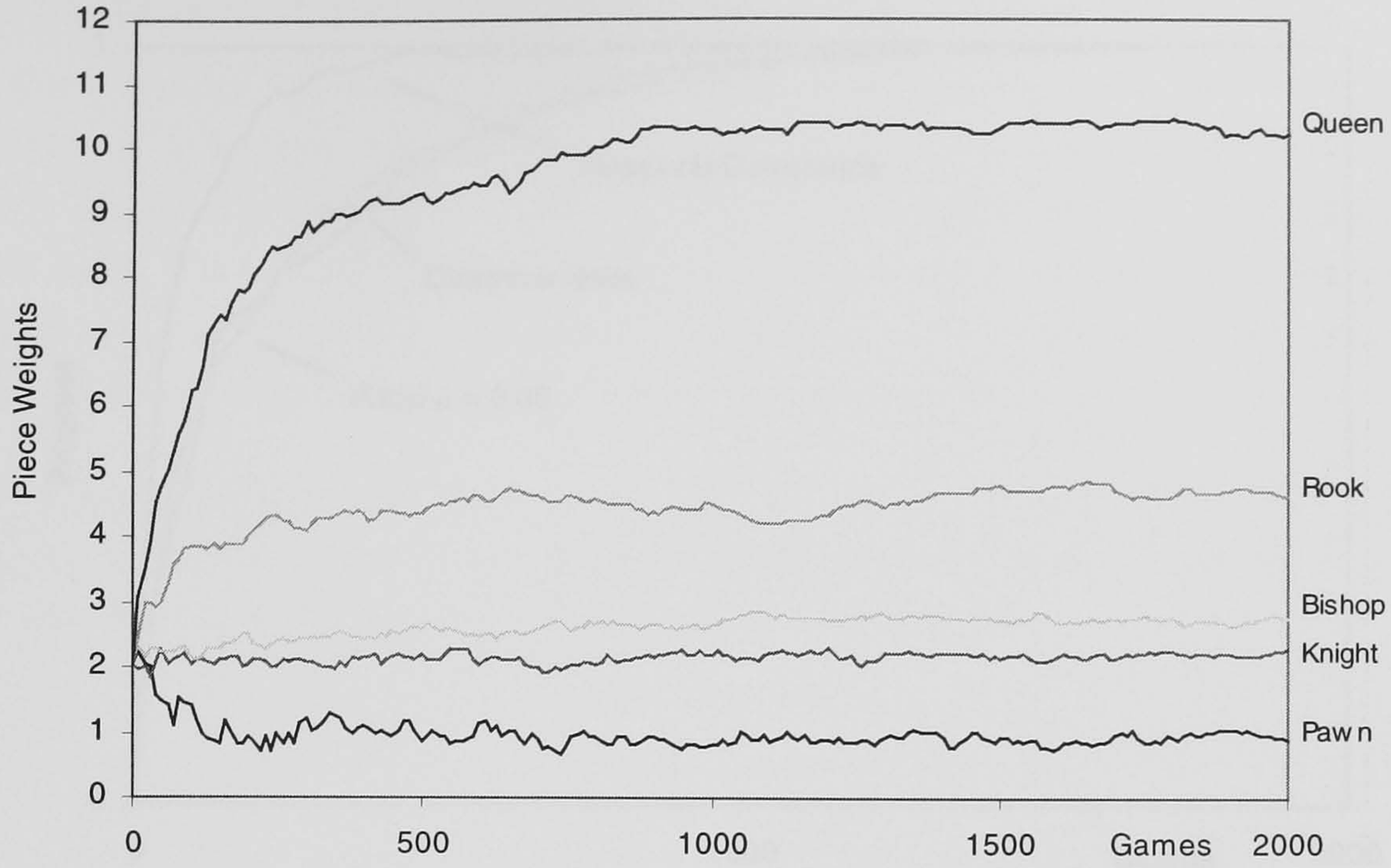

Figure 8.14: Average weight movements from 10 runs using temporal coherence.

Figure 8.14 shows the average weights obtained using temporal coherence. It can be seen from the Figure that all traces have approached their final values after about 900 sequences (some much sooner). Comparing with Figures 8.11 and 8.12 it can be seen the TC algorithm is faster to approach final values, and more stable once they are reached. In addition, the traces in Figure 8.14 are smoother than in Figures 8.12 and 8.13, with less variation due to noise in the individual runs.

Figure 8.15 shows the average piece values over 10 runs for the various methods, combined into a single term measuring progress towards the values achieved at the ends of the runs. From this Figure we can see that delta-bar-delta does not improve much on a carefully-chosen fixed learning rate, and that temporal coherence clearly produces faster learning. The TC and fixed α final weights were not significantly different.

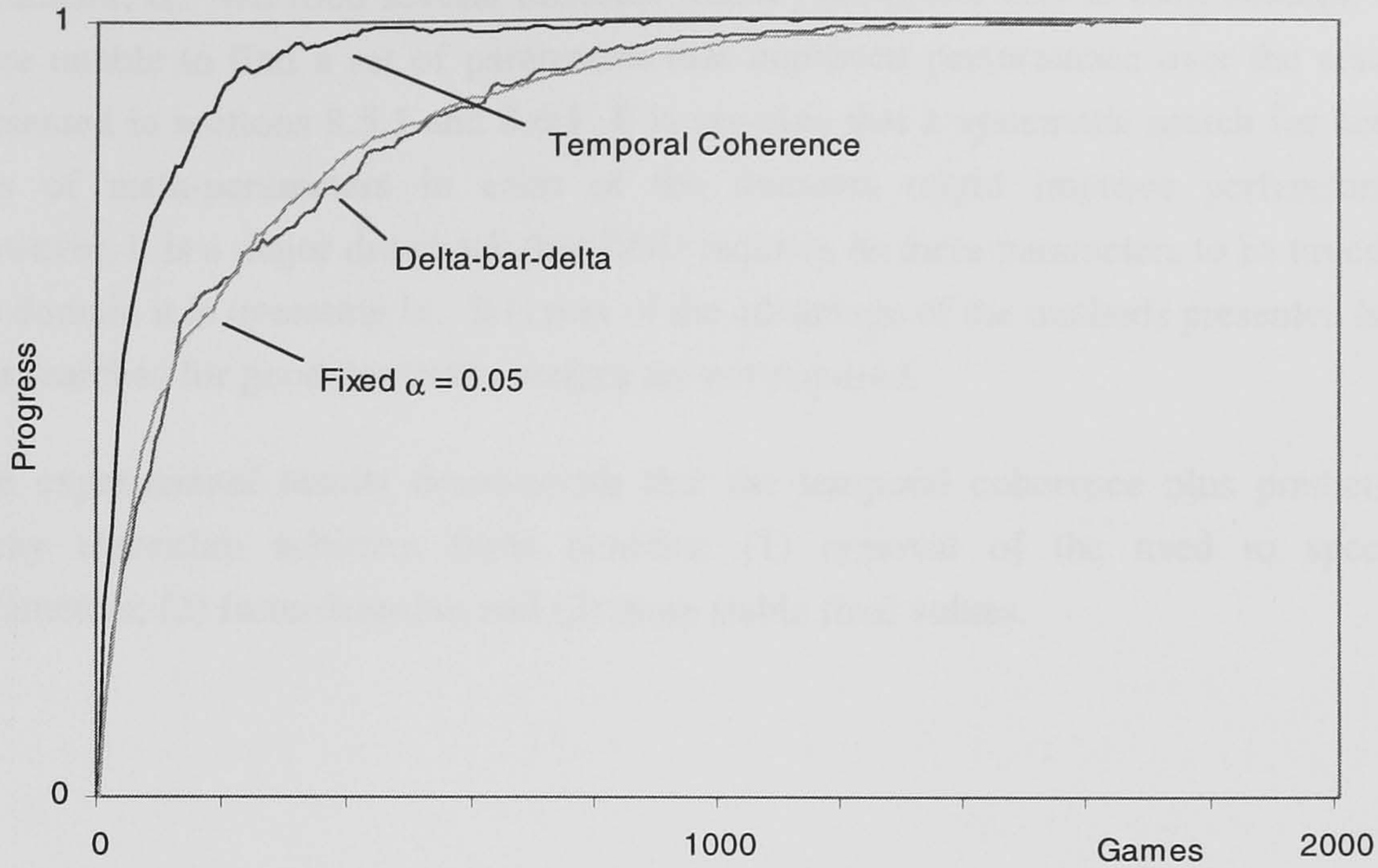

Figure 8.15: Progress in the chess domain averaged over 10 runs.

To confirm that the learning process had produced satisfactory values, a match was played pitting the learnt values against the set 1:3:3:5:9, in the manner reported in Chapter 5. One program used the standard values, the other used the weights learnt using temporal coherence, as a check that the learnt values were at least as good (under our test conditions) as the standard values. In a match of 2,000 games, the TC values achieved a score of 58% (won 1,119; lost 781; drawn 100).

8.7 Discussion

This Chapter described new extensions, temporal coherence, and prediction decay, to the temporal difference learning method that set the major control parameters, learning rate and temporal discount, automatically as learning proceeds. The resulting temporal difference algorithm does not require initial settings for α and λ, and has been tested in depth on two domains.

Results from the two domains demonstrated both faster learning and more stable final values than a previous algorithm and the best of the fixed learning rates. In these test domains values were learnt without supplying any domain-specific knowledge.

In our comparisons with the delta-bar-delta algorithm, we tried to find good parameter sets for DBD, which requires four meta parameters instead of the one control

parameter, α. We tried several different (meta-) parameter sets in each domain, but were unable to find a set of parameters that improved performance over the results presented in sections 8.5.1 and 8.6.1 It is possible that a systematic search for better sets of meta-parameters in each of the domains might improve performance. However, it is a major drawback that DBD requires its meta-parameters to be tuned to the domain it is operating in. It is part of the advantage of the methods presented here that searches for good parameter values are not required.

The experimental results demonstrate that the temporal coherence plus prediction decay algorithm achieves three benefits: (1) removal of the need to specify parameters; (2) faster learning and (3) more stable final values.

9 CONCLUSIONS

The aim of this thesis was to adapt and improve the temporal difference learning methods that were used successfully for backgammon and apply them to other complex games that require search for high-level play. A secondary aim was for the learning to occur with as little input of external knowledge as possible.

In Chapter 3 we described how Sutton's TD(λ) could be applied to minimax searches, involving the introduction of a squashing function to apply to the evaluation of the search's principal position.

The main experimental platform was introduced in Chapter 4, and a number of sophisticated enhancements to the basic platform were discussed. The work described in this Chapter resulted in an efficient and robust search engine that was used for most of the experiments described in the remainder of the thesis.

The learning method described in Chapter 3 was applied successfully to the complex domains of chess and shogi in Chapters 5 and 6. In both domains our methods were successful in learning piece values that compared well with human chosen values. The aim of learning about the chosen domains with minimal knowledge input was achieved by the use of randomised self-play, and no external knowledge was required for the learning to succeed.

In Chapter 7 experiments to learn more complex weight sets were described. Even when the number of weights being learnt was increased to over 300, the methods of Chapter 3 were successful in learning sensible and effective values. The results of the experiments conducted at MIT by Don Dailey show that our methods can be successful in competitive programs playing at the highest level.

Chapter 8 presented a major research contribution in which we described a novel extension to Sutton's TD(λ) which automatically sets and adjusts TD(λ)'s two major control parameters. In both a simple random-walk state-learning task and a complex game our methods of temporal coherence and prediction decay were shown to produce both faster learning and more stable final values than carefully chosen fixed learning rates. The faster learning is important because all the learning in the complex domains of Chapters 5 and 6 required significant computational effort. Our methods also performed better than an alternative method of learning rate adjustment described in the literature.

9.1 Application Areas

In Chapter 8 we showed how the use of Temporal Coherence and Prediction Decay leads to faster learning. In theoretical terms it is possible to make a distinction between better learning (i.e. producing better results) and faster learning (producing the same results in less time). In practice, faster learning will often lead directly to better learning, because reducing the amount of time taken by each trial leaves more time to experiment with different algorithms and architectures. In the literature, typically only results from the most successful experiments are presented, with the authors explaining that the architecture and/or algorithms used were arrived at after some experimentation. It has been our experience that such "preliminary experiments" often have a computational cost far in excess of that of the published results. The use of Temporal Coherence and Prediction Decay not only enables faster learning during `production' runs but also greatly reduces the computation cost of any preliminary, more experimental, trials. This reduction in computational cost allows more effort to be invested in experimentation with alternative algorithms, architectures and feature sets, potentially leading to greatly improved final results.

Temporal Coherence can deliver its advantage anywhere reinforcement learning systems can be used. Prediction Decay can be utilised wherever TD(λ) is used.

Presented below are a number of practical real world domains in which the use of Temporal Coherence and/or Prediction Decay could be expected to result in significant improvement, not only in speed of learning, but also in improved outcomes due to exploring more of the solution spaces.

9.1.1 Elevator Dispatching

Elevator dispatching is a difficult real-world problem that has seen the successful application of reinforcement learning techniques. The elevator domain is especially challenging because elevator systems operate in continuous state spaces and in continuous time as discrete event dynamic systems. Their states are not fully observable and they are non-stationary due to changes in the rate with which new passengers arrive.

An example of the elevator domain (Crites and Barto 1996) consists of a 10-story building with 4 elevator cars, and a passenger profile which dictates arrival rates for every 5 minute interval during a typical afternoon rush hour. Each car has a small set of primitive actions. The performance objective in this example is to minimise the

sum of the squared wait times (the time between the arrival of a passenger and his entry into a car).

The state space for this problem is continuous because it includes the elapsed times since elevator calls were registered, which are real-valued. Even if these real values were approximated as discrete values, Crites and Barto (1996) estimate that the state space would have at least 10^{22} states, making solving this problem by classical dynamic programming methods completely impractical.

Crites and Barto (1996) approached this problem by using a team of reinforcement learning (RL) agents, one controlling each elevator car. The team of agents received a global reinforcement signal which appears noisy to each agent due to the effects of the actions of the other agents, the random nature of the arrivals and the incomplete observation of the state. The elevator system events occur randomly in continuos time, which complicates the use of algorithms that require explicit lookahead as the branching factor is effectively infinite. For this reason Crites and Barto (1996) utilised a team of discrete-event Q-learning agents, with each agent having responsibility for controlling a single elevator car. $Q(x,a)$ is defined as the expected return obtained by taking action a in state x and then following an optimal policy (Watkins, 1989). The Q-values were stored in feed-forward neural networks which received some state information as input, and produced Q-value estimates as output.

For the example described above, Crites and Barto (1996) achieved their best results using networks with 47 hand chosen input units (features), 20 hidden units with sigmoid activation functions, and 2 linear output units. After every training decision the agent's estimate of $Q(x,a)$ was adjusted toward the target output by error backpropagation. The learning rate parameter was set to either 0.01 or 0.001. Crites and Barto presented results that surpassed the performance of the best heuristic elevator control algorithms, but found that it required "considerable experimentation" to achieve their best results.

Temporal Coherence should significantly reduce the computational cost of training, allowing better network architectures to be identified for the same computational cost by using the computational savings for additional experiments. Using fixed learning rates, each RL controller (of which there were 10) was trained by Crites and Barto on 60,000 hours of simulated elevator time, taking four days on a 100 MIPS machine. This was just the time taken for the best, presented, results. As these results were only achieved after "considerable experimentation", presumably equivalent experiments were performed many other times with different network architectures

and algorithmic parameters. Crites and Barto (1996) state "Although this [four days on a 100 MIPS processor] is a considerable amount of computation, it is negligible compared to what any conventional dynamic programming algorithm would require."

9.1.2 Job-shop Scheduling

Many tasks in manufacturing industries require job-shop scheduling. The goal is to schedule a set of tasks to satisfy a set of temporal and resource constraints while also seeking to minimise the total duration of the schedule. It is an example of an important industrial domain where temporal difference learning is very effective.

The NASA space shuttle payload processing (SSPP) domain (Zweben et al., 1994; Zhang and Dietterich, 1995, 1997) requires scheduling the various tasks that must be performed to install and test the payloads that are placed in the cargo bay of the space shuttle. The method regularly used at the Kennedy Space Center is an iterative repair-based scheduling procedure that combines a set of heuristics with a simulated annealing search procedure (Zweben et al., 1994).

A typical SSPP problem involves the simultaneous scheduling of between two and six shuttle missions, with each mission involving between 32 and 164 tasks (Zhang and Dietterich, 1995). This results in scheduling problems containing several hundred tasks. Most of these tasks must be performed prior to launch, but some also take place after the shuttle has landed. Because every shuttle mission has a fixed launch date, but no starting date or ending date, tasks required prior to launch have deadlines but no ready times, and tasks required after landing have ready times but no deadlines. A primary goal of the scheduling system is to minimise the total duration of the schedule. As Zhang and Dietterich (1995) observe, this is a much more demanding problem than simply finding a feasible schedule.

Zhang and Dietterich (1995, 1997) applied TD(λ) to train a neural network to learn an heuristic evaluation function for problems from a SSPP task. They experimented with two different network architectures. In their earlier experiments (1995) they used a feed-forward network with 40 sigmoidal hidden units and 8 sigmoidal output units. They trained eight different network using all combinations of: learning rate = 0.1 or 0.05, λ = 0.2 or 0.7 and two values for their probabilistic exploration schedule. They do not provide any explanation for their choice of parameters other than observing that preliminary experiments showed that λ = 0 did not perform as well.

In their subsequent experiments Zhang and Dietterich (1996) used a larger and more complicated time-delay neural network which included 3 hidden layers with a total of 1123 adjustable parameters. Various experiments were conducted where λ was fixed at 0.2 and 0.7 Their best results were achieved using this more complicated network, which generally took around 10,000 training iterations before its performance stopped changing.

This domain is another in which the use of Temporal Coherence and Prediction Decay would be expected to produce a significant reduction in the computational cost of individual training runs, allowing more time for experimentation with other architectures and methods. Zhang and Dietterich found that their more complex network produced better results, and had they been able to reduce their training times by the use of Temporal Coherence and Prediction Decay it is quite possible that they would have discovered a more effective network architecture.

9.1.3 Dynamic Channel Allocation Strategies

An important problem in cellular communication systems is to allocate the available communication resource (bandwidth) so as to maximise service in an environment where demand changes stochastically. The geographical area covered by the service is divided up into separate cells, where each cell serves the calls that are within its boundaries. The total system bandwidth is divided into channels, with each channel centred around a frequency. Each channel can be used simultaneously by different cells, providing that the cells are separated by enough geographical distance to ensure that there is no interference between them. The minimum separation distance between simultaneous reuse of the same channel is called the channel reuse constraint (Singh and Bertsekas, 1997).

When a request for a call is made to the system, the cell responsible either allocates a free channel (one that does not violate the channel reuse constraint) or else the call is blocked from the system. An additional complexity is introduced when a mobile caller's physical location moves from one cell to another. In this case, responsibility for the call is passed on to the newly entered cell, which must itself allocate a channel to the call to prevent it from being disconnected from the system. One objective of a channel allocation strategy is to minimise the number of blocked calls. Another objective is to minimise the number of calls that are disconnected when they are passed on to an already busy cell. This second objective is often given a higher weighting, as disconnecting existing calls is usually considered more undesirable than blocking new calls (Singh and Bertsekas, 1997).

In practice, many cellular systems use a Fixed Assignment (FA) channel allocation strategy. This means that the set of channels is divided up and allocated to cells in such a way that all cells are able to use all channels allocated to them simultaneously without interference. When a call arrives in a cell it is blocked unless there is an unassigned channel available. This strategy is static and as such is unable to take advantage of any temporal variations in demand for service, and therefore is less efficient than dynamic channel allocation strategies. Dynamic strategies assign channels to different cells so that every channel is available to every cell unless the channel reuse constraint is violated. An example of a dynamical channel allocation strategy is Borrowing with Directional Channel Locking (BDCL), introduced by Zhang and Yum (1989) and shown by them to be superior to its competitors, including FA.

Singh and Bertsekas (1997) formulated the channel assignment problem as a dynamic programming problem, but note that it is too complex to be solved exactly. For this reason they introduced approximations based on the methods of reinforcement learning, using Sutton's (1988) temporal difference algorithm TD(0) to learn approximations to the optimal value function. Singh and Bertsekas presented results from a simulation of a large cellular system with approximately 70^{49} states, and showed that TD(0) with a linear function approximator was able to find better channel allocation policies than the BDCL and FA strategies.

The reinforcement learning system used by Singh and Bertsekas (1997) utilised a linear neural network, and took two sets of features as input. The first set of input features was the number of free channels in each cell, the second measured, for each cell-channel pair, the number of times that channel is used in a 4 cell radius. They found that using non-linear neural networks as function approximators did improve performance in some cases, but at a cost of a large increase in training time. Sutton (1999) highlights four open theoretical questions in reinforcement learning that seem "particularly important, pressing or opportune." One of these open theoretical questions concerns the use of λ by reinforcement learning methods. Sutton presents a collection of empirical results in which λ was varied from 0 to 1. In all cases, as in the experiments presented in this thesis, the best performance was found at an intermediate value of λ. Similar results have been shown analytically by Singh and Dayan (1988) but only for particular tasks and initial settings. Sutton (1999) comments that there is no proof as yet that the use of $0 < \lambda < 1$ is better, but there is a lot of evidence. However Sutton suggests no method for determining or adjusting λ.

Singh and Bertsakas (1997) did not specify the learning rate they used. All of their experiments were conducted using TD(0), i.e. $\lambda = 0$. It is likely that the use of Temporal Coherence to set and adjust the learning rate, and Prediction Decay to determine a value for λ, would have reduced their training time and so allowed for more experimentation with non-linear networks.

9.2 Possible Future Work

The previous section discussed several domains in which the use of Temporal Coherence and/or Prediction Decay could be expected to result in significant improvements. Other directions for possible future work are outlined below.

The application of the methods described in this thesis to fully-featured competitive game-playing programs is potentially very promising. The work of Don Dailey at MIT described in Chapter 7 suggests that weight sets can be found using our methods that perform significantly better than expertly hand-tuned weight sets. This might be of particular use in games such as shogi where much less effort has been invested in choosing weights for evaluation terms.

Preliminary experiments we conducted with the shogi engine suggest that the learning of piece-square weights for shogi is feasible, and that additional *king-proximity* tables are useful for ensuring that the short-ranged pieces common in shogi do not wander too far from the 'action'.

Learning values for piece combinations rather than the individual pieces themselves might be an interesting task. For example, in chess the combination of queen and knight is often considered stronger than queen and bishop, even though individually a bishop is usually more valuable than a knight.

As a step beyond generating weights for given terms, the problem of *identifying* suitable evaluation terms remains as a research frontier. The two-level neural-net approach that was so successful in backgammon (Tesauro 1992, 1994) does not seem likely to work well in complex domains such as chess and shogi, where search based tactical expertise is required.

REFERENCES

Almeida, L.B., Langlois, T., Amaral, J.D. and Plakhov, A. (1998) On-Line Step Size Adaptation. *Technical Report RT07/97 INESC*, 9 Rua Alves Redol, 1000 Lisbon, Portugal.

Anantharaman, T.S., Campbell, M.S. and Hsu, F. (1988) Singular Extensions: adding selectivity to brute force searching. *International Computer Chess Association Journal*, vol. 11, no. 4, pp. 135-143.

Anantharaman, T.S. (1991a) Confidently Selecting a Search Heuristic. *International Computer Chess Association Journal*, vol. 14, no. 1, pp. 3-16.

Anantharaman, T.S. (1991b) Extension Heuristics. *International Computer Chess Association Journal*, vol. 14, no. 2, pp. 47-65.

Anantharaman, T.S. (1997) Evaluation Tuning for Computer Chess: Linear Discriminant Methods. *International Computer Chess Association Journal*, vol. 20, no. 4, pp. 224-242.

Babbage, C. (1864) *Passages on the Life of a Philosopher*. Longman, London.

Baxter, J., Tridgell, A. and Weaver, L. (1998) KnightCap: A chess program that learns by combining TD(λ) with game-tree search. In *Machine Learning, Proceedings of the Fifteenth International Conference (ICML '98)*, Madison, pp. 28-36.

Beal, D. F. (1984) Mating Sequences in the Quiescence Search. *International Computer Chess Association Journal*, vol. 7, no. 3, pp. 133-137.

Beal, D.F. (1989) Experiments with the Null Move. In Beal, D.F. (ed) *Advances in Computer Chess 5*, North-Holland, pp.65-79.

Beal, D.F. and Smith, M.C. (1994) Random Evaluations in Chess. *International Computer Chess Association Journal*, vol. 17, no. 1, pp. 3-9.

Beal, D.F. and Smith, M.C. (1995) Quantification of Search Extension Benefits. *International Computer Chess Association Journal*, vol. 18, no. 4, pp. 205-218.

Beal, D.F. and Smith, M.C. (1996) Multiple Probes of Transposition Tables. *International Computer Chess Association Journal*, vol. 19, no. 4, pp. 227-233.

Beal, D.F. and Smith, M.C. (1997) Learning piece values using temporal differences. *International Computer Chess Association Journal*, vol. 20, no. 3, pp. 147-151.

Beal, D.F. and Smith, M.C. (1998a) Temporal Coherence and Prediction Decay in Temporal Difference Learning. *Technical report #756*, Dept. of Computer Science, Queen Mary and Westfield College, University of London.

Beal, D.F. and Smith, M.C. (1998b) First Results from using Temporal Difference Learning in Shogi. In van den Herik, H. and Iida, H. (eds.) *Proceedings of the First International Conference on Computers and Games (CG'98)* Springer-Verlag, Berlin, pp. 113-125.

Beal, D.F. and Smith, M.C. (1999a) Temporal Difference Learning Applied to Game Playing and the Results of Application to Shogi. To appear in *Theoretical Computer Science.*

Beal, D.F and Smith, M.C. (1999b) Temporal Coherence and Prediction Decay in TD Learning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'99)*, Morgan Kaufman, pp. 564-569.

Beal, D.F. and Smith, M.C. (1999c) Temporal Difference Learning for Heuristic Search and Game Playing. To appear in *Information Sciences Journal.*

Berliner, H.J. (1979) The B* Tree Search Algorithm: A Best First Proof Procedure. *Artificial Intelligence*, vol. 12, no. 1, pp. 23-40.

Berliner, H.J., Goetsch, G., Campbell, M. and Ebeling, C. (1989) Measuring the Performance Potential of Chess Programs. In Beal, D.F. (ed.) *Advances in Computer Chess 5*, North-Holland, pp. 13-29.

Berliner, H.J., Kopec, D. and Northam, E. (1991) A Taxonomy of Concepts for Evaluating Chess Strength. In Beal, D. F. (ed.) *Advances in Computer Chess 6*, Ellis Horwood, pp. 179-191.

Berliner, H.J. (1989) Some Innovations Introduced by Hitech. In Beal, D.F. (ed.) *Advances in Computer Chess 5*, North-Holland, pp. 284-289.

Breuker, D.M., Uiterwijk, J.W.H.M., and van den Herik, H.J. (1994) Replacement Schemes for Transposition Tables. *International Computer Chess Association Journal*, vol. 17, no. 4, pp. 183-193.

Breuker, D.M., Uiterwijk, J.W.H.M., and van den Herik, H.J. (1996) Replacement Schemes and Two-Level Tables. *International Computer Chess Association Journal*, vol. 19, no. 3, 175-180.

Capablanca, J.R. (1921) *Chess Fundamentals,* G. Bell and Sons Ltd., London.

Christensen, J. and Korf, R. (1986) A Unified Theory of Heuristic Evaluation Functions and its Application to Learning. *AAAI-86*, Morgan-Kaufman, pp. 148-152

Crites, R.H., and Barto, A.G. (1996) Improving Elevator Performance Using Reinforcement Learning. In *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, MIT Press, Cambridge MA, pp. 1017-1023.

Dayan, P. (1992) The Convergence of TD(λ) for General λ. *Machine Learning*, vol. 8, pp. 341-362.

Donninger, C. (1993) Null Move and Deep Search: Selective Search Heuristics for Obtuse Chess Programs. *International Computer Chess Association Journal*, vol. 16, no. 3, pp. 137-143

Ebeling, C. (1986) All the Right Moves: A VLSI Architecture for Chess. *Ph.D. thesis, Carnegie-Mellon University*, Pittsburgh, Pa. MIT Press, Cambridge, MA.

Fairbairn, J. (1989) *Shogi for Beginners*. Ishi Press International.

Frey, P.W. (1977) *Chess Skill in Man and Machine*. Springer-Verlag, Heidelberg.

Fürnkranz, J. (1996) Machine Learning in Computer Chess: The Next Generation. *International Computer Chess Association Journal*, vol. 19, no. 3, pp. 147-161.

George, M. and Schaeffer, J. (1991) Chunking for Experience. In Beal, D.F. (ed.) *Advances in Computer Chess 6*, Ellis Horwood, London, pp. 133-146.

Goetsch, G. and Campbell, M.S. (1990) Experiments with the Null-Move Heuristic. In Marsland, T.A. and Schaeffer, J. (eds.) *Computers, Chess and Cognition*, Springer-Verlag, New York, pp.159-168.

Hyatt, R.M., Gower, A.E. and Nelson, H.L. (1990) Cray Blitz. In Marsland, T.A. and Schaeffer, J. (eds.) *Computers, Chess and Cognition*, Springer-Verlag, New York, pp.111-130.

Jacobs, R. A. (1988) Increased Rates of Convergence Through Learning Rate Adaptation. *Neural Networks*, vol. 1, pp. 295-307.

Kaelbling, L.P., Littman, M.L., and Moore, A.W. (1996) Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, vol. 4, pp. 237-285.

Kaindl, H., Horacek, H. and Wagner, M. (1986) Selective Search versus Brute Force. *International Computer Chess Association Journal*, vol. 9, no. 3, pp. 140-145.

Keres, P. (1973) *Practical Chess Endings*, Batsford, London.

Knuth, D.E. (1973) *The Art of Computer Programming. Vol. 3: Sorting and Searching*. Addison-Wesley Publishing Company, Reading Massachusetts.

Knuth, D.E. and Moore, R.W. (1975) An analysis of alpha-beta pruning. *Artificial Intelligence,* vol. 6, no. 4, pp. 293-326.

Kopec, D. (1990) Advances in Man-Machine Play. In Marsland, T.A. and Schaeffer, J. (eds.) *Computers, Chess and Cognition*, Springer-Verlag, New York, pp.9-32.

Lang, K.J. and Smith, W.D. (1993) A Test Suite for Chess Programs. *International Computer Chess Association Journal*, vol. 16, no. 3, pp. 152-161.

Leggett, T. (1993) *Shogi: Japan's Game of Strategy*. Charles E. Tuttle Company. [Reprinted in 1993, first published in 1966].

Levene, M. and Fenner, T. (1995) A Partial Analysis on Minimaxing Game Trees with Random Leaf Values. *International Computer Chess Association Journal*, vol. 18, no. 1, pp. 20-33.

Levinson, R. and Snyder, R. (1991) Adaptive Pattern Oriented Chess. In *Proceedings of AAAI-91*, Morgan-Kaufman, pp. 601-605.

Levy, D., Broughton, D. and Taylor, M. (1989) The SEX Algorithm in Computer Chess. *International Computer Chess Association Journal*, vol. 12, no. 1, pp. 10-21.

Levy D. and Newborn, M. (1991) *How Computers Play Chess*. Computer Science Press, New York. ISBN 0-7167-8239-1

Marsland, T.A. and Rushton, P.G. (1973) Mechanics for Comparing Chess Programs, In *Proceedings of the ACM Annual Conference*, pp. 202-205.

Marsland, T.A. (1983) Relative Efficiency of Alph-Beta Implementations. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pp. 763-766.

Marsland, T.A. (1986) A Review of Game-Tree Pruning, *International Computer Chess Association Journal*, vol.9 no. 1, pp. 3-19.

Marsland, T.A. (1990) A Short History of Computer Chess. In Marsland, T.A. and Schaeffer, J. (eds.) *Computers, Chess, and Cognition*. Springer-Verlag, New York, ISBN 0-387-97415-6, pp. 3-7.

Marsland, T.A. (1992) Computer Chess and Search. In Shapiro, S. (ed.) *Encyclopaedia of Artificial Intelligence (2nd edition)*, J. Wiley & Sons.

Matsubara, H., Iida, H. and Grimbergen, R. (1996) Natural Developments in Game Research: From Chess to Shogi to Go. *International Computer Chess Association Journal*, vol. 19, no. 2, pp. 103-112.

McCarthy, J. (1990) Chess as the Drosophila of AI. In Marsland, T.A. and Schaeffer, J. (eds.) *Computers, Chess, and Cognition*. Springer-Verlag, New York, ISBN 0-387-97415-6, pp. 227-238.

Michie, D. (1990) Brute Force in Chess and Science. In Marsland, T.A. and Schaeffer, J. (eds.) *Computers, Chess, and Cognition*. Springer-Verlag, New York, ISBN 0-387-97415-6, pp. 239-258.

Mutz, M. (1994) *Gnu Shogi v1.2p03*. Available from many sources, including `ftp://ftp.uni.passau.de/pub/local/shogi`

Palay, A.J. (1983) The B* Tree Search Algorithm – New Results. *Artificial Intelligence*, vol 19, pp. 145-163.

Pitrat, J. (1998) Games: The Next Challenge. . *International Computer Chess Association Journal*, vol. 21, no. 3, pp. 147-156.

Pell, B. (1992) METAGAME: A New Challenge for Games and Learning. In van den Herik, H.J. and Allis, L.V. (eds.) *Heuristic Programming in Artificial Intelligence 3*, Ellis Horwood, England, ISBN 0-13-388265-9, pp. 237-251.

Plaat, A., Schaeffer, J., Pijls, W., and de Bruin, A (1994) A New Paradigm for Minimax Search. *Technical Report 94-18*, Department of Computing Science, University of Alberta, Edmonton, Canada.

Pritchard, D.B. (1994) *The Encyclopaedia of Chess Variants.* Games and Puzzles Publications, Surrey, United Kingdom.

Reinfeld, F. (1945) *Win at Chess.* Dover Publications.

Reinfeld, F. (1953) *The Human Side of Chess.* Faber and Faber Ltd., London.

Reinfeld, F. (1955) *1001 Winning Chess Sacrifices and Combinations.* Wilshire Book Company.

Reiss, M. (1999) *Personal communication.*

Rollason, J. (1999) *Personal communication.*

Roycroft, A.J. (1986) Queen and Pawn on b7 against Queen. *Roycroft's 5-Man Chess Endgame Series*, no. 7, Chess Endgame Consultants and Publishers, London, England.

Rumelhart, D.E., Hinton, G.E. and Williams, R.J. (1986) Learning internal representation by error propagation. In Rumelhart, D. and McClelland, J. (eds.) *Parallel Distributed Processing*, vol. 1. MIT Press, Cambridge, Mass.

Samuel, A.L. (1959) Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, vol. 3, no. 3, pp 211-229.

Schaeffer, J. (1983) The History Heuristic. *International Computer Chess Association Journal*, vol. 6, no. 3, pp. 16-19.

Schraudolph, N.N., Dayan, P. and Sejnowski, T.J. (1994) Temporal difference learning of position evaluators in the game of go. In Cowan, J.D., Tesauro, G. and Alspector, (eds.) *Advances in Neural Information Processing 6*, Morgan Kaufmann, San Francisco, pp. 817-824.

Schraudolph, N.N. (1998) Online Local Gain Adaptation for Multi-layer Perceptrons. *Technical Report IDSIA-09-98*, IDSIA, Corso Elvezia 36, 6900 Lugano, Switzerland.

Seirawan, Y. (1997) The Kasparov – Deep Blue Games. *International Computer Chess Association Journal*, vol. 20, no. 2, pp. 102-125.

Shannon, C.E. (1950) Programming a Computer to Play Chess. *Philosophical Magazine*, vol. 41, pp. 256-275.

Singh, S.P., and Dayan, P. (1998) Analytical Mean Squared Error Curves for Temporal Difference Learning. *Machine Learning* 25(1): 5-22.

Singh, S.P., and Bertsekas, D. (1997) Reinforcement Learning for Dynamic Channel Allocation in Cellular Telephone Systems. In *Advances in Neural Information Processing Systems: Proceedings of the 1996 Conference*, MIT Press Cambridge MA.

Slate, D. and Atkin, L. (1977) Chess 4.5 - The Northwestern University Chess Program. In Frey, P.W. (ed.) *Chess Skill in Man and Machine*, Springer-Verlag, pp. 82-118.

Smith, M.C. [co-authored with Beal, D.F.] (1994). Random Evaluations in Chess. *International Computer Chess Association Journal*, vol. 17, no. 1, pp. 3-9.

Smith, M.C. [co-authored with Beal, D.F.] (1995) Quantification of Search Extension Benefits. *International Computer Chess Association Journal*, vol. 18, no. 4, pp. 205-218.

Smith, M.C. [co-authored with Beal, D.F.] (1996) Multiple Probes of Transposition Tables. *International Computer Chess Association Journal*, vol. 19, no. 4, pp. 227-233.

Smith, M.C. [co-authored with Beal, D.F.] (1997) Learning piece values using temporal differences. *International Computer Chess Association Journal*, vol. 20, no. 3, pp. 147-151.

Smith, M.C. [co-authored with Beal, D.F.] (1998a) Temporal Coherence and Prediction Decay in Temporal Difference Learning. *Technical report #756*, Dept. of Computer Science, Queen Mary and Westfield College, University of London.

Smith, M.C. [co-authored with Beal, D.F.] (1998b) First Results from using Temporal Difference Learning in Shogi. In van den Herik, H. and Iida, H. (eds.) *Proceedings of the First International Conference on Computers and Games (CG'98)* Springer-Verlag, Berlin, pp. 113-125.

Smith, M.C. [co-authored with Beal, D.F.] (1999a) Temporal Difference Learning Applied to Game Playing and the Results of Application to Shogi. To appear in *Theoretical Computer Science.*

Smith, M.C. [co-authored with Beal, D.F.] (1999b) Temporal Coherence and Prediction Decay in TD Learning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'99)*, Morgan Kaufman, pp. 564-569.

Smith, M.C. [co-authored with Beal, D.F.] (1999c) Temporal Difference Learning for Heuristic Search and Game Playing. To appear in *Information Sciences Journal.*

Stockman, G.C. (1979) A Minimax Algorithm Better than Alpha-Beta? *Artificial Intelligence*, vol. 12, no. 2, pp. 179-196.

Sutton, R. S. (1988) Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, vol. 3, 9-44.

Sutton, R.S., (1992) Adapting bias by gradient descent: an incremental version of delta-bar-delta. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 171-176.

Sutton, R.S., and Singh, S.P. (1994) On Step-Size and Bias in Temporal-Difference Learning. In *Proceedings of the Eighth Yale Workshop on Adaptive and Learning Systems*, pp. 91-96.

Sutton, R.S. (1999) Open Theoretical Questions in Reinforcement Learning. In Fischer, P. and Simon, H.U. (eds.) *Computational Learning Theory (proceedings of EuroCOLT'99)* pp. 11-17.

Tesauro, G. (1989) Neurogammon wins Computer Olympiad. *Neural Computation,* vol. 1, pp. 321-323.

Tesauro, G. (1992) Practical Issues in Temporal Difference Learning. *Machine Learning*, vol. 8, pp. 257-277.

Tesauro, G. (1994) TD-Gammon, a Self-Teaching Backgammon Program, achieves Master Level Play. *Neural Computation*, vol. 6, no. 2. pp. 215-2.

Tesauro, G. (1995) Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, vol. 38, no. 3.

Thompson, K. (1986) Retrograde Analysis of Certain Endgames. *International Computer Chess Association Journal*, vol. 9, no. 3, pp. 131-139.

Turing, A.M. (1950) Computing Machinery and Intelligence. *Mind LIX*, 2236, pp. 433-460.

Turing, A.M. (1953) Digital Computers Applied to Games. In Bowden, B.V. (ed.) *Faster than Thought*, London, pp. 286-310.

Walker, A.N. (1996) Hybrid Heuristic Search. *International Computer Chess Association Journal*, vol. 19, no. 1, pp. 17-23.

Watkins, C.J.C.H. (1989) *Learning from Delayed Rewards.* Ph.D. thesis, Cambridge University.

Widrow, B., and Hoff, M.E. (1960) Adaptive Switching Circuits. *IRE Western Electronic Show and Convention Record*, Part 4, pp. 94-104.

Yamashita, H. (1997) *YSS: About the Data Structures and the Algorithm.* Published on the WWW at `http://plaza15.mbn.or.jp/~yss`

Ye, C. and Marsland, T.A. (1992) Experiments in Forward Pruning with Limited Extensions. *International Computer Chess Association Journal*, vol. 15, no 2, pp.55-66.

Zhang, M. and Yum, T.P. (1989) Comparisons of Channel-Assignment Strategies in Cellular Mobile Telephone Systems. *IEEE Transactions on Vehicular Technology.* Vol. 38, no. 4.

Zhang, W., Dietterich, T.G., (1995). A Reinforcement Learning Approach to Job-Shop Scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, San Francisco, CA. pp. 1114-1120.

Zhang and Dietterich (1996) High-Performance Job-Shop Scheduling with a Time-Delay TD(λ) Network. In Touretzky, Mozer and Hasselmo (eds.) *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference.* MIT Press Cambridge MA.

Zobrist, A.L. (1970) A New Hashing Method with Application for Game Playing. *Technical Report 88, Computer Science Dept. University of Wisconsin.* Reprinted (1990), *International Computer Chess Association Journal*, vol. 13, no. 2, pp. 69-73.

Zweben, M., Daun, B. and Deale, M. (1994) Scheduling and Rescheduling with Iterative Repair. In M. Zweben and M. S. Fox (eds.) *Intelligent Scheduling.* Morgan Kaufman, San Francisco, CA. Pp. 241-255.

APPENDIX A: EXPERIMENTAL DETAILS FROM CHAPTER FOUR

A.1 Test Positions used in the Transposition Table Experiments

Below are the positions used in the transposition table experiments of section 4.2. All positions occurred in games from the 10th VSB Tournament, Amsterdam, Netherlands, March 1996, and are given after Black's 20th move. These positions are available online at `http://www.dcs.qmw.ac.uk/~martins/research/vsb30.set`

Format: [index, players, position, side to move, depth searched]

1, Topalov-Kasparov, 2b1kb2/1pq3rp/r2p1pp1/p1nBpPN1/8/4BQ2/PPP3PP/R4RK1, W, 9
2, Kramnik-Seirawan, 5rk1/r2n1ppp/p2Bpn2/q7/8/p2Q1B2/2P2PPP/3R1RK1, W, 9
3, Lautier-Short, 1r3r2/1b2bpkp/pp2qnp1/2ppN2P/3P1P2/PQN1PB2/1P3P2/3RK1R1, W, 7
4, Gelfand-Anand, 1bqrr1k1/1p3p1p/p1p5/3n1pp1/1PNP2b1/P2NP1P1/2Q2P1P/R3RBK1, W, 8
5, Timman-Piket, r1b2rk1/1p3pp1/4p2p/1B2n1b1/p2RP3/P1N4P/1PPNR1P1/2K5, W, 10
6, Short-Gelfand, r1r1k3/3nbpp1/p1qp1n2/1p2pPB1/4P3/P1N4Q/1PPN3P/R5RK, W, 8
7, Anand-Topalov, 1rb2rk1/2q2ppp/3p4/3P4/pp3BPb/5B2/PPPQ3P/2KR2R1, W, 8
8, Seirawan-Lautier, r1b1r1k1/pp3ppp/2pQ1n2/P5q1/2PPp2n/2N1P2P/3N1PP1/R2R1BK1, W, 8
9, Piket-Kasparov, r1r3k1/1n1bB1b1/p4pp1/1p1Pp2n/4P3/5P1P/PP1KNN2/R4B1R, W, 8
10, Timman-Kramnik, 3q1rk1/5pp1/2n1pb1p/p1Pp4/3P4/1PN2N2/3Q1PPP/1R4K1, W, 9
11, Topalov-Short, r2qr2k/1p1b2pp/p4p2/2BR1P1Q/8/1B6/PPP3PP/6K1, W, 9
12, Kasparov-Anand, r2qr1k1/p2b1pp1/2n1pb1p/2ppN3/3P1B2/2PB4/P4PPP/1R1QR1K1, W, 9
13, Lautier-Timman, r4rk1/p3qpp1/1p1n3p/1P1P4/3p4/P2B2B1/1PQ2PPP/1K1R4, W, 9
14, Gelfand-Seirawan, 4r1k1/pb1q1rpp/1p3n2/3p1p2/P1pP4/2P1PP2/3BB1PP/RQ3RK1, W, 8
15, Short-Kasparov, 3r1rk1/2q1bpn1/p3p1p1/1p2P2p/1P1BQ1P1/2P5/P1B4P/R4RK1, W, 9
16, Kramnik-Lautier, r4rk1/1p1b1pp1/pq1b3p/8/1P2BP2/P2Q4/3R2PP/2B2RK1, W, 10
17, Seirawan-Topalov, 2r1r1k1/3nqpp1/2p2n1p/p2p4/1PpP4/P3PP2/1Q2NBPP/R4RK1, W, 9
18, Timman-Gelfand, 4rrk1/pp3pp1/2p4p/3pNn2/3P4/1PP5/P4PPP/4RRK1, W, 10
19, Topalov-Timman, r2q2k1/1p1n1rp1/4p2B/p2pPp2/3P3b/1P1BQP1P/P4P2/1R4RK, W, 9
20, Kasparov-Seirawan, r1b1k1r1/1p1n1q2/p3p2p/4PpB1/P1Q5/2N2N1P/1P4P1/R6K, W, 9
21, Anand-Short, r2q1n1r/4bpkn/3p2p1/pQ1Pp1P1/Pp4B1/1N2B3/1PP2P2/2KR3R, W, 9
22, Lautier-Piket, 2r2Bk1/pb3ppp/4pn2/4q3/2B1n3/4P3/4QPPP/R4RK1, W, 9
23, Kramnik-Topalov, r4rk1/pb1nb2p/1q1p2p1/2pPPp2/1p3P1N/1P4P1/1B5P/R2QRBK1, W, 9
24, Lautier-Gelfand, 2kr1b1r/1bq3pp/8/1pn1p3/2p5/4BN2/1PB1QPPP/R4RK1, W, 9
25, Seirawan-Anand, r3r1k1/ppq2pb1/3np1p1/3pN2p/P2P1P1P/2PQ2P1/1P1N2K1/R3R3, W, 8
26, Piket-Short, r1q3k1/p3rpp1/1p5p/1BnR1Q2/8/4PN2/Pb3PPP/5RK1, W, 9
27, Timman-Kasparov, r1b2q1k/3nb2n/p2p1rp1/1p1Pp3/4P2p/1NN1BP1P/PP1QB3/1K1R2R1, W, 8
28, Kasparov-Kramnik, r1b2k1r/4bp2/4pp2/3q1P1p/pp6/3B3Q/PPP1N1PP/1KR4R, W, 9
29, Short-Seirawan, r4rk1/n1q1bpp1/2p1p1bp/P2pP3/2nP4/1NB2N2/Q3BPPP/R3R1K1, W, 9
30, Anand-Timman, r4r1k/1bpqb3/p1np4/3Np2p/Q3Ppp1/2PP1N1P/1P2KPP1/R1B1R3, W, 9

A.2 Details from the Search Extension Experiments

Here we indicate the 563 problems from the book *1001 Winning Chess Sacrifices and Combinations* (Reinfeld, 1955) that were used in the search extension experiments of section 4.4.

Each entry represents: *position number*, *depth to solution*, *target gain*.

The *target gain* is measured relative to the material balance in the position initially. *M* means that the solution leads to checkmate.

The actual positions themselves can be found in machine readable format at

`http://www.dcs.qmw.ac.uk/~martins/research/wcsac563.txt`

1,4,5	2,8,7	3,4,2	5,6,3	6,6,3	7,10,1	8,4,4
9,8,3	11,6,1	12,6,3	14,6,1	15,4,3	16,4,2	18,6,2
19,6,4	22,8,2	23,4,2	25,4,1	26,8,3	28,6,2	29,4,1
30,10,2	31,8,3	33,6,3	34,4,5	35,6,4	36,6,3	41,8,1
44,4,1	45,6,4	48,8,3	49,8,3	50,4,1	52,6,1	57,8,1
58,6,1	59,4,3	61,6,6	63,6,4	65,8,1	66,4,M	68,6,3
70,8,3	72,6,5	73,4,3	74,6,3	75,12,6	76,8,4	77,8,11
79,8,2	82,6,1	87,6,2	88,4,6	89,6,2	90,4,4	91,8,M
92,8,3	93,4,1	94,8,3	100,6,5	102,8,10	103,4,6	105,6,2
106,6,5	109,4,3	111,6,3	113,6,8	116,4,2	117,4,2	118,4,2
119,4,4	120,6,3	122,6,1	123,8,10	124,6,4	125,4,4	126,4,6
127,4,2	128,6,4	129,6,5	131,8,1	132,8,6	134,4,2	135,6,6
136,10,M	137,10,3	138,8,3	141,8,1	142,8,3	143,4,2	144,8,3
146,4,5	149,4,2	150,8,2	152,4,2	153,8,4	154,4,2	155,6,2
159,4,2	160,6,1	163,4,3	165,6,2	166,4,1	167,4,1	168,6,2
171,4,2	172,4,2	173,4,1	174,4,3	177,4,3	178,4,2	180,4,3
181,4,3	185,10,2	186,4,3	187,4,2	188,4,2	189,4,3	193,6,3
194,6,1	195,4,1	198,4,2	199,12,M	200,6,2	202,4,2	203,4,5
204,4,3	205,8,2	207,4,3	208,4,3	209,6,1	210,8,2	211,4,3
213,4,3	214,4,3	215,8,2	217,10,2	218,6,3	221,4,2	222,4,2
223,4,2	225,4,2	226,8,1	227,4,2	228,6,4	231,4,4	233,6,2
235,6,3	236,6,4	237,6,2	239,6,1	240,8,2	241,10,3	245,4,1
246,4,2	247,6,3	249,6,2	250,6,2	251,12,M	252,8,1	254,4,3
255,6,2	256,6,2	257,6,3	262,4,1	263,6,2	264,4,2	265,6,1
268,6,3	269,4,2	272,10,2	273,6,1	275,4,4	276,6,1	277,4,2
278,4,1	279,4,2	280,4,3	281,6,6	282,6,1	284,4,2	285,4,4
287,6,3	288,4,2	290,6,3	291,4,3	292,4,1	293,6,6	295,6,8
297,8,1	298,4,2	300,4,1	301,8,15	303,4,2	306,4,5	307,6,2
308,10,4	310,4,3	311,4,3	312,10,5	313,10,2	314,6,9	316,4,6
318,6,2	320,8,2	321,8,M	323,6,1	324,4,3	326,8,8	329,4,2
332,6,2	334,4,3	337,4,3	338,4,1	339,8,1	340,10,2	341,4,3
342,6,4	343,6,2	344,4,2	346,6,3	347,4,7	348,4,5	350,6,M
351,4,3	352,4,2	353,6,2	354,4,8	355,6,3	356,4,3	357,6,7
360,8,6	363,6,3	364,10,6	365,4,2	370,8,3	372,6,3	374,6,M
375,4,2	377,10,9	378,8,M	379,6,4	380,6,M	382,6,3	383,4,5
385,4,4	386,8,3	388,4,M	390,8,3	392,4,5	394,4,1	399,6,M
400,8,2	402,8,M	405,6,M	406,6,M	409,6,M	410,4,M	411,8,1
412,6,1	414,10,M	415,4,M	418,4,M	419,8,5	420,4,7	421,6,M
422,8,4	423,6,M	424,8,2	426,6,M	427,12,M	428,6,1	430,6,5
432,6,M	433,4,6	436,8,4	438,6,M	440,4,4	443,8,5	444,8,3

445,6,M	446,6,5	447,8,M	448,6,M	449,4,4	452,8,3	454,8,5
455,4,M	456,6,2	457,8,2	458,4,2	460,8,3	461,8,5	462,4,5
463,4,M	467,4,3	468,4,4	469,6,1	476,8,7	478,6,4	479,8,8
480,4,1	482,4,M	485,8,3	486,4,5	487,8,5	492,4,5	497,4,1
498,4,3	499,4,1	501,6,2	503,4,2	504,4,2	505,8,4	507,4,4
509,8,2	510,4,1	511,6,3	513,4,3	514,4,2	520,4,2	521,8,M
523,4,4	524,8,4	532,4,1	535,8,7	536,6,3	537,4,3	538,4,M
539,8,9	543,4,3	544,6,3	545,4,M	548,4,5	549,8,4	551,4,5
556,4,4	557,10,M	558,4,2	560,6,2	562,8,4	564,10,M	565,8,7
567,8,M	572,4,5	577,8,2	581,8,2	583,8,5	585,8,6	587,4,4
588,8,2	590,8,2	594,10,M	597,8,M	598,6,M	600,4,4	603,4,4
605,6,1	607,10,3	608,8,1	610,4,2	613,4,2	614,4,2	615,6,3
619,4,1	622,6,2	625,4,1	626,4,5	629,8,4	630,4,3	631,4,4
632,8,1	634,6,2	635,8,2	637,4,5	638,4,4	641,4,4	642,6,3
643,8,3	645,8,M	646,6,3	647,10,M	648,10,2	649,6,2	650,10,3
651,8,4	652,4,5	654,4,2	660,8,4	662,4,9	664,8,M	666,4,3
667,8,3	668,10,3	669,8,3	670,8,5	671,4,M	675,4,4	676,6,9
677,6,2	678,6,8	680,8,3	681,8,M	682,8,5	684,6,5	685,6,8
686,6,5	687,8,3	689,6,7	690,8,M	691,8,3	692,6,10	693,4,4
694,10,M	695,6,5	697,8,3	699,4,4	700,4,4	701,4,5	702,4,5
703,8,M	704,10,M	705,8,10	706,4,5	712,6,10	715,4,M	717,6,M
718,6,M	719,4,2	720,4,M	721,8,3	722,6,M	723,4,4	725,6,M
726,4,M	727,6,M	729,10,M	731,4,M	735,6,M	736,4,M	737,6,M
739,10,M	740,8,M	741,10,M	744,6,M	745,6,2	746,8,M	749,8,M
751,6,M	752,6,M	753,8,M	755,4,5	758,8,2	759,6,2	761,6,2
762,6,5	763,6,2	764,4,9	766,10,10	767,6,3	769,12,3	770,6,4
772,10,M	773,4,1	774,6,3	776,6,5	777,6,2	778,4,3	781,6,M
783,12,2	784,6,3	785,4,6	786,8,3	787,12,1	788,4,4	789,4,6
795,8,7	797,8,4	798,10,M	800,8,1	802,4,9	804,10,13	805,8,8
806,6,5	807,8,2	809,8,5	810,6,M	812,8,5	813,4,M	814,8,1
816,8,1	821,6,6	822,4,3	823,6,2	825,8,2	826,6,1	831,10,5
837,8,2	838,4,M	841,10,9	848,4,4	851,6,2	854,4,1	855,6,3
856,4,1	858,6,2	871,10,4	872,6,1	874,8,1	878,8,5	879,4,M
881,4,5	882,6,2	885,6,3	886,8,2	888,8,1	892,6,1	893,6,M
894,8,1	897,4,M	898,6,4	899,6,M	900,4,6	902,8,5	904,4,M
905,8,8	906,6,M	908,8,1	912,4,M	913,8,2	918,8,1	920,4,M
921,6,7	922,8,4	923,6,4	925,8,4	927,6,3	930,6,1	931,4,M
934,10,1	935,8,M	937,4,M	939,6,M	944,4,6	947,4,4	948,6,M
955,6,5	957,6,9	959,10,1	960,4,5	962,8,M	964,4,M	969,8,9
971,6,6	975,6,3	981,6,M	983,8,M	989,6,M	993,6,10	994,10,M
997,8,6	999,4,M	1001,6,M				

	Overall (563)			Depths 4-5 (206)			Depths 6-7 (176)			Depths 8-9 (136)			Depths 10+ (45)		
Extensions	*Total*	*SD*	*RD*	*Total*	*SD*	*RD*	*Totals*	*SD*	*RD*	*Totals*	*SD*	*RD*	*Totals*	*SD*	*RD*
None	738,960	0	0	4,629	0	0	57,563	0	0	938,020	0	0	6,163,974	0	0
Checks (1)	88,887	0	0	2,371	0	0	16,653	0	0	256,297	0	0	261,505	0	0
Checks (2)*	84,140	0	0	2,383	0	0	16,437	0	0	233,815	0	0	270,848	0	0
Recapts	661,226	0	59,312	5,540	0	418	64,717	0	5,868	1,011,942	0	110,866	4,935,880	0	382,129
SingEx (1)	360,176	248,930	0	5,907	1,844	0	55,069	27,363	0	788,802	525,406	0	1,879,844	1,411,035	0
SingEx (2)*	508,794	205,913	0	6,045	903	0	85,675	26,038	0	1,286,284.	491,437	0	2,115,379	984,997	0
Null (1)*	345,691	0	0	3,595	0	0	33,056	0	0	380,517	0	0	3,029,223	0	0
Null (2)	129,510	0	0	3,508	0	0	23,263	0	0	197,845	0	0	915,340	0	0
Checks+Recapts	62,259	0	6,669	1,795	0	165	19,789	0	2,250	180,101	0	20,562	149,016	0	11,736
Checks+SingEx	82,947	49,689	0	2,101	477	0	22,513	10,935	0	177,606	103,737	0	403,320	263,192	0
Checks+Null	46,910	0	0	1,997	0	0	9,294	0	0	140,783	0	0	115,929	0	0
Recapts+SingEx	372,378	248,282	34,855	6,029	1,815	496	68,813	34,814	5,802	976,091	649,029	101,218	1,412,164	1,000,304	105,208
Recaps+Null	151,396	0	18,209	3,755	0	260	26,045	0	2,627	259,312	0	31,899	991,379	0	119,945
SingEx+Null	156,214	107,900	0	6,946	2,709	0	37,974	21,994	0	343,603	241,732	0	735,651	520,956	0
Checks+Recapts+SingEx	100,612	59,493	9,167	2,516	558	200	26,161	12,240	2,211	264,912	158,925	27,441	344,303	213,587	22,197
Checks+Recapts+Null	30,614	0	2,885	2,159	0	124	10,556	0	808	74,940	0	8,054	105,361	0	8,023
Checks+SingEx+Null	64,106	41,387	0	2,923	768	0	17,048	8,814	0	120,453	78,039	0	357,950	243,961	0
Recapts+SingEx+Null	163,433	111,493	15,591	6,299	2,351	444	48,230	27,607	4,139	421,994	301,593	41,525	551,898	364,679	51,344
Chck+Rcpt+SingEx+Null	69,738	44,837	5,693	3,433	1,015	218	19,983	10,432	1,326	145,299	95,805	12,805	339,507	225,970	26,344

Table A.1: Full results from the search extension experiments.

Table A.1 presents the full results[1] from the search extension experiments of section 4.4. Extension heuristic variants marked * were not carried forwards into combinations. The values presented are measured in nodes to solution, where *Total* is the total search effort, and *SD* and *RD* are the cost of singular detection and recapture detection respectively.

[1] Except the extension rule 'all-captures'. This rule was clearly deleterious and was not included.

APPENDIX B: EXPERIMENTAL DETAILS FROM CHAPTER FIVE

	Pawn	*Knight*	*Bishop*	*Rook*	*Queen*	*Pawn*	*Knight*	*Bishop*	*Rook*	*Queen*
Depth 1										
Run A	0.52	0.92	1.10	1.76	3.77	1.00	1.77	2.13	3.39	7.28
Run B	0.48	0.90	1.13	1.77	4.05	1.00	1.85	2.34	3.66	8.36
Run C	0.56	1.12	1.18	1.86	3.90	1.00	1.99	2.11	3.31	6.95
Run D	0.58	1.11	1.19	1.83	3.92	1.00	1.92	2.05	3.15	6.74
Run E	0.56	1.01	1.08	1.78	3.90	1.00	1.80	1.94	3.18	6.98
Ave.	0.54	1.01	1.14	1.80	3.91	1.00	1.86	2.11	3.34	7.26
Stdev.	0.04	0.10	0.05	0.04	0.10	0.00	0.09	0.15	0.20	0.64
Depth 2										
Run A	0.39	1.11	1.50	2.08	4.30	1.00	2.87	3.86	5.37	11.09
Run B	0.40	1.00	1.26	1.84	4.18	1.00	2.52	3.17	4.65	10.56
Run C	0.37	1.13	1.39	1.68	4.41	1.00	3.07	3.76	4.54	11.95
Run D	0.40	1.03	1.25	1.82	4.20	1.00	2.61	3.16	4.61	10.64
Run E	0.40	1.07	1.44	1.99	4.34	1.00	2.70	3.63	5.03	10.95
Ave.	0.39	1.07	1.37	1.88	4.29	1.00	2.75	3.52	4.84	11.04
Stdev	0.01	0.06	0.11	0.16	0.09	0.00	0.22	0.33	0.35	0.55
Depth 3										
Run A	0.76	1.36	1.76	3.15	6.18	1.00	1.78	2.31	4.13	8.11
Run B	0.80	1.47	1.79	3.21	6.25	1.00	1.83	2.24	4.01	7.81
Run C	0.79	1.29	1.88	2.92	6.47	1.00	1.64	2.38	3.69	8.19
Run D	0.77	1.41	1.72	2.98	6.40	1.00	1.83	2.23	3.87	8.31
Run E	0.79	1.31	1.90	2.93	6.36	1.00	1.65	2.39	3.69	8.00
Ave.	0.78	1.37	1.81	3.04	6.33	1.00	1.75	2.31	3.88	8.08
Stdev.	0.02	0.07	0.08	0.14	0.12	0.00	0.10	0.08	0.20	0.19
Depth 4										
Run A	0.60	1.66	2.02	2.75	6.61	1.00	2.76	3.36	4.57	11.00
Run B	0.58	1.49	1.93	2.92	6.43	1.00	2.56	3.31	5.00	11.01
Run C	0.53	1.60	1.93	2.81	6.79	1.00	3.01	3.62	5.29	12.76
Run D	0.58	1.56	2.02	2.81	6.64	1.00	2.71	3.50	4.87	11.51
Run E	0.57	1.47	1.78	2.92	6.36	1.00	2.58	3.13	5.13	11.17
Ave.	0.57	1.56	1.93	2.84	6.57	1.00	2.72	3.38	4.97	11.49
Stdev.	0.03	0.08	0.10	0.07	0.17	0.00	0.18	0.19	0.27	0.74
Depth 5										
Run A	0.96	1.97	2.31	3.78	7.45	1.00	2.04	2.40	3.92	7.73
Run B	0.86	1.99	2.28	3.81	7.14	1.00	2.33	2.67	4.45	8.35
Run C	0.88	1.81	2.11	3.36	7.26	1.00	2.06	2.40	3.81	8.25
Run D	0.92	1.95	2.33	3.49	7.22	1.00	2.12	2.53	3.79	7.85
Run E	0.89	1.87	2.34	3.90	7.61	1.00	2.10	2.63	4.38	8.55
Ave.	0.90	1.92	2.23	3.65	7.28	1.00	2.14	2.49	4.06	8.11
Stdev.	0.04	0.08	0.10	0.23	0.19	0.00	0.12	0.13	0.32	0.35
Depth 6										
Run A	0.72	2.16	2.59	3.61	7.63	1.00	2.99	3.58	4.99	10.53
Run B	0.74	2.11	2.49	3.69	8.18	1.00	2.86	3.37	5.00	11.09
Run C	0.72	2.02	2.47	3.67	7.52	1.00	2.80	3.42	5.08	10.43
Run D	0.69	2.05	2.44	3.55	7.49	1.00	2.97	3.54	5.14	10.86
Run E	0.71	2.09	2.50	3.69	7.45	1.00	2.94	3.52	5.19	10.49
Ave.	0.72	2.09	2.50	3.64	7.65	1.00	2.91	3.49	5.08	10.68
Stdev.	0.02	0.05	0.06	0.06	0.30	0.00	0.08	0.09	0.09	0.28

Table B.1: Final values for individual chess runs, at various depths.

Table B.1 presents the final piece values (averaged over the last 20% of the runs) for the individual chess runs described in Chapter 5. The values are presented first in their absolute forms and then again normalised to pawn = 1.

APPENDIX C: EXPERIMENTAL DETAILS FROM CHAPTER SIX

		Pawn	Lance	Knight	Silver	Gold	Bishop	Rook
Depth1	A	0.08	0.16	0.19	0.33	0.39	0.45	0.59
	B	0.08	0.17	0.21	0.37	0.39	0.50	0.61
	C	0.08	0.16	0.23	0.34	0.41	0.49	0.60
	D	0.08	0.15	0.23	0.31	0.37	0.47	0.62
	E	0.09	0.19	0.20	0.33	0.33	0.44	0.56
Depth2	A	0.13	0.28	0.32	0.49	0.56	0.71	0.93
	B	0.13	0.32	0.26	0.44	0.54	0.70	0.93
	C	0.12	0.29	0.31	0.49	0.57	0.72	0.95
	D	0.11	0.30	0.30	0.52	0.59	0.73	0.99
	E	0.13	0.27	0.29	0.48	0.58	0.74	0.91
Depth3	A	0.12	0.24	0.27	0.60	0.70	0.83	1.15
	B	0.13	0.32	0.34	0.71	0.80	0.99	1.35
	C	0.12	0.26	0.38	0.63	0.79	0.87	1.21
	D	0.13	0.32	0.35	0.64	0.77	0.92	1.25
	E	0.14	0.26	0.32	0.64	0.81	0.98	1.30
Depth4	A	0.11	0.35	0.37	0.58	0.69	0.90	1.07
	B	0.16	0.36	0.36	0.65	0.68	0.84	1.10
	C	0.14	0.32	0.38	0.57	0.71	0.86	1.14
	D	0.10	0.21	0.25	0.52	0.62	0.77	1.05
	E	0.12	0.23	0.33	0.55	0.64	0.74	1.06

Table C.1: Main piece values for each of the five runs (a-e) at depths 1-4

		Pawn	Lance	Knight	Silver	Gold	Bishop	Rook
Depth 1	A	0.19	0.09	0.20	0.23	-	0.95	1.07
	B	0.22	0.12	0.24	0.13	-	0.93	1.04
	C	0.27	0.13	0.04	0.43	-	0.97	1.15
	D	0.25	0.11	0.04	0.28	-	0.95	1.11
	E	0.17	0.13	0.15	0.11	-	0.89	1.05
Depth2	A	0.76	0.58	0.59	0.67	-	1.39	1.60
	B	0.72	0.49	0.60	0.59	-	1.23	1.71
	C	0.74	0.54	0.58	0.67	-	1.35	1.65
	D	0.72	0.48	0.53	0.69	-	1.30	1.72
	E	0.73	0.50	0.48	0.65	-	1.33	1.63
Depth3	A	0.72	0.35	0.54	0.59	-	1.46	1.88
	B	0.72	0.25	0.57	0.02	-	1.84	2.11
	C	0.64	0.18	0.51	0.22	-	1.53	1.93
	D	0.63	0.43	0.30	0.27	-	1.70	2.03
	E	0.82	0.38	0.56	0.06	-	1.85	2.13
Depth4	A	0.74	0.65	0.73	0.85	-	1.43	1.74
	B	0.85	0.78	0.78	0.85	-	1.32	1.61
	C	0.75	0.80	0.79	0.80	-	1.37	1.69
	D	0.56	0.36	0.58	0.67	-	1.31	1.66
	E	0.65	0.39	0.57	0.69	-	1.31	1.57

Table C.2: Promoted piece values for each of the five runs (a-e) at depths 1-4

		Pawn	Lance	Knight	Silver	Gold	Bishop	Rook
Main	Begin.	0.50	1.00	1.00	2.00	2.00	4.00	5.00
	Gnu	0.42	2.11	2.11	2.83	3.03	4.74	5.00
	YSS	0.48	2.07	2.16	3.08	3.32	4.28	5.00
Promoted	Begin.	1.50	1.75	1.75	2.00	-	5.00	6.00
	Gnu	1.91	2.50	2.63	3.16	-	5.21	5.16
	YSS	2.02	3.03	3.08	3.22	-	5.53	6.25

Table C.3: Piece values used in matches (normalised to rook=5)

APPENDIX D: EXPERIMENTAL DETAILS FROM CHAPTER SEVEN

Pawn ranks	*Rank 2*	*Rank 3*	*Rank 4*	*Rank 5*	*Rank 6*	*Rank 7*
Pawn	0.000	0.033	0.100	0.233	0.500	1.000
Piece centrality	*Ring 1*	*Ring 2*	*Ring 3*	*Ring 4*		
Knight	0.000	0.067	0.133	0.200		
Bishop	0.000	0.033	0.067	0.100		
Rook	0.000	0.000	0.000	0.000		
Queen	0.000	0.033	0.067	0.100		

Table D.1: The pawn rank and piece centrality bonuses used by the weight set *Central*

Table D.1 presents the pawn rank and piece centrality bonuses used by the weight set *Central* in section 7.2.3.

	Halfboard		*Fullboard*	
Pawn	1.00	(0.00)	1.00	(0.00)
Knight	2.67	(0.17)	2.76	(0.11)
Bishop	3.07	(0.19)	3.08	(0.11)
Rook	4.72	(0.25)	4.62	(0.17)
Queen	9.39	(0.57)	9.46	(0.50)

Table D.2: Chess piece values learnt using piece-square tables

The learnt chess piece values used in Figure 7.12 are given in Table D.2. These values were learnt in conjunction with the piece-square values presented in Table D.3 below. The values are normalised to pawn=1, and their standard deviations are given in parentheses.

	Pawn					**Knight**			
8	0.00	0.00	0.00	0.00	8	0.00	0.71	0.73	0.73
7	2.14	1.92	1.68	1.63	7	0.53	0.77	1.08	0.81
6	1.25	1.20	1.16	1.13	6	0.79	1.14	1.16	1.44
5	0.46	0.61	0.54	0.65	5	0.99	1.24	1.29	1.45
4	0.20	0.38	0.35	0.53	4	0.77	1.05	1.23	1.29
3	0.10	0.34	0.26	0.37	3	0.57	0.86	0.95	1.11
2	0.00	0.30	0.27	0.10	2	0.48	0.79	0.91	0.76
1	0.00	0.00	0.00	0.00	1	0.64	0.41	0.60	0.67
	a,h	*b,g*	*c,f*	*d,e*		*a,h*	*b,g*	*c,f*	*d,e*
	Bishop					**Rook**			
8	0.25	0.19	0.21	0.15	8	1.01	0.99	1.05	0.86
7	0.17	0.39	0.31	0.25	7	1.11	1.13	1.10	1.03
6	0.30	0.42	0.55	0.43	6	0.92	0.91	0.92	0.88
5	0.30	0.43	0.52	0.50	5	0.66	0.60	0.88	0.81
4	0.21	0.26	0.47	0.53	4	0.40	0.61	0.66	0.76
3	0.13	0.36	0.40	0.31	3	0.13	0.46	0.50	0.60
2	0.23	0.28	0.26	0.26	2	0.00	0.36	0.56	0.60
1	0.08	0.06	0.00	0.08	1	0.09	0.13	0.36	0.55
	a,h	*b,g*	*c,f*	*d,e*		*a,h*	*b,g*	*c,f*	*d,e*
	Queen								
8	0.27	0.55	0.78	0.57					
7	0.41	0.70	1.11	0.73					
6	0.71	1.12	1.15	1.10					
5	0.45	0.31	0.89	0.74					
4	0.20	0.48	0.50	0.55					
3	0.21	0.28	0.30	0.40					
2	0.14	0.29	0.32	0.30					
1	0.29	0.00	0.04	0.19					
	a,h	*b,g*	*c,f*	*d,e*					

Table D.3: Half-board piece-square values.

Table D.3 shows the piece-square values learnt by the half-board runs, and used to create Figures 7.6 to 7.10. Table D.4 shows the full-board values learnt for queens and used in Figure 7.11. All values are presented from White's point of view.

	Queen (full-board)							
8	0.14	0.55	0.55	0.38	0.43	0.55	0.51	0.31
7	0.31	0.58	0.64	0.49	0.48	1.24	0.85	0.47
6	0.33	0.50	0.94	0.88	0.92	0.99	1.13	0.56
5	0.28	0.26	0.56	0.63	0.80	0.89	0.39	0.53
4	0.15	0.38	0.49	0.36	0.53	0.49	0.45	0.20
3	0.18	0.30	0.34	0.32	0.31	0.22	0.38	0.18
2	0.14	0.24	0.29	0.14	0.20	0.29	0.20	0.27
1	0.18	0.00	0.04	0.19	0.09	0.11	0.17	0.23
	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*

Table D.4: Full-board piece-square values for queens.

APPENDIX E: RANDOM EVALUATIONS IN CHESS

This Appendix is a lightly edited version of a co-authored paper which appeared in the International Computer Chess Association Journal (vol. 17, no 1, 1994). It is included here because although it does not form part of the main results presented in this thesis, it is relevant to the discussion of minimax in Chapter 3, and influenced and informed the experiments presented in Chapter 7. Subsequent to the publication of the paper, Levene and Fenner (1995) presented a combinatorial analysis of minimax using random evaluations, based on our experiments, that supported our results.

E.1 Introduction

This Appendix reports on experiments using random numbers as 'evaluations' in chess. Although this results in random play with a depth-1 search, it is found that strength of play rises rapidly with increasing depth of lookahead. This counter-intuitive result and its implications for game-playing are discussed.

On first encounter it is surprising to most people that random 'evaluations' produce anything better than random play when used in a minimax lookahead search. The natural assumption is that lookahead on pure random numbers will result in a random choice at the root.

This is, of course, a theoretical investigation, and 'strength' of play is used in a purely relative sense. Lookahead on random numbers produces extremely weak play compared with lookahead using even very simple chess-specific evaluations. There is no expectation that random numbers can do as well as chess-specific evaluations. Nevertheless, it is an intellectually interesting observation that lookahead on random evaluations produces better-than-random play, and the matter is perhaps useful in illuminating some of the fundamental principles of minimax search. It is possible to imagine circumstances in which the effect might have implications for practical work in game playing and these are discussed at the end.

E.2 The First Experiment

Imagine two competing chess programs: one makes a list of all legal moves and then chooses a move at random. Call this strategy *root-random*. The second program performs a full-width minimax search down to a fixed depth d to select its move. The

'evaluation' function used to score the nodes at depth *d* is entirely random. Hence the move chosen at the root is determined from the backed-up random values from the nodes at depth *d*. Call this strategy *lookahead-random*. Neither of these programs utilises any chess-specific knowledge other than the rules of the game, which define the game tree.

One might intuitively expect these two programs to have equivalent performance, e.g. if played against each other over a large number of games they would each score 50%. Contrary to intuition, lookahead-random decisively out-performs root-random, even at very low search depths.

However, there is a relatively obvious mechanism, of no interest to the present research, by which lookahead-random, as described above, might outperform root-random. Notably, the lookahead program will have a significant advantage when game-terminal nodes lie within the lookahead. The lookahead program sees imminent game terminations and root-random does not. If the game wanders into a position with a mate-in-2 available, the root-random program has no reason to select that move, whereas a 3-ply lookahead program (assuming game-terminal positions are recognised and scored using the rules of the game) *will* select it. Similarly a 5-ply lookahead will see and select mates up to mate-in-3.

In order to eliminate this advantage of lookahead-random over root-random, the experiments described here perform a slightly different comparison. Instead of 'root-random' as described above, a program called *lookahead-zero* is used. Lookahead-zero performs a lookahead search to the same depth *d* as lookahead-random, but lookahead-zero uses a constant, zero, as its evaluation at all non-terminal nodes. Lookahead-zero usually finds all top-level moves have the same score - zero. It makes its choice by tie-breaking with a single random number applied at the root only. Thus it usually behaves like root-random, except that when game-terminal nodes are encountered, the values are backed up and used to give lookahead-zero the same capability to play and avoid imminent mates as lookahead-random has.

E.2.1 Results of the first experiment

Figure E.1 shows the results of the first experiment, pitting lookahead-random (LR) against lookahead-zero (LZ) at depths from 1 to 5. The results are decisive. Lookahead on random numbers improves play substantially. At 1 ply, the two programs are equivalent, and the results show 50-50 scores. By 5 ply, all games are won by the side using random numbers, none by lookahead-zero, not even a single draw!

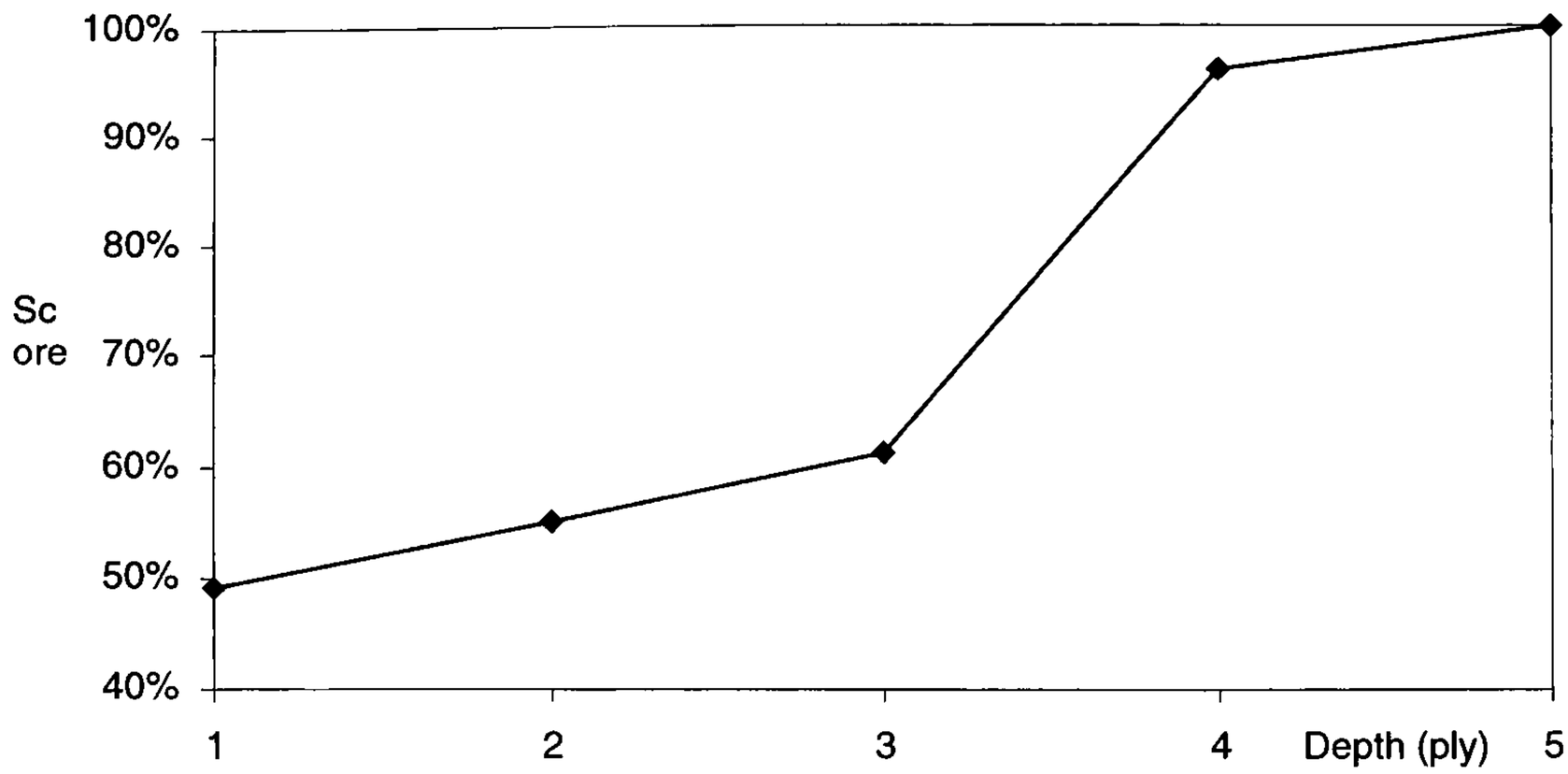

Figure E.1: Percentage scores for lookahead-random (LR) playing against lookahead-zero (LZ).

E.3 Two Additional Experiments

The concept of random evaluations can also be applied to evaluation components rather than the whole evaluation. If we have a existing chess-specific evaluation - material, for example, then we can combine it with a random evaluation, weighted to be less significant than any change in the chess-specific term. Thus, for instance, we could construct the combined evaluation <MB,Rand> = 1000*MB + rand(1000), where MB stands for *material balance* and rand(N) means a random number in the range 0..(N-1). We could then look to see if <MB,Rand> performs any better than <MB,Zero> . (The constant 1000 is for illustration: the principle is independent of the weight chosen.)

This experiment is similar in principle to the comparison between lookahead-random and lookahead-zero. In fact it is exactly the same mechanism applied within a different tree. This may be seen by considering the subtree of the game tree in which all branches have the same backed-up MB value. Within this (perhaps narrow) subtree, all paths are equivalent as far as MB is concerned. Using MB alone, with a single application of random at the root to tie-break, is the equivalent of lookahead-zero *within the subtree*. Using <MB,Rand> is the equivalent of lookahead-rand, *within its subtree*. Thus we would expect the same effect to occur, although the reduction in effective branching factor, and other differences due to exploring different regions of the tree, might affect the strength of the effect.

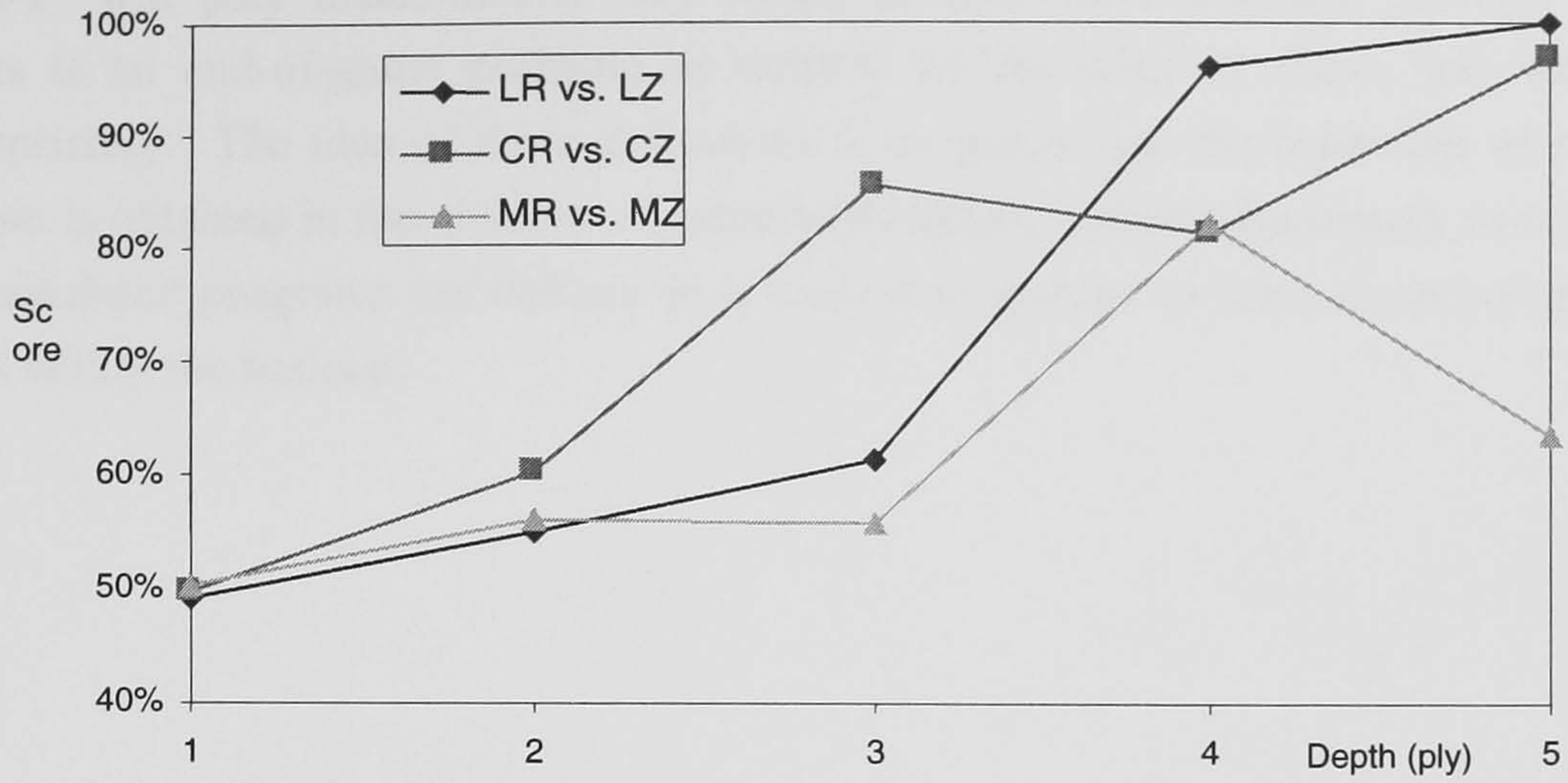

Figure E.2: Percentage scores for LR v LZ, CR v CZ, and MR v MZ.

We performed two such additional experiments, using <MB, Rand> and <Ctree, Rand>. <Ctree, Rand> is also based on material balance, but quiesces material by exploring the capture tree, in order to obtain the chess-specific part of the evaluation. The experiments showed that the effect was nearly as strong as for random versus zero without a chess-specific component. Figure E2 shows the results. CR stands for Ctree Random, CZ for Ctree Zero, MR for Material-balance Random and MZ for Material-balance Zero.

E.4 Further Details of the Experiments

Before discussing the results, we give some more detail of the experimental implementation, in order to answer some questions that might be raised.

E.4.1 Handling of the game-terminal positions

The handling of game-terminal positions is critical to ensuring fair comparison between the different lookahead algorithms.

Figure E.3 is intended to clarify the operation of, and differences between, the programs we have called *root-random*, *lookahead-random* and *lookahead-zero*.

Root-random and lookahead-random differ in their behaviour at search-depth 1. *Root-random*, defined as merely choosing randomly from the move list, would not notice mate-in-1, and might not choose it. However, all versions of the lookahead programs are defined to detect end-of-game-tree nodes, stop the search at that point, and use the value defined by the rules of the game. Thus *lookahead-random-to-*

depth-1 will play mates-in-1 if they occur, because the checkmate position that results is an end-of-game position, as defined by the rules of chess, and valued appropriately. The idea of these definitions is to ensure that the behaviour of root-random is obtained in the absence of game termination, and simultaneously to ensure the lookahead programs are defined in a consistent manner to react to end-of-game nodes within the horizon.

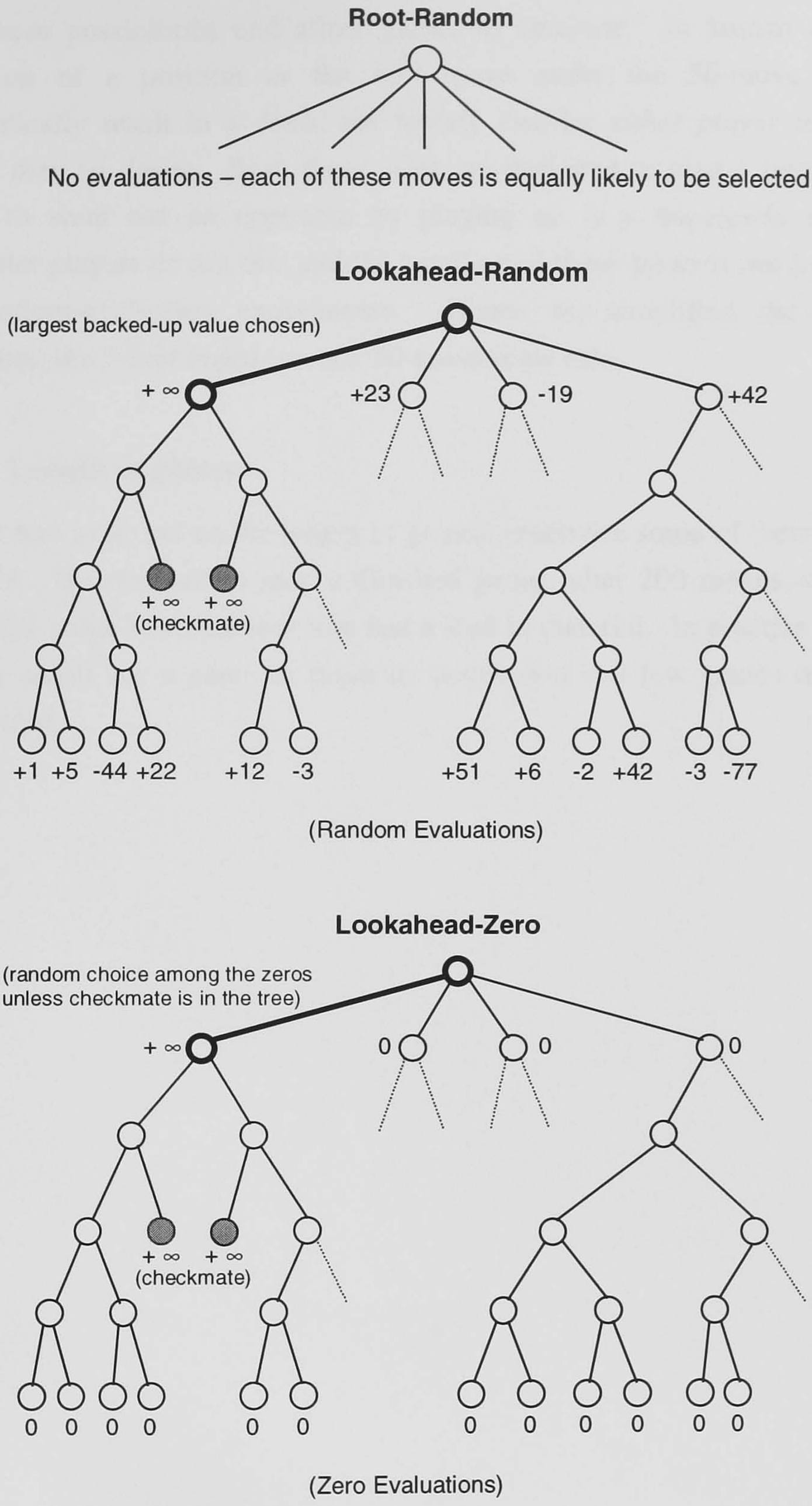

Figure E3: Schematics: Root-Random, Lookahead-Random, and Lookahead-Zero. The bold line is the branch selected.

E.4.2 Draws by repetition

One small decision that had to made was how to deal with the question of draws by repetition and draws under the 50 move rule. We decided at an early stage to ignore

both these possibilities and allow games to continue. In human chess, the third repetition of a position or the 50th move under the 50-move rule does not automatically result in a draw, but merely *enables either player to claim a draw should they so desire*. Both these rules are designed to stop a human player from trying to wear out an opponent by playing on in a hopelessly drawn position. Computer players do not tire, and the handling of these draws is not a critical factor in the random-evaluation experiments. Hence we simplified the experiment by discarding the 3-fold repetition and 50-move draw rules.

E.4.3 Length of games

A limit had to be set on the length of games, otherwise some of them might continue for ever. We decided to stop unfinished games after 200 moves, and at that time award the game to whichever side had a lead in material. In practice 200 moves was usually ample for a game to reach its conclusion and few games reached the 200-move limit.

E.5 Numerical Results

The following Tables present the experimental results in fuller detail.

LR vs. LZ	*No. Games*	*LR wins*	*LZ wins*	*Draws*	*LR%*
Depth 1	1000	222	239	539	49.2%
Depth 2	1000	389	288	323	55.1%
Depth 3	1000	331	106	563	61.3%
Depth 4	1000	953	32	15	96.1%
Depth 5	200	200	0	0	100%

Table E.1: Comparison of Lookahead Random (LR) andLookahead Zero (LZ) at various depths.

CR vs. CZ	*No. Games*	*CR wins*	*CZ wins*	*Draws*	*CR%*
Depth 1	1000	347	356	297	49.6
Depth 2	1000	591	384	25	60.4%
Depth 3	1000	829	120	51	85.5%
Depth 4	1000	794	168	38	81.3%
Depth 5	200	192	4	4	97.0%

Table E.2: Comparison of Ctree Random (CR) and Ctree Zero (CZ) at various depths.

MR vs. MZ	*No. Games*	*MR wins*	*MZ wins*	*Draws*	*MR%*
Depth 1	1000	67	61	872	50.3%
Depth 2	1000	560	439	1	56.1%
Depth 3	1000	536	422	42	55.7%
Depth 4	1000	808	164	28	82.2%
Depth 5	600	373	209	18	63.7%

Table E.3: Comparison of Material-balance Random (MR) and Material-balance Zero (MZ) at various depths.

Tables E.1 to E.3 show the results of the three experiment classes. There are 6 different types of result that we recognise:

(a) *White Wins*	White checkmated black.
(b) *Black Wins*	Black checkmated white.
(c) *Draw*	Draw by stalemate, or 4 cases of insufficient material (K-K, KN-K, KB-K, KNN-K).
(d) *White wins (at 200)*	The game reached 200 moves, white was ahead.
(e) *Black wins (at 200)*	The game reached 200 moves, black was ahead.
(f) *Draw (at 200)*	The game reached 200 moves, material was level.

We ran 1000 games for each class at depths 1 to 4, and 200 at depth 5. The games were divided with competing algorithms having an equal share of playing the white pieces - though at this low level of chess being white gives very little advantage, if any.

E.6 Interpretation of the Random Evaluation Results

As can been seen from Figure E.1, the algorithms using *Random* positional evaluation functions in their search perform significantly better than their *Zero* equivalents, and this advantage increases at greater depths. As expected, at depth 1, *Random* and *Zero* perform equally well (at depth 1 the algorithms are equivalent).

Overall, in all three experiments the results show strong advantage to lookahead on random numbers compared with the root-level-only use of random numbers. It could be argued that scoring games with a material advantage at 200 moves as a win for the side with the material advantage is unsatisfactory, and might have overstated the advantage. We counter by noting that this scoring is occurs rarely and is of marginal impact when invoked. Less than 5% of the games at depth 2 and above reached 200 moves. (Depth-1 is excluded because the games are random on both sides and no bias is being measured). In most of the games reaching 200 moves the material advantage was 5 pawns or more. Changes in the result categories for the remainder are very unlikely to affect the conclusions presented here.

The graphs also show clearly visible swings from ply to ply. The most extreme occur in *MB-Random* from depth 4 to 5, counter to the overall trend. Fluctuation of behaviour with the parity of ply (odd/even) is a familiar phenomenon in minimax searches *to fixed depths*. It is probably caused by evaluations being biased either towards or against the player to move. The direction of bias and its magnitude depend on the evaluation function and characteristics of the game.

The large swings in *MB-Random* are undoubtedly associated with the large swings that occur in material score when conducting fixed-depth searches. (This is the reason that quiescence searches are employed in chess programs). Depth-6 results can be expected to follow the pattern set by the depth-3 to depth-4 transition – i.e. a large increase in strength for lookahead-random (LR in Table E.1).

Supplementary evidence that systematic parity-of-ply fluctuation is at work can be derived from Figure E.4, the graph of game length averaged over all games in all three experiments. It can be seen that games played with an odd search depth last on average longer than those with an even search depth. This probably has to do with the question of which side is to play at the search horizon. This determines whether it is more likely that forced play will be seen first by the side that can play it, or by the side that can avoid it.

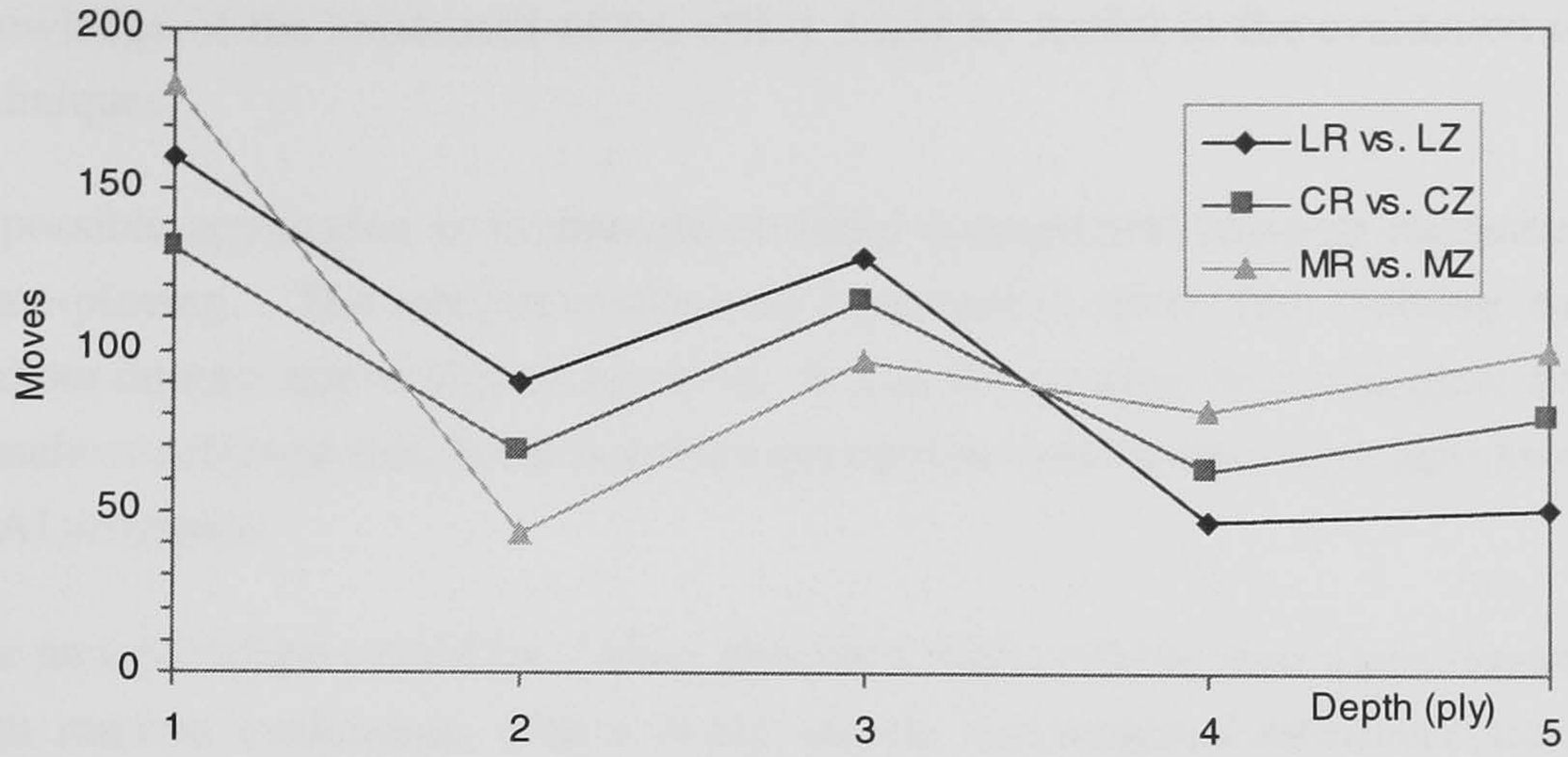

Figure E.4: Average lengths of games, by experiment.

E.7 Why Does the Effect Occur?

Once the effect is pointed out, it does not take long to arrive at the conclusion that it arises from a natural correlation between high branching factor in the game tree and higher numbers of winning moves (i.e. high branching factor in the tree of winning moves) at winning positions. In other words, mobility (in the sense of having many moves available) is associated with better positions. This would be typical of most games, although it would doubtless be possible to find atypical games in which this was not the case. The branching factor influences the random value obtained from lookahead: the more branches there are, the more likely it is that a high random value will be found.

One might suggest that random numbers are merely an inefficient way to estimate mobility. This is true in one sense, but deep lookahead on random evaluations does more subtle things where the effect is not easily quantified. For instance, it responds to available mobility at all depths – alternative branches from positions near the root of the search contribute on an equal basis with alternatives occurring deeper in the tree. In contrast, a program with minimax lookahead using an evaluation function explicitly counting branches to obtain a value would only respond to choices available at the horizon.

E.8 Possible Applications of the Effect

It is unlikely that random numbers have much practical use in game-playing. However, there are some places where either the effect itself might be useful, or

knowledge of the existences of the effect might be useful in the evaluation of other techniques.

A possible application is in attempts to build very-general learning mechanisms for game-playing. The random-evaluation experiments show that learning can start without domain knowledge of any kind. It was always clear that biological evolution somehow achieved this, but it is not always obvious how to start from zero knowledge in AI programs.

The new paradigm would be: "when presented with a totally new game domain, start with random evaluations with a N-ply search, use temporal-difference learning to develop new functions that mimic the backed-up random values, replace the evaluation function, and repeat".

It is possible that this strategy will soon have an environment in which it can be tested. There have been recent proposals for computer game-playing tournaments in which the rules of newly-invented games are given for each contest. Such tournaments will require non-game-specific mechanisms, and will have to start from zero domain knowledge each time (Pell 1992).

A minor way that random-evaluation technique could have a practical application, even in high-performance game-playing programs, would be the introduction of new evaluation terms against a background of random evaluation instead of zero.

What this means is: suppose a current evaluation function consisting of just a material term, **M**, and pawn-structure term, **P**, was in use. The evaluation at each leaf node would be **<M + P>**. With the superiority of random values over constant values in mind, a better evaluation will be **<M + P + random>**, with a suitably small weight for **random** to ensure that changes in **M+P** always dominate. So the existing terms *plus* **random** should be the baseline for introducing new terms. A proposed new term, open-file-control (**OFC**) say, should be compared with **random** by testing the function **<M + P + OFC + smaller_random>** against the existing function **<M + P + random>**. (The process of testing is complicated in practice by the need to adjust weight vectors to find the best combination for the newly-augmented set of terms, but that is ignored here.)

If the enlarged evaluation is superior to the simpler one, then it should be retained, and further evaluation terms considered. At each stage each putative evaluation term would, in effect, be compared with **random**, which differs from present-style testing, that effectively compares with **zero**. Evaluation terms of marginal value might fail to

justify their existence against **random**. This would provide a new criterion of 'minimum utility' for evaluation terms.

E.9 Discussion

These results demonstrate a counter-intuitive effect, namely that a chess program with no chess-specific knowledge whatsoever, using random numbers as evaluation functions, can play purposefully, and improve its performance with increasing lookahead. It brings to mind the old saying sometimes applied to human chess: "any plan is better than no plan".

The results also show that combining a random term with existing evaluation terms can also be beneficial. Although they may be beneficial, random evaluations are poor evaluations - in chess they are vastly inferior to typical evaluation functions in current programs. However, it is suggested that the strong effects observed in some of the experiments could have implications for game-playing programs, either because they permit zero-knowledge approaches to game playing or because they provide a criterion for minimum significant utility for additional terms in a proposed evaluation function.