The Nature of Minimax Search

The Nature of Minimax Search

DISSERTATION

to obtain the degree of Doctor at
the University of Maastricht,
on the authority of the Rector, Prof. Dr. A.C. Nieuwenhuijzen Kruseman
in accordance with the decision of the Board of Deans,
to be defended in public
on Friday June 11 1999 at 12.00 hours

by

Donald Francis Beal

SIKS Dissertation Series No. 99-3

ISBN 90 62 16 6348
NUGI 855
Subject headings: artificial intelligence/games/search
© Institute of Knowledge and Agent Technology, UM
Cover design: Van Spijk, Venlo

Contents

Preface

This thesis would never have appeared but for the encouragement and intervention of Professor Jaap van den Herik. Many years ago, I had been appointed to my London University lectureship with just a Masters degree. At that time, only a minority of the computer science lecturers had a doctoral title. For many years, I had been content to fulfil my lectureship role of teaching and research without much concern for titles. Over the years however, expectations had changed, and it would now be extremely unusual to appoint a new lecturer without a doctorate. When I commented on this in a conversation in London in 1997, Jaap immediately said "you have so many published papers and scientific credentials – you should make a PhD from your results – I'd be pleased to invite you to Maastricht to do that".

Perhaps to his surprise, a few months later I decided to take up his suggestion. True to his word, he was most supportive and directly helpful in the thesis production. Jaap has created an environment at Maastricht which provides copious encouragement and support for the many PhD students past and present who have benefited thereby. I was delighted to join them, and experience for myself his detailed contributions to technical content and expression of ideas, plus encyclopaedic knowledge of the literature acquired from his editorial roles during the past twenty years. I also recognise the past contributions of the late Professor Bob Herschberg, whose editorial pen clarified and improved ICCA Journal articles over so many years.

I am grateful to the University of London which provided the research opportunities over the years, and the UK funding council SERC which supported some of the earlier research now reflected in this thesis. Particular acknowledgement is due to Professor Mike Clarke (sadly, now prematurely deceased) who held the research grant that enabled me to work on pattern knowledge in KPK, reported in Chapter eight.

The typing of this thesis was enormously assisted by Sabine Vanhouwe, whose rapid and efficient help I am pleased to acknowledge.

Lastly and most importantly, I would like to thank my parents, family and friends for their forbearance throughout all the times when I have spent so much time on academic work.

Don Beal,
London/Maastricht 1998/9

Chapter 1

Introduction

Minimax, as presented by von Neumann in 1928, is a theoretical cornerstone for solving two-sided, zero-sum, perfect-information competitive problems. Minimax search is an essential component of almost all of today's programs that play games at the performance level of expert humans (although alternatives have been tried: Junghanns (1998) contains a survey). It remains an essential component even for programs that also incorporate large amounts of knowledge derived from a human-like approach to understanding game states and move choices (Winkler and Fürnkranz (1998) give examples of such programs). However, the computation time required for minimax search is a limiting factor for the performance of all game-playing programs, including IBM's DEEP BLUE, which resorted to hundreds of special-purpose silicon chips to scan the gargantuan search trees required to play chess at World Championship level (Seirawan, 1997).

In one sense, minimax search is well understood today. All programmers use it and there is a large literature, surveyed in, among others, Van den Herik (1983), Marsland (1986, 1993), and Scheucher and Kaindl (1998). Yet in another sense, as commented in Scheucher and Kaindl (1998), we still do not fully understand the underlying principles. It can be observed that, even with all the refinements embedded in the literature, the best game-playing programs require enormous searches to play complex games well. It is not known whether such large searches are essential to match the performance of human brains on these tasks, or whether better algorithms might require only modest computers.

1.1 Problem Statement

This thesis is concerned with the principles underlying minimax search. The motivation is the belief that despite all the published refinements to algorithms incorporating minimax search, a careful understanding of the fundamentals of minimax will yield even better algorithms. The problem statement of this thesis is "can we increase our understanding of minimax search and use it to improve search algorithms?" This problem statement leads to two inter-related research questions: (1) can we formulate models of minimax search that increase our understanding of how minimax search works, and (2) can we use the minimax models to create effective search algorithms?

1.2 Some Early History

The theory of minimax is commonly referenced as going back to 1944[1]. In that year a well-known publication by von Neumann and Morgenstern described how the process called minimax could be used to identify the final outcome, and a best move, for all positions in a perfect-information game. However, that minimax process requires starting at positions where the game ends (and hence where the outcome is known exactly). The process visits all game-terminal positions that could be reached, in order to work backwards, assigning values to all intermediate positions, towards the starting position, eventually assigning an evaluation there. This is infeasible for complex games. Van den Herik (1997) estimates that, even under extremely favourable assumptions about the size of the search tree needed, the chess starting position would take 100 centuries to evaluate using a computer evaluating 10^{12} positions per second.

The same method of working backward from distant positions to assign a heuristic evaluation (and a heuristically-best move choice) to a position under investigation was suggested as a practical method of choosing moves by Norbert Wiener (1948). It was found to work quite well, even when the 'terminal' positions were only few moves away, and the 'outcome' was not an exact value, but a crude, error-prone, erratic guess at the true value. 'Terminal' and 'outcome' are given here in quotes because the original minimax as defined by Von Neumann is based on exact values from game-terminal positions, whereas the *minimax search* suggested by Wiener is based on heuristic evaluations from positions a few moves distant, and far from the

[1] The question of the correct historical attribution of the first published exposition of the minimax concept is a complex one. Jaap van den Herik's Ph.D. thesis (1983) contains a detailed account of the known publications on this topic. It concludes that although von Neumann's name is usually associated with the concept (von Neumann, 1928), primacy probably belongs to Borel (1921), although there is a conceivable claim that the first credit should go to Charles Babbage (Morrison and Morrison, 1961)!

end of the game. This important idea was the starting point for five decades of research into game-playing algorithms, including that presented in this thesis.

1.3 Later Developments

Minimax search seemed to be effective at improving any imperfect evaluation function at the price of the computation time required to look ahead to all the 'terminal' positions and propagate appropriate resultant 'outcomes' backwards to the position being investigated.

Curiously though, when some people (myself included) first tried to analyse the expected gains from minimax search, using simple assumptions about the frequency or extent of errors, the analysis showed no gains (Beal, 1980; Nau, 1980, 1981, 1982). Worse, the analyses even indicated that minimax search should degrade the evaluation. Only a procedure that combined all the evaluations using the theory of combining probabilities in the standard multiplicative way could be expected to preserve the quality of the raw evaluations (Pearl, 1980). Yet minimax search was successful in practice almost universally. Surprisingly, it can still improve evaluations even when the 'evaluations' are completely random and bear no relationship to the true values (Beal and Smith, 1994).

Clearly the first attempts to understand the nature of minimax search had missed some essential aspect, rectified later (Beal, 1982; Bratko and Gams, 1982). It is highly desirable to understand minimax thoroughly, because the computation times for deep search are a limiting factor in performance. If the search process could be improved, it should lead to immediate performance improvements. Moreover, it seemed there might be general (i.e., domain independent) algorithm improvements to be found if there was a better understanding of the principles. Over the years, many programmers had found heuristic pruning (or extension) methods to improve the performance of their programs – e.g., N-best-moves-only (Greenblatt *et al.* 1967), forward pruning (Slagle, 1971), razoring (Birmingham and Kent, 1977), capture search (Slate and Atkin, 1983). These methods were to some extent anticipated by Shannon's (1950) idea of a Type-B strategy. A quantification of some popular methods appeared in Beal and Smith (1995). All of the methods modify the set of positions examined so that the search becomes selective, or in other words, goes to more distant positions in some parts of the tree, and stops earlier in others. Perhaps there is a theoretical basis for the success of these methods?

This thesis is the story of a number of visits to questions about minimax. The motivation was that a better understanding of the minimax process might assist the goal of pushing the boundaries of computer performance towards the best levels achieved by humans, or even higher. It is a story that spans a long time. It began over twenty years ago when I became interested in improving a computer chess program by using game independent, theoretically sound methods (in addition to as many practical tricks and heuristic concepts as I was able to program). The accumulation of those occasional visits to minimax models and theory is now presented here, together with some practical issues. Most of the chapters have their origins in previous publications, which are identified by footnotes where appropriate.

1.4 Thesis Structure

The thesis structure is as follows. Chapters 2 and 3 describe models of tree structures and the values processed by minimax search. Chapter 2 shows how a simple but plausible model predicts worsening performance with search depth. A solution to the mystery is offered by Chapter 3, which extends the model to reflect an important additional property of real game trees. The essential addition to the model is the clustering of values. Instead of uniformly distributed values, the model assumes that sibling nodes are more likely to have the same value than unrelated nodes. With the extended model, although it is still very simple, the analysis shows that we can now expect minimax search to improve evaluations, and that the advantage will increase with depth of search. This model captures an essential characteristic of typical game trees. Chapters 2 and 3 also contain some fruits of theoretical analysis in the form of game-independent algorithms (consistency search and locked-value search) derived from a theoretical perspective. These chapters address both research question one (improved models) and research question two (better algorithms) of this thesis.

Chapters 4-8 address research question two. Chapters 4 and 5 discuss some technical details of minimax search procedures that are appropriate when the search handles bounds and value ranges instead of single values. If superior-quality evaluations are available in a restricted subset of states, minimax can still propagate all possible knowledge about the scores from terminal positions towards the tree root, but it requires backing up three values instead of one. A study of the way minimax handles and compares the values produced by the search process reveals that, if desired, bounds can be unified with exact scores into a single number for coding convenience.

Chapter 6 amplifies the reasons why evaluations might vary in quality due to known reasons, and considers the technical administration of more than two levels of evaluation quality. Chapter 7 moves from analysis to exploitation by introducing a simple technique of selectivity, and discusses results from applying it to the specialised task of mating attacks in chess.

Chapter 8 arises from an investigation into solving the smaller problems within chess. Minimax is still required, but when the domain is small, complete cataloguing and exact pattern processing become feasible (*cf.*, Van den Herik and Herschberg, 1986 – Omniscience the Rule-giver?). Unfortunately no way has yet been found to scale up this approach to assist with solving the main part of chess, or other large games (*cf.*, Fürnkranz, 1996).

Chapter 9 returns to research question one and discusses the concept of search envelope. This concept is introduced to separate the rules for terminating the search from the alpha-beta rule. Alpha-beta prunes branches from the tree, but does not change the backed-up value. This contrasts with changes to the search envelope, which may change the backed-up minimax value. Search algorithms implicitly define a horizon (the set of 'terminal' nodes) determined by depth or other criteria. Every horizon node that the search visits (or would visit if it were not for alpha-beta) lies on the envelope for that search. The key question "how cost-effective is a search algorithm?" is intimately linked to the question "what is the search envelope?" Consistency search and locked-value search are examples of algorithms with search envelopes determined by values the search finds rather than determined by depth.

Chapter 10 addresses research question two, and plays a major role in this thesis. As a consequence of finding a reason why minimax search is useful for choosing moves, the minimax model of Chapter 3 also gives an explanation of why not all portions of the search tree are of equal value. The model embodies a hierarchic clustering of values. Below the root node of a cluster, there is only a low probability that changes to node values below the cluster root will propagate above the cluster root. In the absence of clusters, changes to node values propagate upwards with relatively high probability. Could this characteristic be used to reduce the search depth below cluster roots, while extending the search depth in 'unstable' parts of the tree? There are several answers to this question. Some of them lead to different search algorithms that have been found successful to a greater or lesser extent. Chapter 10 discusses the general concept of quiescence which arises from such considerations and focuses on the most successful algorithms – those based on the null move. Null-move quiescence, as discussed in Chapter 10, is a general algorithm applicable to any minimax search. It is based on the insight that null moves establish bounds which have a greater reliability than the evaluations which give rise to the bounds. The bounds (which should not be confused with alpha-beta bounds – they are different), enable the search to terminate earlier down some lines than others. Null-move quiescence is therefore a domain-independent selective search.

Chapter 11 draws some conclusions from this work, surveys current work and sets some goals for the future.

Chapter 2

A Preliminary Analysis of Minimax[1]

Minimax search using a heuristic evaluation function is known to be an effective technique for game playing. This chapter presents an analysis of a very simple minimax model which yields the surprising and unsatisfactory conclusion that fixed-depth backed-up values are slightly *less* trustworthy than heuristic values themselves. Six possible reasons are offered.

This chapter also presents a general algorithm arising from the ideas of error probability used in the minimax model. It is based on simple assumptions about the probabilities of error in terminal and backed-up values. It searches without depth or width limits until it achieves a backed-up value of increased reliability. Two well-known and apparently different specific algorithms used successfully in practical chess-playing programs correspond to this general algorithm.

2.1 A Simplified Minimax Model

The model is constructed to be sufficiently simple to analyse so that the following question can be answered: By how much are backed-up values from a fixed-depth minimax search more reliable than the heuristic values themselves?

[1] This Chapter is an edited version of Beal (1980) titled An Analysis of Minimax, in *Advances in Computer Chess 2*. Thanks are due to Edinburgh University Press for permitting its use here.

Our assumptions are:

1. The tree structure has a uniform branching factor b.
2. The node values are either: a win for player A
   or: a win for player B
3. True values conform to the minimax relationship.
4. Values are distributed so that at each level, the proportion of wins of the player to move is the same. (This enables the effect of n-ply search to be deduced directly from the effect of 1-ply search.)
5. Heuristic values are usually identical to true values, the proportion of erroneous values being small compared to $1/b$.
6. Heuristic values have a probability of error p, independent of the distribution of true values.

Values are denoted here by + and –, but note that + denotes a win for the player to move, rather than for either player A or player B and that – denotes a loss for the player to move.

Assumptions 1, 2 and 3 establish a simple tree with minimax values. Assumptions 4, 5 and 6 characterise this particular model. We achieve (4) by defining the ratio of wins and losses as follows. Let terminal nodes have a probability k of being unfavourable for the player to move. That is, the proportion of – nodes is k, and therefore the proportion of + nodes is $1 - k$. These ratios will be the same for all levels in the tree if k is the solution of:

$$(1-x)^b = x$$

This definition of win/loss ratios ensures that the proportion of wins for either player is the same at depth $d + 2$ as at depth d, and the inverse of the proportion at depth $d + 1$.

2.1.1 Analysis

What are the probabilities of error in 1-ply backed-up values? In order to answer this question conveniently, we split the possible occurrences of true values into three cases (see Figure 2.1), chosen because there exists a simple formula for each case, as given in Table 2.1. The table also gives an appropriate simplification derived from assumption 5, that error rates are low. This assumption enables us to ignore the possibility that two descendants of a node are simultaneously in error.

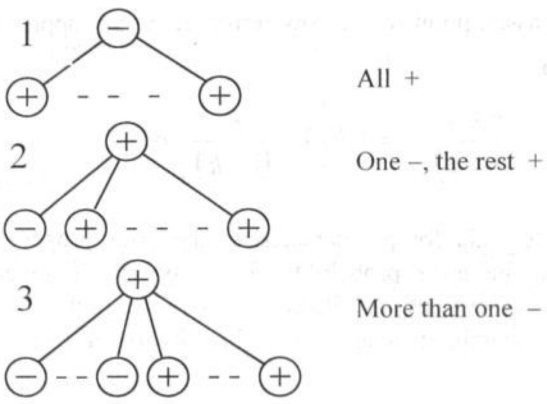


Figure 2.1: The three error cases.

	Probability of occurrence	Probability of error in backed-up value
1.	$(1-k)^b = k$	$1-(1-p)^b$
2.	$b.k.(1-k)^{b-1} = \dfrac{b.k.k}{(1-k)}$	approximately p, since the assumption of low error rates means that more than one descendent node in error is second order
3.	$1-k-\dfrac{b.k.k}{(1-k)}$	0, by assumption of low error rates

Table 2.1: Error probabilities.

Case 1 gives rise to the only – nodes at the 1-ply level. Case 2 gives rise to only + nodes at the 1-ply level that have significant probability of error. The probability of error is approximately p, since an error in the single – descendant will propagate, and an error in any other descendant will not. Case 3 yields no significant error. In the left-hand column of Table 2.1, rows 1 and 2 give first the formula as constructed by examining the tree structure, then an equivalent formula derived by using the equation for k. Column 1, row 3 is obtained as: one minus the probabilities of cases 1 and 2.

Now we are ready to calculate the average probability of error over all nodes.

Let p' denote the probability of error in 1-ply backed-up values.

Therefore $p' = k\left[1-(1-p)^b\right] + \frac{b.k.k}{(1-k)}.p$

Again using the assumption of a low error rate, an approximation to $1-(1-p)^b$ is $b.p$.

Hence $p' \approx k.b.p + \frac{b.k.k}{(1-k)}.p = k.b.\left[1+\frac{k}{(1-k)}\right].p$

This approximate formula for p′ characterises the error propagation in the model. It tells us the error probability for 1-ply search, given an error probability for terminal evaluations. Raised to the power n, it tells us the error probability for n-ply search, given an error probability for terminal evaluations.

Some approximate numerical values for $\frac{p'}{p} \approx k.b\left[1+\frac{k}{(1-k)}\right]$ are given in Table 2.2.

b	*p′/p*
2	1.24
5	1.62
10	1.97
20	2.37
40	2.82

Table 2.2: Example values for *p′/p*.

One might notice that a further (but crude) approximation, if *k* is small, may be made by taking $k = \frac{1}{(b+1)}$, which makes $k.b\left[1+\frac{k}{(1-k)}\right]=1$. With this approximation, there would be no increase in the probability of error with increasing search depth. However, this approximation would be an invalid simplification, because k is determined by the branching factor, unlike the error probability which can be arbitrarily low.

A realistic approximation of $\frac{p'}{p}$ for large *b* may be made using the fact that *k* asymptotically approaches $\frac{(\log b)}{b}$ for large *b*. Hence p' approaches $(\log b).p$.

2.1.2 Discussion

The analysis of the minimax model above shows that for large branching factors, the probability of error is *increased* by the logarithm of the branching factor for every ply of search. This result is disappointing. It was hoped that the analysis would show that the probability of error reduced with backing-up. Clearly this model does not capture some essential property of minimax search in typical game trees. It is not so clear, however, why this is so.

The analysis is of fixed-depth search and there are both empirical and intuitive grounds for believing that a suitable selective search will be far more effective than a similar-size fixed-depth search. Nevertheless, even fixed-depth search is well known to yield substantial benefits in practice and so the fixed depth cannot be singled out as a satisfactory reason for the negative result.

The following might be thought possible explanations:

1. Errors in heuristic values may not be distributed independently of the true values: they may be related in such a way as to render minimax search more useful.
2. The notion of probability of error as a measure of the usefulness of minimax values may be inappropriate. (Game-playing requires the choice of a move, not a judgement about the absolute value of positions.)
3. The distribution of true values in typical game trees may differ from the model in such a way as to render minimax search more useful.

4. The restriction to two values may be too extreme. Perhaps a range of values is needed so that errors in the heuristic value might be graduated, with large errors being less likely than small errors.
5. It may be necessary to assume different ranges of values for heuristic values and true values. For instance, true values of +1 and –1 but also heuristic values drawn from the real numbers in between.
6. The assumption of very low probabilities of error may be inappropriate.

The problem is intriguing: minimax search is so well-known to be useful that one expects a simple convincing explanation of why.

There have been investigations into specific anomalies of minimax search. Nau (1980) proved that there is an infinite class of game trees for which deeper searching produces worse evaluations. Nau (1981) describes a game, based on a randomly chosen initial configuration, for which an undoubtedly reasonable evaluation function gives worse play with deeper search. These are potentially illuminating, but do not explain why minimax search on typical game trees is successful.

2.2 Consistency Search

Fixed-depth search, despite the analysis above, is known in practice to be beneficial, and can be assumed to decrease the probability of error.

What I have termed consistency search is a method of decreasing the probability of error by selective searching. It is based on the following four assumptions.

1. The true values of the tree have a minimax relationship.
2. Each node has a heuristic value equal to the true value plus an error value.
3. The error rate is low.
4. All error values have a probability p of being non-zero, independently of values at any other nodes.

We define a node to be *consistent* if its heuristic value is the same as the backed-up value from a 1-ply search over its descendants.

Assumption (4) implies that a 1-ply backed-up value and the heuristic value applied directly are independent estimates of the node's value. Taking the apparently conservative assumption that 1-ply backed-up values have the same probability of error as heuristic values themselves (ignoring the disturbing result of Subsection 2.1.1), the probability of error if they agree is p^2. Hence consistent values are much more reliable than direct heuristic values.

A consistency search returns a value derived from consistent values. Its simplest implementation is a depth-first search terminating only at consistent nodes, as illustrated by the pseudo-program in Figure 2.2:

```
      HV  =  heuristic value for the current node
(assumed to use the 'negamax' convention under which numerically-
greater values are better for the current side to move, rather than
for a fixed side).
      CV1()
      {  v ← -INF
         foreach branch  b  do
         {  traverse(b);  v ← max(v, -HV);  retrace(b)  }
         if (v = HV)  return(v)
         v ← -INF
         foreach branch  b  do
         {  traverse(b);  v ← max(v, -CV1);  retrace(b)  }
         return(v)
      }
```

Figure 2.2: Simple consistency search.

Figure 2.3 shows a version taking advantage of alpha-beta cut-offs:

```
HV2(α, β) = heuristic value bounded by α and β
CV2(α, β)
{      v ← α
       foreach branch  b  do
       {  traverse(b);  v ← max(v,-HV2(-β, -v));  retrace(b)  }
       if (HV2(α, β) = v)  return(v)
       v ← α
       foreach branch  b  do
       {  traverse(b);  v ← max(v,-CV2(-β, -v));  retrace(b);
          if (v = β)  return(v);
       }
       return(v)
}
```

Figure 2.3: Consistency search using alpha-beta.

2.2.1 Consistency Cut-offs

In addition to alpha-beta cut-offs it is possible to exploit consistency cut-offs. These can occur at inconsistent nodes after one or more descendants have been searched. If the reliable value(s) thereby found render the node consistent when used in place of the direct heuristic values for those descendants, there is no need to search the remaining descendants. An example of inconsistency is given in Figure 2.4.

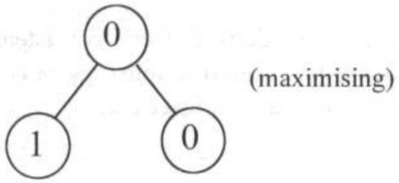


Figure 2.4: An example of an inconsistent node.

Suppose that, after searching below the left-hand descendant, a consistent value of 0 is backed up for it. This consistent value is much more reliable than the direct value of 1. If it overrides the direct value, the inconsistency of the original node disappears. The right-hand node can be cut off.

These consistency cut-offs do not reduce the expected reliability of the result (although, unlike alpha-beta, they may change it).

To incorporate consistency cut-offs into the search requires storing the heuristic values of all the descendant nodes if repeated evaluations are to be avoided. The stored values can be used to order the branches. This has the important effect of ensuring that only the errant node is searched at single-deviant nodes. A single-deviant node is defined as an inconsistent node that has only one high-valued descendant. Such nodes occur frequently.

The pseudo-program in Figure 2.4 illustrates a consistency search taking advantage of consistency cut-offs.

```
CV3()
{       array B
        function maxB = largest value in B

        foreach branch  b  do
        { traverse(b);  B[b] ← -HV;  retrace(b)  }
        hv ← HV
        if (maxB = hv)  return(hv)
        sort B and the branches into descending order
        v ← -INF
        foreach branch  b  do
        { traverse(b); B[b] ← -CV3(); v ← max(v, B[b]); retrace(b)
          if (maxB = hv)  return(hv)
        }
        return(maxB)
}
```

Figure 2.4: Consistency search with consistency cut-offs.

A version using alpha-beta is given in Figure 2.5.

```
CV4(α, β)
{ array B
  function maxB = largest value in B
  foreach branch  b  do
  {  traverse(b);  B[b] ← -HV2(-β,-α);  retrace(b)  }
  hv = HV2 (α, β)
  if (maxB = hv) return(hv)
  sort B and the branches into descending order
  v ← α
  foreach branch  b  do
  {  traverse(b); B[b] ← -CV4(-β,-v); v ← max(v, B[b]); retrace(b)
     if (v = β) return(v)
     if (maxB = hv) return(hv)
  }
  return(maxB)
}
```

Figure 2.5: Consistency search with consistency cut-offs and alpha-beta.

All these versions of consistency search are intended to show clearly the basic algorithm. They need some revision before they become efficient. In particular, in the case of capture tree search in chess described below, obtaining the values of the heuristic function down each branch by traversing the branch, applying the heuristic function, and retracing the branch is unnecessary. The change in material balance is readily obtainable as a by-product of generating the capture moves.

2.2.2 Practical Algorithms

This Subsection illustrates consistency search by interpreting two well-known practical algorithms, viz. Capture Tree search in chess and Forward Pruning as consistency searches.

Capture Tree Search in Chess.

The usual way to perform a complete capture tree search, as implemented in several chess programs, embodies the following rules.

(a) All captures are searched.

(b) A position where the side to play has no captures is terminal, and the static material balance value is taken.

(c) At any node, if all the captures for the side to play prove unsuccessful, the static material balance value is taken.

Of course, plausible assumptions 3 and 4 of Section 2.2 might not be true for capture tree search in chess.

Nevertheless, capture tree search corresponds to consistency search using material balance as the heuristic evaluation. Rule (b) halts at consistent nodes, and rule (c) corresponds exactly to consistency cut-off.

Forward Pruning

The basic principle of forward pruning is to cut off descendant nodes with a static evaluation lower than the current best backed-up value. The static evaluation is typically complex and there are many variants of the pruning rule. Optimistic and pessimistic versions bias the comparison to encourage or discourage cut-offs. Often the bias varies with depth. The authors of the early chess program MASTER called their version razoring (Birmingham and Kent, 1977).

The basic principle can be interpreted as the consistency cut-off rule within a consistency search.

However, forward pruning only becomes equivalent to consistency search if the pruning rule obeys one other constraint, viz. that it should not be applied if the static value at the current node is greater than the current best backed-up value. If this is not so, the node is still inconsistent.

2.3 Summary

This chapter has done two things. (1) It has analysed a simple model of minimax and found that the model is too simple to reflect the benefits of minimax search observed empirically. (2) It has used the idea of probability of error to define a search algorithm (consistency search) that seeks values having reduced error probability. It therefore addresses both research questions identified in section 1.1. Consistency search closely corresponds to two search algorithms known to be successful in practical chess programs.

Chapter 3 presents a slightly extended, and more realistic, minimax model. In this model, search yields increasing benefit with increasing search depth.

2.2.2 Practical Algorithms

This subsection illustrates consistency search by interpreting two well-known practical algorithms, [illegible]

Capture Tree Search in Chess

The [illegible] several [illegible]

(a) All captures are explored.

(b) A position where the side to play has no captures [illegible] static material balance value is taken.

(c) At any node, if all the captures for the side to play prove unsuccessful, the static material balance value is taken.

Of course, plausible assumptions 2 and 4 of Section 2.2 might not be true for capture tree search in chess.

Nevertheless, capture tree search corresponds to consistency search using material balance as the heuristic evaluation. Rule (b) halts at consistent nodes, and rule (c) corresponds exactly to consistency cut-off.

Forward Pruning

The basic principle of forward pruning is to cut off descendant nodes with a static evaluation lower than the current best backed-up value. The static evaluation is typically complex, and there are many variants of the pruning rule. Optimistic and pessimistic versions bias the comparison to encourage or discourage cutoffs. Often the bias varies with depth. The authors of the early chess program MASTER called their version razoring (Birmingham and Kent, 1977).

The basic principle can be interpreted as the consistency cut-off rule within a consistency search.

However, forward pruning only becomes equivalent to consistency search if the pruning rule obeys one other constraint, viz. that it should not be applied if the static value at the current node is greater than the current best backed-up value. If this is not so, the node is still inconsistent.

Chapter 3

A More Realistic Analysis of Minimax[1]

The mathematical model of Chapter 2 failed to explain the well-established success of minimax search using a heuristic evaluation function. This Chapter presents a model for which lookahead is shown to be beneficial if the game tree conforms to a simple condition. The KPK chess endgame is used for illustration. Finally, *locked-value search* is introduced as a method of increasing confidence in evaluations for a given heuristic function.

3.1 The Aim of the Model

In Chapter 2, the mathematical model led to the surprising result that lookahead search would give less reliable backed-up values than heuristic evaluations alone. The six assumptions of the model seem reasonable, despite its simplicity. It was not clear why it was in such drastic conflict with empirical experience.

This Chapter presents a slightly extended model, for which lookahead is beneficial. When the parameters of the model are chosen to match the KPK endgame, lookahead is shown to be beneficial within the model. The motivation for this work was partly to find a basis for mathematical analysis and modelling. The other part of the motivation was to obtain understanding relevant to developing practical algorithms, if possible general ones applicable to any problem formulated as a minimax search.

[1] This Chapter is an edited version of Beal (1982) titled Benefits of Minimax Search, in *Advances in Computer Chess 3*. Thanks are due to Pergamon Press for permitting its use here.

Consistency search is a general algorithm applicable to any minimax search that meets the assumptions, and two particular cases, already well-known as effective practical heuristics, were pointed out: (1) capture tree search in chess; (2) forward pruning against static evaluations.

This chapter introduces the theoretical idea of locked-value search which is an alternative to consistency search derived from slightly different assumptions.

Both consistency search and locked-value search decide which areas of the tree to search according to the evaluations found, without width or depth limits unless these are imposed in addition. They define, in effect, a *search envelope* (*cf.* Chapter 9), very different in shape from the full-width, fixed-depth one, within which a conventional alpha-beta search can take place.

3.2 An Improved Minimax Model

The improved model adds the following assumption (assumption 7) to the assumptions of the preliminary model given in Section 2.1.

7) The values across each level are randomly distributed but with a tendency to form clusters of identical values (The previous model assumed that values were randomly distributed in a uniform way).

As before node values are denoted by $+$ and $-$, with $+$ meaning favourable for the side to move, rather than for either player A or player B, and $-$ meaning unfavourable for the side to move.

Let k denote the proportion of $-$ nodes. Assumption 4 is that k is constant over all levels in the tree.

The clustering effect of assumption 7 is induced by assuming that a fraction f of the nodes at any level are grouped into families where all members have the same value. ('family' is used here to mean all the descendants of a single node.) A proportion k of the cluster families have $-$ values. The remainder of the nodes at that level receive their values randomly according to the k proportion.

Since $-$ nodes only arise if all descendants are $+$ nodes, and $+$ nodes arise whenever at least one descendant is a $-$ node, it is possible to calculate the proportions of $-$ nodes one ply higher in the tree, given the specification above. Figure 3.1 illustrates the mutually exclusive cases that occur. To achieve assumption four, the fraction of $-$ nodes at the higher level must be k. Therefore, k and f must be related as follows:

$$(1-k).f+(1-k)^b.(1-f)=k$$

where $(1-k)\cdot f$ is the fraction of $-$ nodes at the next higher level which are ancestors of the clustered part of the distribution at this level. $(1-k)^b\cdot(1-f)$ is the fraction of $-$ nodes at the next level which are ancestors of the randomly distributed part. The right-hand section of Figure 3.1 corresponds to the randomly distributed part, with the lower nodes occurring randomly in the proportion of k $-$ nodes to $1-k$ $+$ nodes.

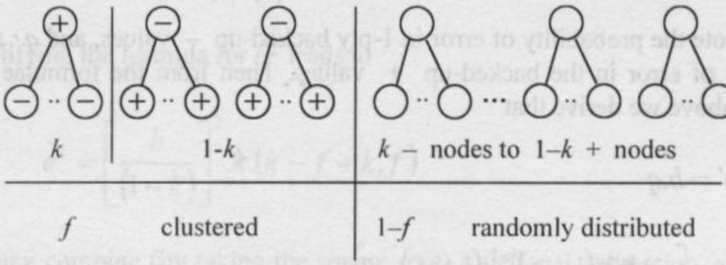


Figure 3.1: Cluster structures.

The model is parameterised by b and f. Once the branching factor and clustering factor are chosen, k is fixed.

3.3 Analysis of the Improved Model

This analysis of error propagations derives a formula for the benefit of lookahead. The analysis will treat error rates at $-$ nodes separately from $+$ nodes. (In the preliminary model this separation was unnecessary.)

Let p denote the error probability at $-$ nodes, q at $+$ nodes. To find the probability of error in 1-ply backed-up values we split the possible occurrences of family values into the same three cases as used in the preliminary model (see Figure 2.1).

	Probability of occurrence	Probability of error in backed-up value
1.	$(1-k)\cdot f+(1-k)^b\cdot(1-f)$	$b\cdot q$
2.	$b\cdot k\cdot(1-k)^{b-1}\cdot(1-f)$	p
3.	$k\cdot f+[1-(1-k)^b-b\cdot k\cdot(1-k)^{b-1}]\cdot(1-f)$	0

Table 3.1: Error probabilities.

The formulae of Table 3.1 for the probability of error in the backed-up values are approximate, based on assumption 5 (of Section 2.1) that error rates are low, which means that more than one descendant node in error is of second order.

Let p' denote the probability of error in 1-ply backed-up $-$ values, and q' the probability of error in the backed-up $+$ values. Then from the formulae in Table 3.1 above we derive that

$$p' = b.q$$

and
$$q' = \left[\frac{b.k.(1-k)^{b-1}.(1-f)}{k.f+\left[1-(1-k)^b\right](1-f)}\right].p$$

Let p'' denote the probability of error in 2-ply $-$ values and q'' the probability of error in the analogous $+$ values. Then we derive that

$$p'' = b.\left[\frac{b.k.(1-k)^{b-1}.(1-f)}{k.f+\left[1-(1-k)^b\right](1-f)}\right].p$$

and similarly, that

$$q'' = \left[\frac{b.k.(1-k)^{b-1}.(1-f)}{k.f+\left[1-(1-k)^b\right](1-f)}\right].b.q$$

Thus, for 2-ply lookahead, the effect on probability of error in $-$ values is the same as the effect on probability of error in $+$ values. Although one ply of backing-up has an asymmetric effect on probabilities of error at $-$ and $+$ values, the next ply has the opposite asymmetry. So a 2-ply lookahead affects error probabilities without disturbing the relative probabilities of errors in $-$ and $+$ values.

Let e^2 denote the major expression in the derivation above

$$e^2 = b.\left[\frac{b.k.(1-k)^{b-1}.(1-f)}{k.f+\left[1-(1-k)^b\right](1-f)}\right]$$

so that $p'' = e^2 \cdot p$ and $q'' = e^2 \cdot q$. Then e^2 is the change in the error probabilities produced by 2-ply lookahead.

Simplifying the formula for e^2 leads to

$$e^2 = \left[\frac{b}{(1-k)}\right]^2 .k.(k-f+k.f)$$

We may compute (by taking the square root) a fictional expression, e, for the effect of 1-ply lookahead, which is convenient for computing the effect of n-ply lookahead.

The fictitional 1-ply lookahead error formula is

$$e = \left[\frac{b}{(1-k)}\right].[k.(k-f+k.f)]^{\frac{1}{2}}$$

When $f = 0$ (no clustering) the formula reduces to

$$\left[\frac{b}{(1-k)}\right].k$$

This is equivalent to the expression derived for the preliminary model.

It is convenient to regard the formula for e as the error reduction factor for a 1-ply lookahead. (Error reduction assumes that e is less than one: if it is greater than one, then lookahead increases the errors instead of reducing them.)

The error reduction factor of n-ply search is e^n.

Figure 3.2 presents a graph of e for $b = 16$, showing how it varies with the clustering factor.

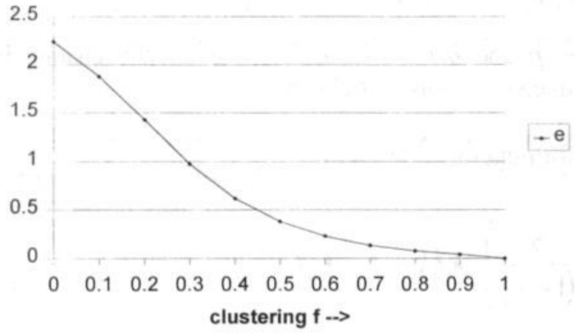


Figure 3.2: Variation of error against clustering factor.

With clustering factors above about 0.3, the lookahead produces a reduction in the error probability. The greater the clustering factor is, the greater the error reduction.

More graphs showing the 1-ply effect on error probability against clustering factor for other branching factors are given in Figure 3.3, which shows $e = \left[\frac{b}{(1-k)}\right].[k.(k - f + k.f)]^{\frac{1}{2}}$ plotted against f for other values of b.

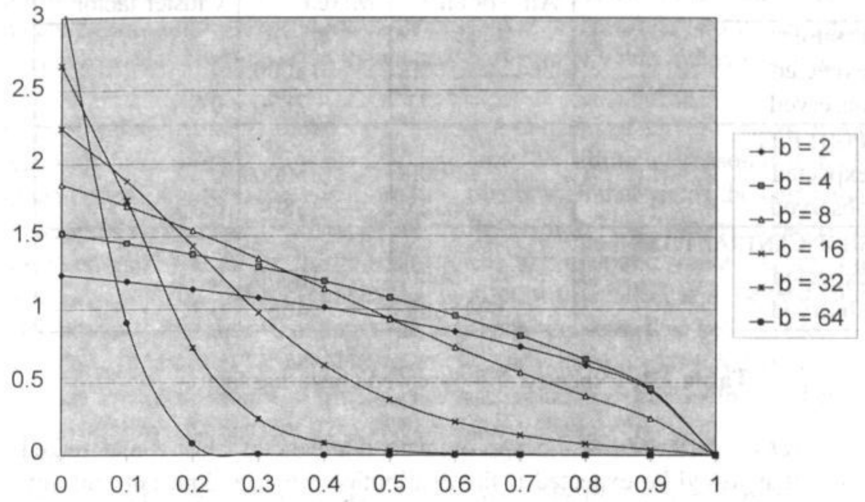


Figure 3.3: Error against clustering factor for different branching factors.

3.4 The Model compared with KPK

The King + Pawn versus King (KPK) ending is a two-valued game. Since complete data on this ending is available, it is examined here to see how much clustering of values occurs.

Positions with the Pawn on different files effectively form separate sub-endings as the Pawn can never change file. So the rook-file positions and knight-file positions are examined separately. The bishop-file and the central-file positions are treated together as they are essentially the same.

Table 3.2 shows the numbers of positions in each of two categories for the pawn on each of the different files. Each position is categorised according to the set of its descendants (its family). The categories are: descendants all the same (cluster family), and descendants mixed (mixed family).

	All – or all +	Mixed	Cluster factor
ROOK FILE			
expected	5000	15000	0
observed	17773	2870	0.8
KNIGHT FILE			
expected	5000	15000	0
observed	16525	3804	0.75
BISHOP/CENTRAL FILES			
expected	5000	15000	0
observed	15685	4661	0.7

Table 3.2: Expected and observed clustering factors.

The *expected* entries show the approximate numbers of cluster and mixed families that would be expected if the distribution were random but uniform. Only approximate numbers are given because without a uniform branching factor an exact calculation is complicated. The cluster factors given are those that would be required by the model to produce the *observed* proportions of cluster and mixed families.

From the formula for e and the graphs of Figure 3.3, it can be seen that with a clustering factor of 0.7, lookahead is beneficial at all branching factors. This may be compared with the cluster factors in Table 3.2 which show 0.7 or greater.

No claim is made that the model of this Chapter is a fully realistic model for KPK, but Table 3.2 shows that the extent of clustering in KPK is sufficient to render lookahead worthwhile given the assumptions of the model.

Note also that in the model, families which are not clusters are assumed to have unbiased distributions of values, whereas, in reality, partial clustering can be expected, biasing the proportions of the different cases of mixed families. A more sophisticated model that took account of this would show greater benefit of lookahead for the same proportion of full-cluster families.

3.5 Locked-Value Search

Consistency search (*cf.* Section 2.2) is based on the idea that (i) static evaluation at a node, and (ii) 1-ply minimax search are independent estimates of the value of that node. This idea ignores the obvious similarity of all the positions involved, yet there is some truth in it. Here 'independent' really means is "not totally dependent". The reasoning is that if the two values agree, we have greater confidence in the value than in either value if they disagree.

Locked-value search is based on a different concept of circumstances where we might have more confidence in evaluations. This time we reason that if, at a given node in the tree, there is more than one branch that yields the best value, then we have greater confidence that the true value is at least as great as the observed best value. In this circumstance, the heuristic value of any one descendant can be in error, concealing an inferior true value, and the best value still holds. If more than two descendants possess the best value, our confidence increases further. However, we only have a one-sided conclusion because any descendant might have a true value superior to the backed-up value which would cause the backed-up value to be incorrect.

If we consider two plies of search it is possible that a tree containing the subtree of Figure 3.4 might arise.

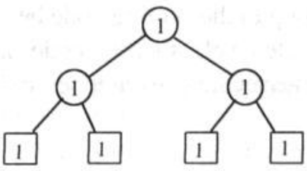


Figure 3.4: The simplest locked-value subtree.

The square nodes contain static heuristic evaluations, the round nodes contain backed-up values.

The top node's value is ≥1 with 'high' confidence because two descendants have the value 1. Analogously the left intermediate node's value is ≤1 with 'high' confidence by the complementary argument for min nodes. The same holds for the right intermediate node's value. Hence for all immediate descendants of the top node, $v \leq 1$ with high confidence. Therefore for the top node itself also $v \leq 1$ with high confidence. So we have simultaneously high confidence that the value is ≤1 and ≥1. It is locked above and below and we have high confidence it is equal to 1.

Another way to look at the tree of Figure 3.4 is to establish that no *single* error in the terminal values can change the top value.

Figure 3.5 provides a more complex example.

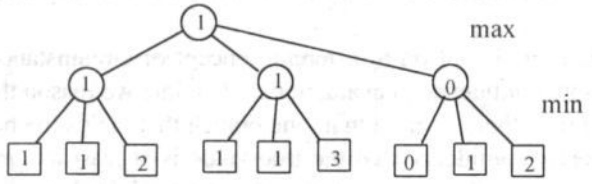


Figure 3.5: Another locked-value subtree.

Either by reasoning about "high-confidence" bounds or by observing that no single change in the terminal evaluations can alter the top value, we conclude that this top node is also locked equal to 1.

A straightforward procedure for detecting that a node has a locked value is as follows:

```
Do a 2-ply search.
Let the value backed-up to the starting node be $v$.
At ply one (intermediate level) at least 2 nodes must have backed-up
value $v$, and for all intermediate level nodes, at least 2 descendants
must have value $\leq v$.
```

The procedure may be generalised to demand n rather than 2 nodes with best value, thus setting a higher standard before placing confidence in the value.

A locked-value search returns a minimax value derived only from locked values. Its simplest implementation is a depth-first search terminating only at locked values. It may, of course, make use of alpha-beta within the search envelope delimited by the locked values.

3.6 A Conclusion on Clustering

The improved minimax model is still straightforward and the modelling of clustering is a special case amongst a range of more complex models that might be constructed.

However, it is plausible that clustering is a major feature of all practical games because game trees are derived from structures, such as the array of squares in chess, of which only a small part changes as a move is made. The true values for the game are defined in terms of the game structures and not assigned randomly. This means that positions in which a single move reverses the value of a position are rare, and that positions in which all moves lead to wins (or loss) are common. We remark that 'common' here refers to averages computed over the entire game tree, and *not* to positions that are typically reached in human play, or those which typically engage the attention of humans.

Cluster families which are themselves clustered at the next higher level of the tree form a subtree which is overwhelmingly in favour of one player. Such subtrees have low probability of error in backed-up values, and also have locked values.

Locked-value search has a strong tendency to stop early in 'clearly' advantageous to one side or the other side, and explore more deeply those subtrees where advantage is more balanced.

Bratko and Gams (1982) provide an additional analysis of the preliminary model and also conclude that clustering is the reason for the effectiveness of minimax search.

This Chapter has addressed both research questions given in Section 1.1. It has presented a model of minimax which is more realistic than the preliminary model of Chapter two and established that the more realistic model predicts benefits from minimax lookahead in the KPK endgame. It has also presented a search algorithm that reacts to clustering encountered in the search, which is a potential contribution to practical search algorithms.

Chapter 4

Nested Minimax Search[1]

The original classic form of minimax search is based on perfect values, the usual one on heuristic values. When perfect evaluations are infeasible, a tree search is made, i.e., the backing up of heuristic values from deeper nodes to produce a heuristic evaluation for the root which is more reliable than a raw evaluation. Alpha-beta is used to reduce the effort.

4.1 Perfect Values

The question addressed here is: "How should the heuristic minimax search algorithm be extended to cope with perfect values being obtained at some nodes in the tree?" Naturally, chess programs detect a perfect value for checkmate or stalemate. Since complete tables or pattern knowledge have become available for many endings (Thompson, 1997), perfect values could be encountered frequently in some endgame searches. Next to chess, this also happens in other games, e.g., checkers (Schaeffer, 1997) and awari (Allis, Van der Meulen,and Van den Herik, 1992). In the examples of this chapter, perfect is taken to mean a perfect indication of win, loss, or draw, without any depth-to-win information. Depth-to-win information could be included without affecting the argument.

There would seem to be no problem with introducing the perfect values into the backing-up process. Nodes before terminal depth may now have to be examined to see if a perfect value is available, but otherwise the search can proceed as before with an expectation of more reliable values or an earlier result, or both.

[1] This Chapter is an edited version of Beal (1984) published under the title Mixing Heuristic and Perfect Evaluations: Nested Minimax. Thanks are due to the Editorial Board of the *ICCA Journal* for permitting its use here.

However, extracting maximum advantage from the perfect values requires the search algorithm to back up three values at every node instead of one. This chapter explains why, and suggests the term *nested minimax* for the general case of a search using two evaluation functions, one much more reliable than the other, the more reliable one being not always available.

4.2 A Problem with Perfect Values.

If perfect values and heuristic values are treated uniformly in the backing-up process, information is lost. A backed-up value is obtained, but no conclusion about whether the value is perfect or heuristic can be drawn. This loss of distinction can be unacceptable.

Most minimax searches are performed iteratively, i.e., a 1-ply search is followed by a 2-ply search, a 3-ply one, *etc*., all from the same root node. (Iterative searches order the moves within an *n*-ply search according to those found best within the *n*-1 ply search. This is found to result in alpha-beta cut-offs approaching the theoretical optimum (*cf.* Knuth and Moore, 1975).)

If a search to 3-ply, say, happens to produce a perfect value, there is no need to perform the 4-ply, 5-ply, *etc*. searches at all. Thus iterative searching requires an indication at the root, whether the value is perfect or heuristic. In general, any search scheme which uses small searches to guide or contribute to a bigger search will also have a requirement for a perfect/heuristic indication.

4.3 A Partial Solution

A first thought might be to append a perfect/heuristic flag to values, so that if the value backed up happens to be perfect, this information is retained. However, this method could still lose much of the information about perfect values. Consider the tree fragments of Figures 4.1 and 4.2.

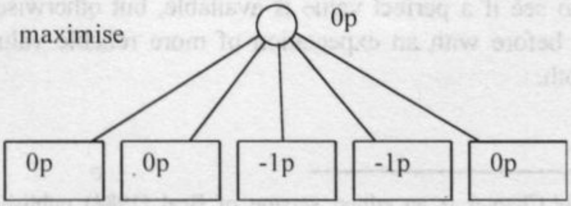


Figure 4.1: A 1-ply tree with perfect values.

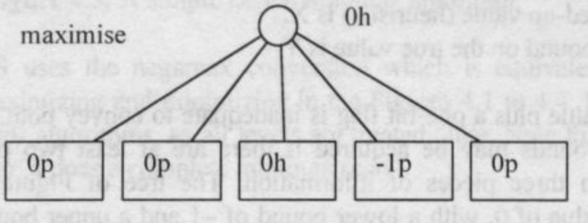


Figure 4.2: A 1-ply tree with mixed values.

In Figure 4.1 the root has a backed-up value of 0. It is perfect (indicated by p) because all the descendant values are perfect. In Figure 4.2 the root has a backed-up value of 0. However, because the middle descendant has only a heuristic value, and therefore might have a true value of anything, the backed-up value must be marked as merely heuristic (indicated by h).

There is however, more information available. The root in Figure 4.2 must have a true value greater or equal to 0, because of its perfect descendants. Now consider the tree fragment of Figure 4.3.

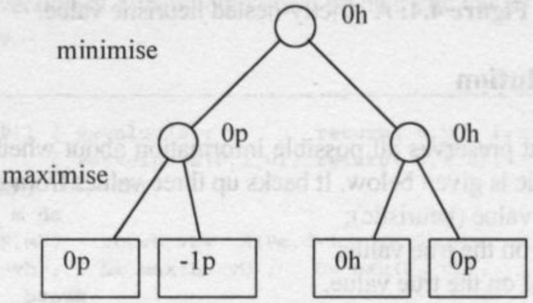


Figure 4.3: A 2-ply tree with mixed values.

In Figure 4.3, the right-hand node at the middle level receives a heuristic backed-up value. If all that gets passed up from this node is '0/heuristic', then the top node also gets marked as merely heuristic. However, if we use the additional ‘true value ≥0’ information about that right-hand middle node, then we can deduce that the top-level node has a true value of 0, not depending on the heuristic value at the bottom.

This illustrates that two separate pieces of information may be generated at a node:
1. the backed-up value (heuristic) is X;
2. a lower bound on the true value is Y.

A single value plus a one-bit flag is inadequate to convey both. Upper as well as lower bounds may be acquired if there are at least two plies of search, resulting in three pieces of information. The tree of Figure 4.4 returns a heuristic value of 0, with a lower bound of –1 and a upper bound of 1 on the true value:

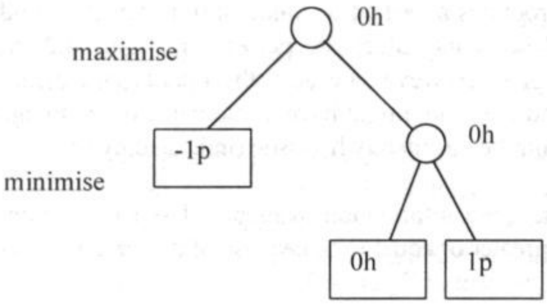


Figure 4.4: A strictly-nested heuristic value.

4.4 A Full Solution

An algorithm that preserves all possible information about whether values are perfect or heuristic is given below. It backs up three values from every node:
1. the backed-up value (heuristic);
2. a lower bound on the true value;
3. an upper bound on the true value.

There is no explicit flagging of a backed-up value being perfect: this is implicit in all three values coinciding. We exhibit the algorithm in pseudo-code (P is a position, and d is a depth limit in the search).

```
S(P,d)
{ if (perfect(P)) { v=value(P);  return( v,v,v );  }
```

```
  if (d=0)   { v=evaluate(P); return( v,-INF,INF );  }
  v= -INF;  L= -INF;  U= -INF;
  foreach move m do
  { Pm= make(P,m);    vh,vL,vU=  S(Pm,d-1);
     v= max(v,-vh);    L= max(L,-vU);    U= max(U,-vL);    }
  return( v,L,U );
}
```

Figure 4.5: A simple nested minimax algorithm.

The procedure S uses the negamax convention which is equivalent to the alternation of maximizing and minimizing in the Figures 4.1 to 4.4. Negamax is more elegant for algorithms, as all levels are treated alike. Note that S does not use alpha-beta: it does a complete minimax scan.

4.5 Incorporating Alpha-Beta

When alpha-beta is incorporated into the nested minimax search procedure there is a choice not in classic minimax. The alpha-beta technique can be applied to either kind of value: heuristic or perfect. If it is applied to heuristic values, it will make more cut-offs since heuristic bounds will usually be tighter than perfect bounds. Alternatively, if it is applied only to perfect values, it increases the chance of ending up with a perfect value for the root. It would usually be preferable to cut on heuristic values, although in searches where perfect values are very common, or much more useful than heuristic ones, cut-offs on perfect values only might be chosen.

A pseudo-code version of S using alpha-beta on heuristic values is given in Figure 4.6 below.

```
S(P,d,l,u) =
{ if (perfect(P)) { v=value(P);          return( v,v,v );  }
  if (d=0)        { v=evaluate(P,l,u); return( v,l,u);   }
  v= l;  L= -INF;  U= -INF;
  foreach move m do
  {  Pm= make(P,m);    vh,vL,vU=  S(Pm,d-1,-u,-v);
     v= max(v,-vh);    L= max(L,-vU);    U= max(U,-vL);
     if (v>=u)  break;
  }
  if (not all moves searched)  U= INF;
  return( v,L,U );
}
```

Figure 4.6: A nested minimax algorithm using alpha-beta.

4.6 Arriving at Nested Minimax

A search with mixed heuristic/perfect evaluations can be regarded as a perfect one which is incomplete and hence returns:

1. a pair of bounds defining a range of possible true values instead of a single value, plus
2. a heuristic value lying within that interval.

This is what suggests the term *nested minimax*: the outer shell of the minimax deals in perfect values and returns a range; the inner part deals in heuristic values and returns a point within that range.

Although the illustration above has used outer values that are perfect, the concept of nested minimax does not require that outer values be perfect. Nested minimax also applies to searches which do not encounter perfect values, but which can encounter evaluations known to be much more reliable. For example, nested minimax would also apply to a middle-game search that has a separate 'evaluation' to detect positions in which evaluations are much more reliable.

This Chapter has presented a search algorithm that retains more information from the evaluations it encounters than the original minimax algorithm in the literature. It is therefore a contribution to answering the problem statement of this thesis "can we increase our understanding of minimax search and use it to improve search algorithms?"

Chapter 5

AN INTEGRATED-BOUNDS-AND-VALUES (IBV) NUMERIC SCALE FOR MINIMAX SEARCH[1]

Search procedures generally discover bounds as partial results before a final value is obtained. In particular, game-playing programs typically store such bounds in a hash table, along with any exact results, for every position processed. This information saves computing time if the position is encountered again. For minimax search, upper bounds, lower bounds or exact values may occur. This chapter shows how upper bounds, lower bounds and exact values can conveniently be represented using a single numeric scale, which slightly simplifies existing program code, and avoids the necessity of a separate data item to distinguish bounds from exact values.

5.1 Bounds and Hash Tables

Game-playing programs typically store position scores in a hash table for use if the same position is reached again (e.g., via a transposition of moves). Due to the working of alpha-beta cut-offs, the position's 'score' will often be an upper bound, or lower bound, on the actual value, rather than an exact value. The stored values are often described and implemented (e.g., Thompson and Condon, 1983; Nelson, 1985; Marsland, 1986) as consisting of a position value (e.g., as a 16-bit integer) plus a separate flag (e.g., as a 2-bit field) that can take one of three significant values (exact, upper bound, lower bound) in another byte or word.

---

[1] This Chapter is an edited version of Beal (1995) An Integrated-Bounds-and-Values (IBV) Numeric Scale for Minimax Search. Thanks are due to the Editorial Board of the *ICCA Journal* for permitting its use here.

Mathematically, it is possible to map a set of numbers, together with a set of upper bound values, and a set of lower bound values, into a single number scale, called here an integrated-bounds-and-values (IBV) system. These single numbers can then be manipulated conveniently by ordinary operations such as negation, and comparison (e.g., less than, greater than, or equal), as well as by specialised operations to examine and set the embedded bound or exact status of the number.

This mapping provides a convenient notation for coding and processing backed-up values, which can be used both during the search and when storing or retrieving hash-table values. It can even provide slightly simpler and faster program code than handling flag values in a separate data item or byte. Any performance improvement may be insignificant in practice, but the IBV method has the advantage of neatness.

5.2 The IBV Representation

Exact numbers	(n)	are represented as $4n$.
Upper bounds	$(\leq n)$	are represented as $4n-1$.
Lower bounds	$(\geq n)$	are represented as $4n+1$.

This yields the encoded IBV scale, the central part of which is illustrated in Figure 5.1.

n	–1			0			1		
ibv	–5	–4	–3	–1	0	1	3	4	5
ibv meaning	≤ -1	$= -1$	≥ -1	≤ 0	$= 0$	≥ 0	≤ 1	$= 1$	≥ 1

Figure 5.1: Some sample values of n, with their possible IBV values and their meanings.

Using this encoded IBV scale, the following properties hold:

(a) negating a bound produces the corresponding bound from the opponent's point of view, thus allowing a 'negamax' version of minimax to transfer bounds in the same way as exact values;

(b) a lower bound at n (i.e., $\geq n$) is greater than an exact n;

(c) an exact value (i.e., n) is greater than an upper bound at n (i.e., $\leq n$).

These properties enable a minimax search procedure to treat encoded values in exactly the same way as exact values (i.e., scanning all moves, and selecting the highest value found). No explicit testing for lower/upper/exact status is required. This, in turn, means that when a minimax search is augmented by a hash table to store and retrieve values, the encoded values can be used everywhere. It is not necessary to add or remove the upper/exact/lower flag when storing and retrieving values. By needing no manipulation, they simplify the program logic.

5.3 A Program using IBV

A popular version of alpha-beta search has been fail-soft alpha-beta as described, for example, in Marsland (1986) and Kaindl (1990). The following C-language code (Figures 5.2 and 5.3) performs fail-soft alpha-beta assisted by hash-table storage, using the integrated bound-and-value number convention throughout.

```
ibvsearch(alpha, beta, d)  /* alpha and beta are IBV values  */
{
 if( hash_entry_found()  andand  stored_depth ≥ d )
 { v= stored_value;
   switch FLAG(v)
   { case EXACT: return(v);
     case LB: if(v ≥ beta) return(v); else alpha= max(alpha,v); break;
     case UB: if(v ≤ alpha) return(v); else beta= min(beta,v); break;
   }
 }
 if( d==0 )   best= evaluate();     /* evaluate() is EXACT  */
 else
 { best= -INFINITY;
   cut= FORCE_EXACT(beta);
   for(m=firstmove;  m;  m=nextmove)
   { make(m);
     v= - ibvsearch( -beta, -max(alpha,best), d-1);
     undo(m);
     if(v > best)  best= v;
     if(best >= cut) { if( !lastmove ) best= FORCE_LB(best); break; }
        /* when a cut-off occurs, the result is a lower bound */
   }
 }
 hash_store(best, d); // best may be LB, EXACT or UB, but if best is a
 return(best);        // LB it will be ≥beta, and if UB will be ≤alpha
}
```

Figure 5.2: IBV-based code to perform fail-soft alpha-beta.

The function 'evaluate' can be obtained from an existing evaluation function by multiplying its result by 4. Alternatively, one may adjust the evaluation function to use scores 4 times larger throughout.

The function FORCE_EXACT delivers $4n$ given any of $4n-1$, $4n$, or $4n+1$. FORCE_LB delivers $4n+1$ given any of $4n-1$, $4n$ or $4n+1$. FLAG delivers UB, EXACT, or LB given $4n-1$, $4n$ or $4n+1$, respectively. They can be efficiently implemented as shown in Figure 5.3.

```
#define FLAG(x)         (x and 3)
#define FORCE_EXACT(x)  ((x+1) and ~3)
#define FORCE_LB(x)     (((x+1) and ~3) + 1)
#define EXACT  0
#define UB     3
#define LB     1
```

Figure 5.3: Code to implement some essential functions on IBV-represented values.

The function definitions in Figure 5.3 assume binary two's-complement representation of integers. This is effectively universal throughout modern machines, but a few rare machine types might require some other implementation.

5.4 Confirming the Correctness of IBV

We note that Figure 5.2 contains some unfamiliar operations. After negating an encoded value (which may be a lower or upper bound rather than an exact value), then comparing it with another number which also may be a bound rather than a value, it is not immediately clear what the comparison is comparing, and what the possible outcomes are.

In order to be sure that the various operations in Figure 5.2 perform as claimed, the following analyses are given.

Let e be an encoded value,
v be a position value, and
f be $-1/0/1$, meaning upper/exact/lower.

Then we define $e = e(v, f)$, with $e(v, f) ::= 4v + f$ (but see footnote [2]).

We note that e can always be uniquely decomposed into v and f by division by 4 such that the remainder is -1, 0, or 1, a remainder of 2 being excluded. Henceforth, we denote $e(v, f)$ by (v, f) for brevity.

Let $V(e)$ denote the position value v embedded in e. Properties (1) to (3c) below can now be listed.

1) Negating the encoded number produces $-e = -4v - f = (-v, -f)$.

For
$$\begin{aligned} e &= (v, 0) &\Rightarrow\quad -e &= (-v, 0)\\ e &= (v, -1) &\Rightarrow\quad -e &= (-v, 1)\\ e &= (v, 1) &\Rightarrow\quad -e &= (-v, -1) \end{aligned}$$

Thus, negating an encoded value always produces an encoding of the negated position's value, with reversal of the type of bound if a bound, and retention of its exact status if exact.

2) The position values are dominant.

Therefore, comparisons of encoded values where the positions' values are not equal will behave in accordance with the positions' values.

3) Particular cases of comparisons.

(a) The comparison $(x \geq y)$, where x is LB:
if $V(x) > V(y)$, then $(x \geq y)$ is true because the Vs dominate:
if $V(x) = V(y)$, then $(x \geq y)$ is true because the LB flag is largest.

Thus $(x \geq y)$, where x is LB, is identical to $(V(x) \geq V(y))$.

(b) The comparison $(x \leq y)$, where x is UB:
similar observations as under (a) hold.

Thus, $(x \leq y)$, where x is UB, is identical to $(V(x) \leq V(y))$.

2 The use of 4 as a multiplier is convenient for decomposing the encoded numbers using bit-pattern masks, but any multiplier ≥ 3 will produce a numeric scheme having essentially the same properties.

(c) The comparison $(x \geq y)$, where y is EXACT:
This will behave as $(V(x) \geq V(y))$, except when $V(x) = V(y)$ and x is UB.

The following considers the correctness of the procedure of Figure 5.2

Properties (1) to (3c) above support the following claim:

The procedure of Figure 5.2 delivers the same result, and makes the same cut-offs, as a version using ordinary position values and separate bounds flags.

This may be confirmed by examining the behaviour of the IBV comparison statements, and the bound information embedded in the IBV-result value. The following is a proof outline.

It is assumed that $V(alpha) < V(beta)$ on the initial call to ibvsearch.

- From property 2 above, the statement: ***if***$(v > best)$ $best = v$ will store in *best* the largest position value encountered. This matches the behaviour of programs using ordinary position values.

- From (3c) above, and noting that $V(cut) = V(beta)$, the statement ***if***$(best \geq cut)$ ... will activate if $V(best) \geq V(beta)$, except in the case when *best* is a UB. However, *best* can only be a UB when $best \leq alpha$ (see below), which implies $V(best) < V(beta)$, which means activation is not desired. So the IBV version behaves in the same way as ***if***$(best \geq beta)$ ... in programs using ordinary position values.

  Verifying the activation of ***if***$(best \geq cut)$ ... also shows that recursive calls of ibvsearch preserve the $V(alpha) < V(beta)$ property.

- From properties (3a) and (3b) above, the statements ***if***$(v \geq beta)$ .. and ***if***$(v \leq alpha)$ ... will behave in the same way as comparisons using ordinary position values.

For correctness, we also require that the result returned indicates EXACT, LB or UB as appropriate. In particular, we require that UB is only indicated when the value is alpha or below, and LB only indicated when the value is beta or above. This can be verified with an inductive argument as follows.

At depth $d = 0$, we assume the routine *evaluate* is correct (base case). For greater depths, we assume (inductive hypothesis) that the recursive calls to obey the following rule (given as a comment against the final return in the code above): "the result may be EXACT, LB or UB, but if LB, result ≥ beta and if UB, result ≤ alpha".

Hence, *ibvsearch* will be ≤ –beta if UB. Hence, *–ibvsearch* ≥ beta if LB. Thus if an LB is assigned to best, best ≥ beta. This establishes the LB half of the inductive conclusion.

For *–ibvsearch* to be UB, *ibvsearch* must be LB, and hence ≥ –max(alpha,best). If max(alpha,best) = alpha, then *ibvsearch* ≥ –alpha, *–ibvsearch* ≤ alpha and the desired result holds. If max(alpha,best) = best, then, if *ibvsearch* is LB, then *ibvsearch* must be ≥ –best, –*ibvsearch* ≤ best. So, when UB, *–ibvsearch* will not replace *best*. Hence, no UB greater than alpha can ever be assigned to *best*. This establishes the UB half of the inductive conclusion.

This concludes the proof outline.

5.5 Advantages of IBV

This chapter described a method of using an integrated-bounds-and-values (IBV) number scale for minimax searches. Although analysis of the correctness of procedures of the operations on the encoded values is lengthy, the use of the IBV technique is very simple.

An integrated-bounds-and-values scale offers a convenient notation and simplified program logic. The advantages are: (1) the use of a single variable rather than the use of a number and a separate flag, and (2) identical handling of backed-up values while searching and while storing or retrieving from a hash table. It can be applied to any simple minimax search, but does not extend usefully to handling the multiple bounds encountered in nested minimax. The programming simplifications provided by IBV constitute a small contribution to answering the problem statement of this thesis.

At depth $d = 0$, we assume the routine *evaluate* is correct (base case). For greater depths, we assume (inductive hypothesis) that the recursive calls to *ibvsearch* obey the following rule (given as a comment against the final return in the code above): "the result may be EXACT, LB or UB; but if LB, result ≥ beta and if UB, result ≤ alpha".

Hence, *ibvsearch* will be ≤ −beta if UB. Hence, −*ibvsearch* ≥ beta if LB. Thus if an LB is assigned to best, best ≥ beta. This establishes the LB half of the inductive conclusion.

For −*ibvsearch* to be UB, *ibvsearch* must be LB, and hence ≥ −max(alpha,best). If max(alpha,best) = alpha, then *ibvsearch* ≥ −alpha, −*ibvsearch* ≤ alpha and the desired result holds. If max(alpha,best) = best, then, if *ibvsearch* is LB, then *ibvsearch* must be ≥ −best, −*ibvsearch* ≤ best. So, when UB, −*ibvsearch* will not replace *best*. Hence, no UB greater than alpha can ever be assigned to *best*. This establishes the UB half of the inductive conclusion.

This concludes the proof sketch.

5.5 Advantages of IBV

This chapter described a method of using an integrated-bounds-and-values (IBV) number scale for minimax searches. Although analysis of the correctness of procedures of the operations on the encoded values is lengthy, the use of the IBV technique is very simple.

An integrated-bounds-and-values scale offers a convenient notation and simplified program logic. The advantages are: (1) the use of a single variable rather than the use of a number and a separate flag, and (2) identical handling of backed-up values while searching and while storing or retrieving from a hash table. It can be applied to any simple minimax search, but does not extend easily to handling the multiple bounds encountered in nested minimax. The programming simplifications provided by IBV constitute a small contribution to answering the problem statement of this thesis.

Chapter 6

Selective Search Using Nested Minimax[1]

It is argued that an important requirement for doing an effective selective search is distinguishing different levels of reliability of evaluations arising in the search. The use of nested minimax for administering the information from multiple levels of reliability is illustrated with examples using three levels of reliability. Comparisons with other algorithms are given.

6.1 Specialised Knowledge in the Endgame

It is possible to improve the performance of a King-and-Pawns-versus-King-and-Pawns endgame program (for instance) by adding some specialised knowledge for evaluating positions with specific properties. The question being considered here is "how should that knowledge be integrated into the search process?". It is expected that positions in which the specialised knowledge is applicable will be much more likely to be evaluated correctly than other positions. This knowledge can be assumed to improve the performance, since the search will now have accurate values more often than before.

However, if the knowledge is simply merged into the existing evaluation function, there is some loss of potential. Consider the search tree of Figure 6.1.

[1] This Chapter is an edited version of Beal (1986) titled Selective Search without Tears. Thanks are due to the Editorial Board of the *ICCA Journal* for permitting its use here.

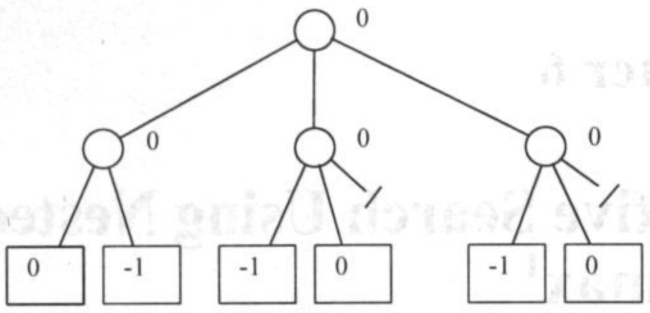


Figure 6.1: Example of a search tree. / indicates alpha-beta cut-off; backed-up values are on the right of the nodes.

It is clear that the minimax result is 0. But what kind of 0? If it happens to be based entirely on the specialised, and hence reliable, knowledge, there would probably be little advantage in performing a deeper search.

Also, the possibility of evaluating at the interior nodes should be considered. A conventional minimax search only evaluates at terminal nodes. The question is: would one of the middle nodes (or even the root node) have a reliable value if the evaluation function was applied there? If so, the search could be truncated at that node. Why perform even a 1-ply search when a 0-ply search will do?!

The remedy for these problems is clearly to distinguish between reliable knowledge and unreliable guesses, and treat them differently. For example, we can readily identify three types of reliability: (1) perfect values (e.g. checkmate, stalemate, or database values), (2) specialised endgame evaluations, (3) heuristic values. Different levels of reliability arise in the middle game too.

6.2 Reliability in the Middle Game

Consider the position of Diagram 6.1 that could arise in the late middle game.

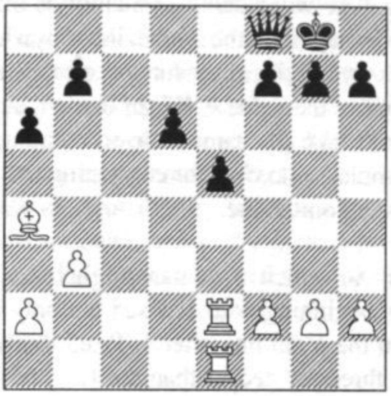


Diagram 6.1: White to play.

In Diagram 6.1, there are several elementary tactical lines which a search has to investigate. White has a sequence 1. Rxe5 dxe5 2. Rxe5 which generates a threat to win the Queen. It fails to 2. ... b5 of course. The crucial question is: how large should the search tree be at various nodes?

Consider the position after 1. Rxe5 has been refuted by 1. ... dxe5 (assuming dxe5 was examined first). Would a human consider alternative black moves to 1. ... dxe5 at this point?

My suggestion is that a human would ignore the alternatives. A human would have looked at the starting position and seen that the worst Black can do is trap the Bishop with … b5, resulting in a approximately level score (Q+P v. R+R). Hence when, after 1. Rxe5 dxe5, Black is doing better than that, the human will abandon 1. Rxe5.

However, from Black's point of view in a conventional alpha-beta search, the search bounds are still unbounded upwards. In human terms: the black player using conventional alpha-beta will be trying to show exactly how bad 1. Rxe5 is, before White tries the next move. The remedy here is not as clear as in endgame positions.

Assume there were a mechanism that could evaluate the position of Diagram 6.1 and produce the answer "worst score for White is 0". How reliable is that score of 0?

Consider the following. The worst-score evaluation is effectively producing a range as its result. It is saying: "the value lies anywhere between W and infinity". This contrasts with a direct evaluation or a quiescence search which both produce point values: "the value is X". In particular, if a quiescence value and a worst-value would take the same time to compute, and were of equal reliability, then there would be no case for computing the worst value. It would always be better to have a point value.

Now consider the way in which a human might reason in the position of Diagram 6.1. The lines beginning with 1. Rxe5 are only a few ply deep, since the move b5 appears in the main line after 1. Rxe5 dxe5 2. Rxe5 as 2. … b5. These lines are at least three ply deeper than the 1. … b5 analysis showing that White's Bishop can be trapped. The initial worst value is being used to cut off moves such as 1. ... h6, which need an analysis deeper than the original worst-case analysis.

My conclusion from the above is that worst-case values should be designed to be, and treated as being, of much higher reliability than direct evaluations and also of higher reliability than quiescence scores.

6.3 Reliability in the Endgame

Consider again the tree of Figure 6.1, but now with reliablility information supplied (see Figure 6.2).

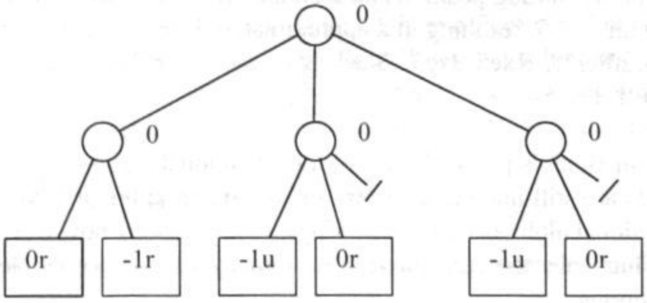


Figure 6.2: A tree with reliable (r) and unreliable (u) values.

Sufficient reliable evaluations have been encountered at the terminal nodes to determine a reliable value of 0 for the starting node.

Note that there are two issues here. The first issue is recognising nodes at which a reliable value has been obtained, so that search below them can be avoided (at least while there remain uncertainties elsewhere due to unreliable values). The second issue is making all possible deductions about the possible range of reliable values for the root node (or any backed-up node). Both kinds of information need to be correctly handled by the search. In the tree of Figure 6.2, it is not immediately obvious that the top value is a reliable 0, since there were three unreliable values examined at the bottom level. In Section 6.5 we treat the relation with nested minimax and perfect values.

6.4 Other Algorithms

This view of worst-case values as being of higher reliability than quiescence scores is implicit in existing algorithms. Berliner's B* algorithm treats optimistic and pessimistic scores (assumed to be computable at *all* interior nodes of a search) as bounds. This implies that they are of at least equal weight to the results of deep minimax searches (if the bound is being generated high in the tree). Slate (1984) describes some endgame evaluations in NUCHESS that generate bounds at interior nodes, and suggests that middle-game evaluations could be handled in the same manner.

6.5 How Nested Minimax Handles the Values

Regardless of how the reliable values come into existence, we now consider how to handle them in a minimax search. When dealing with perfect values into a heuristic search, Chapter 4 showed that there is a need to back up three values instead of a single value at every node, in order to be sure of obtaining, at the root, the maximum information regarding its perfect value.

If the better values are more reliable, but still not perfect, the same requirement holds: three values must be backed up. If perfect values can occur as well, it would be necessary to back up five values at every node! Each additional level of reliability introduces a requirement for backing up a range (i.e., two endpoint values) within which the current position has been found to lie. Henceforth we use the term *fat value* to denote a range.

A position value in a generalized nested-minimax algorithm is a package which has fat values (i.e., ranges) for the reliable layers. It typically has point values for the least reliable evaluations (i.e., the conventional heuristic evaluation function). For instance, (–999:999 0:999 1.2) could express the information that nothing is known with absolute reliability (assuming 999 means infinity), but that a reliable lower bound is 0, and the heuristic value is 1.2.

Fat values can be used to express the idea of bounds. For instance, a fat value of 4:999 is equivalent to saying the value is >=4. However, fat values are more general than a single bound. They are equivalent to two bounds.

(N.B. The endpoints of fat values are *not* akin to alpha-beta bounds, since the alpha-beta bounds express the idea of limitation of interest in what the value is, and are not bounds on the value itself.)

The pseudo-code algorithm in Figure 6.3 is a minimax skeleton that applies to any search where:

- three levels of evaluation reliability exist,
- interior evaluations can become available at any node (but not necessarily).

```
S2(P,d,L,U)
{ if (d=0)  return( EV(P,L,U) );
  Bv ← IV(P,L,U);
  foreach move m do
    { bv ← innerpoint(Bv);    if (bv>=U)  breakloop;
      Pm ← make(P,m);
      Vm ← minus( S2(Pm,d-1,-U,-bv) );
      Bv ← backup(Bv,Vm);
    }
  if (not all moves searched)  upperpoints(Bv) ← INF;
  return(Bv);
}
```

where:

```
  minus(L:U l:u v) = (-U:-L -l:-u -v);

  innerpoint(L:U l:u v) = v;

  backup(L1:U1 l1:u1 v1,  L2:U2 l2:u2 v2) =
  (max(L1,L2):max(U1,U2) max(l1,l2):max(u1,u2) max(v1,v2));
```

Figure 6.3: A three-layer nested-minimax algorithm.

The evaluation function EV is an evaluator for terminal nodes. It is expected to return something like (–999:999 –999:999 v), where v is the least reliable heuristic value. IV is an interior-node evaluator that returns (sometimes) non-infinite endpoints in its result package.

The skeleton above can be fleshed out to include iterative deepening, and the other usual refinements found in conventional chess programs (Heinz, 1997).

6.6 Observations on Nested Minimax

The most reliable layer could be allocated to game-theoretic values. In Figure 6.3 IV would test for game-theoretic values (e.g., checkmate) and if found return (999:999 999:999 999).

Slate's (1984) algorithm would fit into the scheme by arranging that the function IV tested for the applicability of a specialised evaluation, and if available returned (–999:999 v:999 v).

A depth-first version of Berliner's (1979) B* algorithm could also use this skeleton, although a slightly different routine would be needed for the top level.

In favour of this complicated backing-up scheme is the ability to represent different kinds of draw values. The value package (0:0, 0:0, 0) indicates a game-theoretic draw. The value package (–999:999 0:0 0) indicates that specialised and reliable knowledge has evaluated the position a draw. The value package (–999:999 –999:999 0) indicates that the position is level according to a heuristic evaluation. Simpler schemes may not be able to distinguish these different cases.

Finally, we remark that this Chapter has discussed two ideas relevant to selective searching, and therefore to the research questions of this thesis:
1. The importance of a deliberate design of evaluations more reliable than a quiescence search.
2. The use of nested minimax to administer the values that result.

Chapter 7

Perfect Values in Quiescence Search[1]

This chapter deals with perfect values in quiescence search. In chess perfect values sometimes occur in the quiescence search. Quiescence searches often deliberately include checks, and all moves escaping from check, in order to detect at least some checkmating sequences. Such detection is at lower cost than a full width search. When dealing with quiescence searches that deliberately include checks, and moves out of check, the checkmate positions that result are, of course, perfect values. The search therefore deals with values of two distinct reliability levels: heuristic values from the evaluation function, and perfect values from checkmates encountered. This means that nested minimax is an appropriate algorithm to use.

To illustrate the benefits of handling perfect values in a quiescence search, we discuss one particular rule for selecting allowable checks within a quiescence search.

Hyatt, Gower and Nelson (1984) gave an example of CRAY BLITZ performance on a position in which there is a mate in 10 moves (i.e., 19 ply) for White. At that time, using two Cray-1 processors in parallel, the program was able to examine nearly 40,000 positions per second and in less than 3 minutes (5.6 million nodes) found the mate in 10.

The same position was given to the contemporary program BCP running on a Z8000 microcomputer. It did not outperform CRAY BLITZ. However, the solution time was less than a comparison of processor speeds would lead one to

1 This Chapter is an edited version of Beal (1984) titled Mating Sequences in the Quiescence Search. Thanks are due to the Editorial Board of the *ICCA Journal* for permitting its use here.

expect. BCP found the mate in just over 5 minutes (400,000 nodes). The most important variable affecting BCP's time is a rule governing which moves are considered in the quiescence search. It is not obvious though, either from this one example, or from wider experiments, what quiescence rules are best overall. This chapter discusses one rule in particular.

7.1 Typical Quiescence Search

Since CHESS 4.5 (Slate and Atkin, 1977), most chess programs perform an exhaustive search to some fixed depth followed by the application of a variety of search extensions, after which, the bottom nodes are processed by a quiescence search. The quiescence search explores all disruptive moves from its current position, terminating at quiescent positions, defined as those in which no disruptive moves are available.

Disruptive moves are usually defined to be captures, promotions, checks, etc. Some programs omit certain moves to reduce the size of the search without suffering much risk of missing a crucial move. Captures are self-limiting but checking sequences can go on indefinitely. Allowing checks in the quiescence is generally more expensive in search time than captures. All moves getting out of check are defined to be disruptive.

The rules for checks in CRAY BLITZ are:

1. Up to two non-capturing checks can be included in any variation of the quiescence tree.
2. To be included, non-capturing checks must be part of a consecutive sequence of checks in the quiescence variation, and the main search variation above the quiescence search must include checks.
3. Checks which are captures are included without limit.

The idea is to find mating attacks, and material gain brought about by king attacks, which go deeper than the main depth, yet to limit the additional search cost to an acceptable proportion of the time spent on more mundane evaluations.

It should be noted that CRAY BLITZ has two other major rules affecting the content of the quiescence tree:

1. many captures are eliminated by doing a simple exchange evaluation first, and

2. pawn advances to the seventh rank are included and are treated like checks.

7.2 An Augmented Quiescence Search

The rule singled out for discussing in this Chapter concerns checks: checks are included if and only if they have exactly one legal reply. The reply could be moving the King, capturing the attacker, or blockading. There is (almost) no depth or count limit. This rule increases the proportion of perfect values encountered in the search. The original intention of this rule was to detect some (relatively) frequent cases of mate in 2. An example of such a mate in 2 is a back-rank mate where the victim has a piece that can interpose, but not on a defended square. It was then observed that there were cases of deeper 'one-reply' mates in test positions, so the depth limit was removed. The search time only increased marginally.

Of course, this 'only one reply' rule will miss many mates and combinations. It is a cost-effectiveness issue. More complex positions will be resolved by the main search an iteration or two later. The idea is to pick up some (relatively) frequent cases at low cost within the quiescence search.

The cost of including additional moves in a quiescence search is twofold:
1. the additional time spent in the move generator (a linear cost);
2. the tree expansion caused by increasing the average branching factor (an exponential cost).

Checks are more expensive than captures to detect. Counting the number of legal replies to a check increases the expense further. However, one-reply checks only make a tiny increase in the average number of branches, which are mainly captures. Since exponential costs are usually dominant, the one-reply rule is low-cost, compared to rules that have significant exponential cost. The other rules used in the test program are: all captures are included; pawn promotions are included; but not advances to the seventh rank.

7.3 A Deep Position with Perfect Values

In Diagram 7.1 we show the position as given by Hyatt, Gower and Nelson (1994).

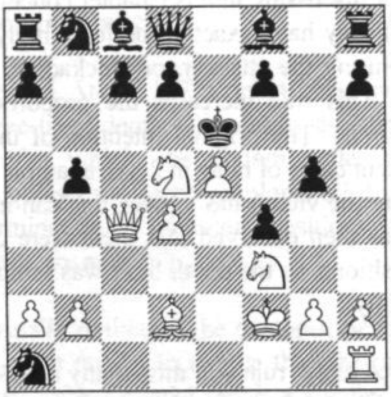


Diagram 7.1: Mate in 10.

They give as the main line: 1. Nxg5+ Qxg5 2. Nxf4+ Ke7 3. Nd5+ Ke6 4. Nxc7+ Ke7 5. Nd5+ Ke8 6. Qxc8+ Qd8 7. Nc7+ Ke7 8. Bb4+ d6 9. Bxd6+ Qxd6 10. Qe8++. BCP has a parameter, usually set to 'infinity', which determines the maximum length of one-reply check sequences allowed in the quiescence search. Table 7.1 shows the effect of progressively adjusting this parameter.

The times go up as the 'one-reply' length is reduced because the main search has to reach a deeper iteration before the mate in 10 is seen. The program used for this experiment, BCP, has a selective, rather than exhaustive, main search which is why the expansion factor for an extra ply of main search is somewhat less than the usual 5 or 6.

BCP's execution time		
quiescence rule	time	nodes
4 or more checks	5 minutes	382,664
3 checks	20 minutes	1,528,152
2 checks	65 minutes	2,442,758
1 check only	not tried	

Table 7.1: The effect of the one-reply extensions.

The main line ends in a 'one-reply' line of 5 checks. However, Black does not have to enter it. He can choose other mates in 10 that have only 4 one-reply checks at the end. This is why lengths of 5 or more one-reply checks do not shorten BCP's solving time.

It should also be noted, when comparing with CRAY BLITZ, that although BCP solves the mate problem with fewer nodes, BCP was deliberately engineered to be efficient at tactical problems. Positional considerations were relegated to second place.

7.4 Effect of the One-Reply Heuristic

Of course, it is inevitable that any quiescence rule will have some positions on which it is advantageous. The important question is: how often are mates in 2, 3, 4, … 'one-reply' sequences? In order to get some idea of the answer to this question, the first ten problems achieving mate in *Win at Chess* by Reinfeld (1958) were examined. Figure 7.2 shows the time required for BCP solutions when the 'one-reply' sequences of various length are perceived at the quiescence depth. This shows that 'one-reply' sequences are at the tail end of all the mating sequences (therefore justify the rule) in 5 out of the 10 positions. Since the reductions in search time are massive when they occur, and the increase is low in the others, this suggests that the rule may be cost effective.

	Time (in seconds) for BCP solution				
Position	L = 1	L = 2	L = 3	L = 4	L =- 5
4 (a mate in 2)	2.60	2.60	-	-	-
5 (a mate in 2)	0.22	0.06	-	-	-
9 (a mate in 5)	16.00	13.00	2.00	2.00	2.00
12 (a mate in 2)	0.40	0.40	-	-	-
14 (a mate in 4)	13.00	14.00	14.00	14.00	-
27 (a mate in 2)	0.60	0.20	-	-	-
35 (a mate in 4)	7.00	8.00	8.00	8.00	-
50 (a mate in 3)	2.80	0.60	0.60	-	-
54 (a mate in 2)	0.60	0.10	-	-	-
55 (a mate in 4)	10.00	10.00	10.00	10.00	-

Figure 7.2: Benefits of one-reply extensions on ten test positions. (L is the length of one-reply sequences allowed in the quiescence.)

The experiment and discussion show that the one-reply rule is worth considering as contribution to search efficiency in chess programs. The occurrences of perfect values make the use of nested minimax for operating the search desirable.

This Chapter has an indirect connection with the research questions of this thesis. It is a case study of an important component of all high-performance chess programs. Emerging from this case study is an example of the need to handle perfect values intermixed with heuristic values, which can be administered effectively by the nested minimax algorithm introduced in Chapter 4.

Chapter 8

Minimax and Retrograde Minimax using Patterns[1]

Retrograde minimax embodies the idea that the study of heuristic methods that often give the wrong answer, is replaced by studying the properties of algorithms that always return the perfect value.

A good analogy to bear in mind is an example used by Dijkstra (1972) in the book *Structured Programming*. A table of prime numbers is not just a table of arbitrary numbers. There are simple algorithms for computing it, which are undoubtedly correct because they embody a clear mathematical definition. There are also heuristic programs that attempt to calculate primes, leaving unknown which of the numbers output actually are prime. Where exact algorithms are computationally feasible they are naturally preferred. Even for heuristic programs some exact guarantee is valuable (e.g., this number may not be prime but it has no factors of less than eight digits), which may be regarded as finding an exact algorithm for a clearly defined modification of the problem. Similarly, we would prefer exact programs for chess functions.

8.1 KPK: Tables and Concepts

For many chess endgames a clear exact algorithm (systematic valuing of every position) is computationally feasible (Van den Herik and Herschberg 1985; Thompson 1986,1997; Stiller 1992; Schaeffer, 1997). This Chapter is concerned with ways of constructing more compact algorithms without relaxing the requirement of proving them correct.

---

[1] This Chapter is an edited version of Beal (1980). The Construction of Economical and Correct Algorithms for King and Pawn against King, in *Advances in Computer Chess 2*. Thanks are due to Edinburgh University Press for permitting its use here.

King and Pawn against King (KPK) is the simplest ending requiring non-trivial concepts for its solution: only the best human players are perfect evaluators over the whole space, which, reflections and redundancies removed, consists of 98304 distinct configurations. The analogue of the table of prime numbers is the database, Clarke (1977), that labels each of these configurations draw or win in a specified number of moves, obtained by retrograde minimaxing (backing-up) from primitive terminal positions. Such a table enables a simple program to be written to play perfectly, whose time complexity is small but whose space complexity is correspondingly large (the simple forward minimax method of evaluation is at the other end of the time/store trade-off axis). Similar databases have now been constructed for a number of more complicated endings (see references given above).

These tables are, of course, very large and the method soon becomes as space bound as simple minimaxing is time bound.

Conventional programs written for the KPK ending in a high-level language include those of Tan (1972) and Harris (1977), which, using little or no stored data, are more or less direct attempts to encode knowledge obtained from books or personal experience. Bramer (1977) and Beal (1977) constructed programs consisting of simple search routines accessing a position evaluator, which is based on a hierarchy of functions and predicates developed and refined by interaction with expert players or stored databases. Michalski and Negri (1977) described a program that starts with a repertoire of useful-looking predicates and a sample of positions whose value is known and builds up descriptions of winning positions in terms of these. It is also possible to combine pattern-knowledge with search. Van den Herik (1980) describes a program that does this for KNPK.

The Tan and Harris KPK programs are strong but not completely correct; those of Bramer and Beal are known to be correct but only as a result of comparison with the database over the entire set of legal positions. Programs stated to be nearly correct may nevertheless contain 'pockets' of completely unsound play. Only the simple search program that directly accesses the database can be said in any sense to be 'correct by construction', and this of course is at the extremely bulky end of the time/store trade-off axis.

8.2 Minimax Lookahead Tables

In Clarke (1977) a simple diagram was used to illustrate the relationship between search-based and data-based programs. Figure 8.1 is an elaboration of that diagram with an extra dimension for program correctness. The slope from the origin up on to the plateau of fully correct play represents the trade-off as it is currently known between economy and correctness for conventionally written computer programs; programs such as Tan's (1972) lie somewhere on this slope. The metaphor seems appropriate because standing on such a convex hill it is hard to know how much further you have to go to get to the top. Will such programs grow indefinitely to database dimensions as they are finally made perfectly correct?

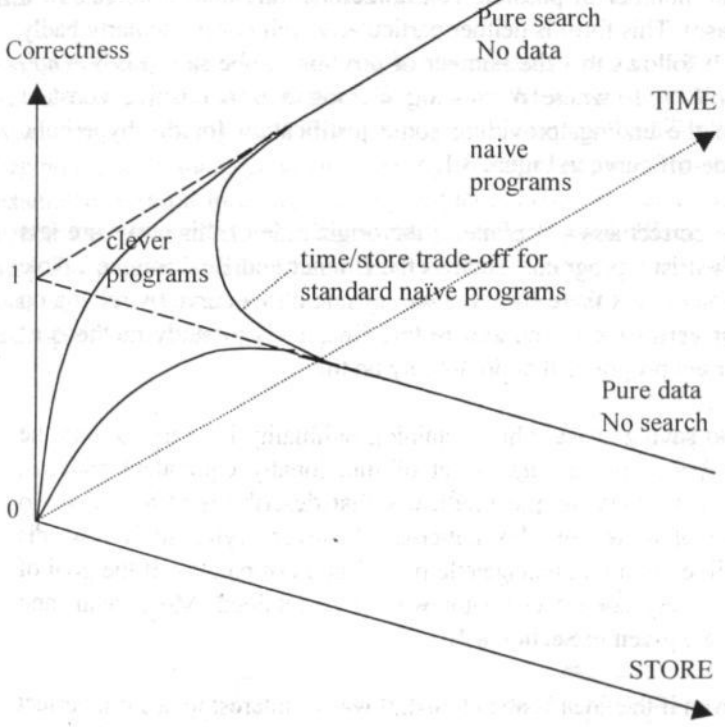


Figure 8.1: Trade-off curves for time, space and correctness.

A rough estimate of where the top of the hill is can be obtained by constructing a standard 'naïve' program. Assume for simplicity all positions to be either won for one side or drawn, so that the game is only two-valued, as in KPK or

KBNK. Further assume that we require the program to respond correctly to a request for a value within time T. Let the average branching factor with randomly ordered alpha-beta search be b (= real branching factor raised to the power 0.75), and let the processing time per node be t.

Then the maximum search depth i is given by $T = tb^i$ approximately, and the values of positions lying more deeply than this will have to be based on stored material. We suppose these to be stored explicitly: another search to depth i would involve storing the values of many terminal nodes.

To make further progress we need an assumption about the shape of the tree. Assume the number of positions at depth j to be of the form na^j, where $a < 1$ and n is the total number of possible configurations (and hence the size of a complete database). This form is neither particularly well nor particularly badly fitted by KPK. It follows that the number of positions to be stored is $S = na^i$, and so $(T/t)(S/n)^p = 1$, where $p = - \log b \,/\, \log a$ is a positive constant characteristic of the ending, providing some justification for the hyperbolic shape of the trade-off curve in Figure 8.1.

Programs in the correctness = 1 plane to the origin side of this curve are less naive than the heuristic programs, such as the Bramer and Beal routines. Note that these have had to get there from the wrong side of the curve by relying on the database for verification. The aim in this Chapter is to study methods of constructing correct programs that do not rely on this.

One approach to saving space whilst retaining optimality is to try to map the complete state space onto a smaller set of functionally equivalent patterns, backing up from the very simple predicates that describe terminal nodes to force evaluation of more complex patterns. However, trying to follow this through in detail led to an unmanageable proliferation of patterns if the goal of generating a provably correct evaluator was to be retained. More detail and some examples are given in Section 8.10.

Nevertheless, even if this idea is abandoned, it was of interest to see if a perfect evaluator could be constructed by relaxing the requirement for deduction as a method of verification and instead using the database. At the time of this work I constructed a Fortran function of about 120 statements embodying 48 logical tests of geometric features, which discriminates perfectly between wins and draws over the whole space. The method was an iterative process of comparing the values returned by the function against the database for every position, noting features that they had in common and debugging the code in an essentially *ad hoc* though methodical way. Further details are to be found in Section 8.11, and the complete text of this program is given in Appendix A. For more complex endgames the database may be too large for this method to be feasible.

Since quite complicated geometric predicates seem to be necessary for optimal evaluation, and since these do not agree very well with the way people describe the winning process (in terms of the Dijkstra example my function for KPK is equivalent to an obscure formula that just happens to generate the first 5000 primes), I tried a more algorithmic and less data-driven approach.

8.3 KPK: Patterns and Sequences

My algorithmic approach started with the observation that if the side with the Pawn (assume it to be White) is to win, White must have a method of repeatedly advancing the Pawn safely until it reaches the 8th rank. Instead of beginning with static geometric descriptions the set of move sequences was examined. Central to many winning sequences were repeating occurrences of patterns (expressed as coordinates relative to the Pawn). Black's replies could vary the sequences, but White could always force the sequence to go through one or other of two particular patterns with each advance of the Pawn. An example is given in Figure 8.2, which shows a sequence of length seven.

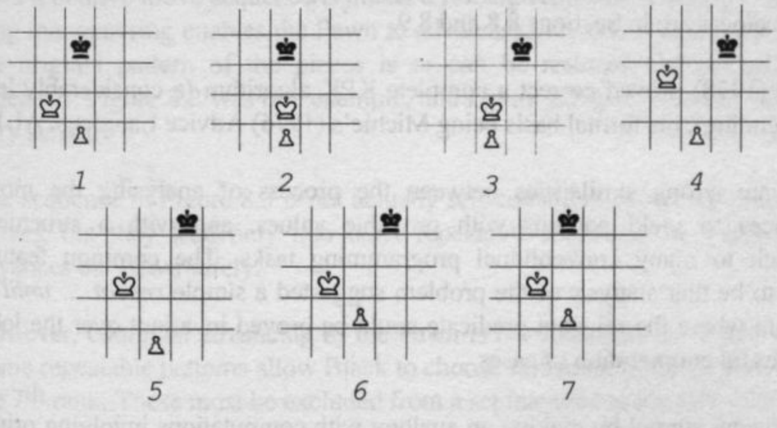


Figure 8.2: A forcing sequence.

The first pattern is an instance of one of the two key patterns. Black has alternatives that can be shown to lead only into other sequences in which White can continue to advance the Pawn safely. The proof that this is so uses a combination of induction and enumeration, in the sense in which Dijkstra uses the words when demonstrating that programs do what they are supposed to. Details are given in Section 8.8.

Not many positions match the two key patterns, of course, and for positions that do not, analysis is required to determine whether an *initialising* sequence (one forcing a key pattern) is available to White. In many cases one King is obviously nearer to critical squares, and in Section 8.9 we present some definitions that make this idea precise. From them are derived, and proved correct, extensions to the key patterns. The extended patterns match positions in which White can force the key pattern by an initialising sequence. The simpler ideas of White running the Pawn and Black capturing it are also covered by patterns proved correct in the same way.

The algorithm values positions by comparing them with its set of patterns: if the position matches one of them the value is known, if not a search is necessary. Ideally, this search should be small for all positions. For a complete algorithm (see next paragraph) it would be possible to determine precise upper bounds on the size of the search. Not included in the algorithm are key patterns for Black to draw (in particular those where the Pawn is on the Rook's file). The algorithm described so far is given precisely in Sections 8.4 to 8.6; proofs of correctness are in Sections 8.8 and 8.9.

Bratko (1978) proved correct a complete KPK algorithm (a considerably less subtle ending), his formal basis being Michie's (1976) Advice Language AL1.

There are strong similarities between the process of analysing the move sequences to yield patterns with provable values, and with a structured approach to many conventional programming tasks. The common feature seems to be that analysis of the problem suggested a simple *repeat ... until ...* structure where the relevant predicate could be proved invariant over the loop by a careful enumeration of cases.

The Chapter started by making an analogy with computations involving prime numbers. That analogy can be extended by pointing to other papers, e.g., Gries and Misra (1978), in which substantial improvements to an old algorithm (the sieve of Eratosthenes) have resulted not from new programming tricks, but from new lemmas about primes suggested by the fresh ways of looking at old problems that programming often requires. Good chess programs should ideally stimulate similar insights, by being well-structured embodiments of new theorems about the geometry of the game and more generally about the effects of certain kinds of tree searches.

8.4 Initialising and Repetitive Move Sequences

The algorithm decides, for a given KPK position with White to play, whether it is a win or draw. Many other KPK algorithms can be found: for example Beal (1977), Bramer (1977), Clarke (1977), Harris (1977), Piasetski (1977), and Tan (1972). Two of these (Beal's and Bramer's) were found to be perfectly correct by exhaustive comparison with Clarke's complete catalogue of KPK positions. This algorithm differs in attempting to be provably correct from its structure, without using a complete catalogue either during its construction or for testing its outputs afterwards. Moreover, the aim was to lay the foundations of automatic construction of the algorithm.

Central to the organisation of the algorithm is the concept of representing most of the wins in terms of repetitive move sequences and initialising move sequences. The repetitive move sequences have to be known before the initialisation sequences can be fully constructed.

By a repetitive move sequence is meant a forcing sequence where a little bit of king manoeuvring enables the Pawn to advance safely one square, after which the original pattern of the pieces is or can be restored and the sequence repeated. Figure 8.2 was one example, and Figure 8.3 gives another, which is very similar.

The sequence in Figure 8.3 is not actually self-contained. Black has alternative moves, but they lead only into other repeatable sequences in which White advances the Pawn safely.

However, continual advancing of the Pawn is not sufficient for White to win. Some repeatable patterns allow Black to choose stalemate when the Pawn is on the 7th rank. These must be excluded from a set intended to identify wins.

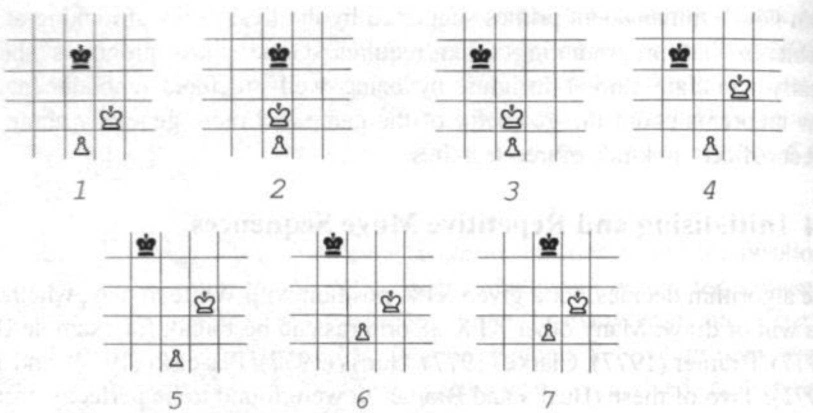


Figure 8.3: A repeatable sequence.

By an initialising sequence we mean a sequence of moves by the Kings only, that does not repeat, but which leads into a winning repetitive sequence. An example is given in Figure 8.4.

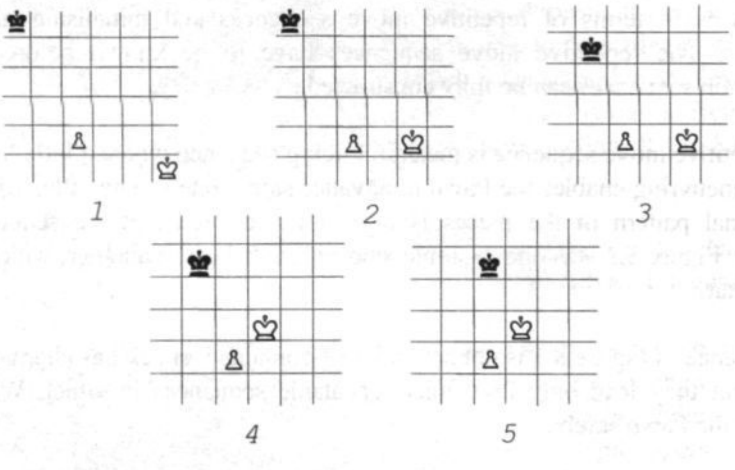


Figure 8.4: An initialising sequence.

After the initialising sequence of Figure 8.4 White can begin the repetitive sequence of Figure 8.3.

The algorithm is constructed from three parts: (i) a set of fundamental patterns with known values; (ii) a set of king-distance conditions extending some of the basic patterns; (iii) the exhaustive-search procedure. The patterns recognise simple cases and the starting points of winning repetitive sequences; the king-distance conditions recognise some positions requiring an initialising sequence; and the exhaustive search finds values for positions not matched directly. All patterns assume White to play. At all Black-to-play positions, therefore, one ply of exhaustive search is required.

Two remarks are needed: (a) 'pattern' is used here to mean some specified relationship between the pieces, not necessarily easily expressible by means of a diagram; (b) the Pawn is always assumed to be white, playing up the board, and on the Queen's side. (The Pawn can never change files, so Q-side positions can be analysed independently of K-side positions. K-side positions are equivalent to their mirror images on the Q-side. Positions with a black pawn are equivalent to positions with colours and player's sides reversed.)

8.5 The Fundamental Patterns with Known Values

In Figure 8.5 we provide the fundamental patterns upon which the algorithm is based.

The pawn-can-run pattern is a version of the simple "black King outside the square of the Pawn" rule given in elementary chess books.

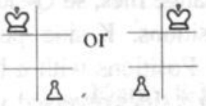
Bramer (1977) published an algorithm that recognises every pawn-can-run position and which was verified to be correct, but it is quite complex and the intention here was to keep the patterns and program structure as simple as possible. Most pawn-can-run positions not recognised by the simple rule can also be won, although requiring more moves, by king manoeuvring leading to a winning pattern that is included.

The two repeatable patterns are the basis for recognising the bulk of the remaining wins. The two non-repeatable patterns recognise a few extra positions where White's win does not pass through the other patterns.

The stalemate and pawn-captured patterns are draws by definition. A proof that the other patterns are wins appears in Section 8.8. Essentially, the method is simply to examine the move tree from a given pattern establishing that White can reach some already-known winning pattern no matter what moves Black plays. However, unlike examining a move tree from an actual position, consideration must be given to whether each move from a particular pattern will be possible and legal in every position that corresponds to the pattern, and the analysis must split into separate cases if not.

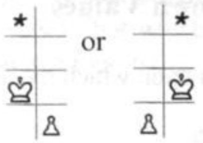
(White-to-play positions only)

1) Stalemate.
2) Pawn captured.
3) Pawn-can-run. A position matches this pattern if:
   ♚rank < ♙rank *or* | ♚file − ♙file | > (8 − ♙rank).
4) Repeatable pattern 1:



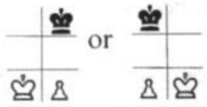
♙ not on rook file;
♚ anywhere

5) Repeatable pattern 2:



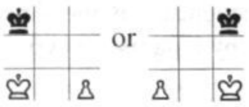
♙ not on rook file;
♚ anywhere, except★
if ♙rank<5

6) Non-repeatable pattern 1:



♙ not on rook file;
♙rank = 6

7) Non-repeatable pattern 2:



♙ not on rook file;
♙rank = 5

Figure 8.5: Basic patterns.

8.6 The King-Distance Conditions and the Exhaustive Search

The playing algorithm searches the move tree to evaluate a position, with positions matching a known pattern being terminal. In general, the more comprehensive the set of known patterns, the smaller such a search will be: yet it is desirable to keep the patterns few and simple to facilitate proof of correctness and obtain economy of representation.

A good solution is to extend three of the fundamental patterns by king-distance conditions (given in detail in section 8.9). These recognise many positions where one King is too far away from important squares to stop the other one establishing a known pattern by reaching a key square. For example, the position at the start of the initialising sequence of Figure 8.4 matches fundamental pattern 2 via distance condition 2 and is therefore directly recognised as a win.

The first two king-distance conditions are sufficient for White to be sure of reaching repeatable patterns 1 and 2 respectively, and therefore identify wins.

The third condition is derived from the pawn-captured pattern and identifies draws.

The king-distance conditions are given in detail, and proved correct, in Section 8.9.

8.7 Algorithm Performance

The playing algorithm is compact to code and the pattern matching can be executed very quickly. The overall speed of execution is proportional to the average number of lookahead positions in the exhaustive search. Whilst little or no search is needed for large classes of positions ranging from trivial to difficult, excessive search is needed for some others. This is due to three omissions from the algorithm as it stands: (1) the Pawn was assumed not to be on the Rook's file; (2) the double-pawn advance was ignored; (3) too few patterns recognise draws, leaving most drawn positions to be evaluated by search.

The first two omissions were deliberate, to simplify matters. The third omission results from an early aim of detecting all wins by patterns, intending the algorithm to be entirely static with no search. Draws could then have been identified indirectly by not matching any winning pattern. This aim was not reached with the patterns created, therefore search is required. In the absence of sufficient patterns to achieve an entirely static algorithm, performance would be improved if repeatable draw patterns were included in addition to winning patterns.

An additional winning pattern for the Rook's file cases, one or two repeatable drawing patterns, and associated king-distance conditions are required to rectify these omissions. In contrast, the two non-repeatable winning patterns could perhaps be discarded, as they recognise only eleven positions and these require little search to detect the win. In summary, it is possible to create an efficient KPK algorithm that uses only simple, easily defined patterns, and which can be proved correct from its structure.

The algorithm was not simple to devise. Typical chess books give incomplete information and the missing information is not only detail, but also concepts that each human has to learn (by means not understood so far) from the examples. Worse, humans who have learnt an adequate set of concepts from examples seem to find it very difficult, if not impossible, to define those concepts afterwards.

The largest effort is in verifying pattern correctness by analysis of the move tree in pattern space, and this is apparently well suited to automation. A major difficulty though, as noted in Section 8.8, is detecting when the analysis must be split into separate cases because a critical move is legal in some positions corresponding to the pattern but not in others. Moreover, splitting the analysis means dividing the original pattern into two separate patterns and there may be a choice of ways to do it. This choice may involve similar considerations to the original choice of what kind of pattern to use. Thus, move analysis in pattern space may not be straightforward. However, the attempt to program it should illuminate the goal of automatically generating suitable patterns.

8.8 Analysis of the Fundamental Patterns

The analysis consists of examining a move tree derived from a pattern: nodes in the tree represent patterns rather than individual positions. Patterns with already known values are terminal nodes in the tree.

In the case of repeatable patterns, if a result is established for the pattern with the Pawn on a particular rank, then, in the analysis of the pattern with the Pawn on a lesser rank, a recurrence of the pattern with the Pawn further advanced may be assigned the same value. In the case of two repeatable patterns, each of which leads into the other, the rank-7 result can be achieved, provided that a pawn advance is included in every cycle.

Two patterns are initially assumed to be wins, and therefore terminal nodes of a tree in pattern space, proving other patterns to be wins:

1. ♙ on rank 7; ♚ not on or adjacent to promotion square; White to play. (In two positions, White needs to promote to Rook to avoid stalemate)
2. ♙ on rank 7; ♔ adjacent to promotion square; White to play.

Analysing patterns instead of individual positions introduces additional complexities. In particular, the analysis must be split into separate cases whenever it would otherwise contain a node at which the value is not the same for all positions that correspond to the pattern. This problem occurs when a critical move (or moves) is (are) legal in some positions corresponding to the pattern but not in others.

The reasoning necessary to detect these cases and split the pattern appropriately is only given for the analysis of repeatable pattern 1. It would be the most difficult part of automating this kind of analysis. Also, the reasoning necessary to deduce what pattern exists after moves are made is only illustrated in the analysis of the pawn-can-run pattern. However, in the case of 'diagram-type' patterns, this reasoning is fairly straightforward.

8.8.1 The Pawn-Can-Run Pattern

The pawn-can-run pattern (pattern 3 in Figure 8.5) is:

♚rank < ♙rank *or* | ♚file - ♙file | > (8 – ♙rank)

This pattern is analysed by splitting into two cases, each of which is further divided.

Case (*a*): White King not in front of the Pawn on the same file

This case is analysed, rather than just assumed to be an elementary win, as it demonstrates that the verification can be treated as a search in the pattern tree.

Case (*a. 1*): Pawn on rank 7
= terminal pattern 1 : win

Case (*a.2*): Pawn rank ≤ 6

Analysis tree Advance ♙, ♚ any
= pawn-can-run pattern (with ♙ further advanced) : win

Justification (a program to search move trees using patterns of this kind needs to make these deductions): The pawn advance must be legal (in all positions matching the pattern) since ♔ is stated to be not in front and ♚ cannot be while the pawn-can-run pattern is satisfied. After the pawn advance, the pawn rank is then 1 more, and as any ♚ move can change rank and file by 1 at most, each inequality in the pawn-can-run pattern will still be true after Black's reply if it is true before ♙'s advance. Since one of them is true initially, the new node is also a pawn-can-run pattern.

Case (*b*): White King in front of the Pawn on the same file (not rook file)

Case (*b.1*): White King immediately in front of the Pawn

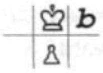


Figure 8.6: Case (b.1) of pawn-can-run pattern.

Analysis tree: ♔ plays to ***b***, ♚ any
= repeatable pattern 2 : win

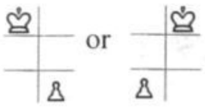
(The analysis of repeatable pattern 2 does not depend on case *b* of the pawn-can-run pattern.)

Justification: White's move is legal since ♚ cannot be adjacent to ***b*** while the pawn-can-run pattern is true. Nor can ♚ be adjacent to ***a*** in repeatable pattern 2 for the same reason.

Case (*b. 2*): White King not immediately in front of the Pawn.

Analysis tree: ♙ advance, ♚ any
either pattern (*b. 1*): win
or pattern (*b.2*) again with ♙ advanced : win

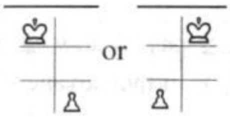
8.8.2 Repeatable Pattern 1



♙ not on rook file;
♚ anywhere

Figure 8.7: Repeatable pattern 1.

Case (*a*):

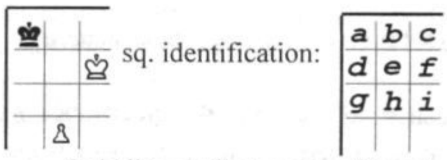


♙ not on rook file;
♚ anywhere; ♙ rank=6

Figure 8.8: Case (a) of repeatable pattern 1.

Analysis tree: Advance ♙, ♚ any
= terminal pattern 2 : win

Case (*b*):



Bold lines indicate edge of board

Figure 8.9: Case (b) of repeatable pattern 1.

Analysis tree: ♔ to ***h***, ♚ to ***b***, ♔ to ***g***, if ♚ to ***a***, advance ♙, ♚ to ***b***
= non-repeatable pattern 1 : win
if ♚ other, ♔ to ***d***, ♚ any
= case (f) of repeatable pattern 1 : win

Case (*c*):

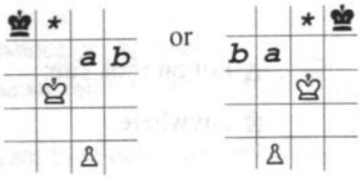


Figure 8.10: Case (c) of repeatable pattern 1.

Analysis tree: ♔ to ***a***, ♚ any, advance ♙, ♚ any, ♔ to ***b***, ♚ any
= repeatable pattern 1 (with ♙ further advanced): win

Case (*d*):

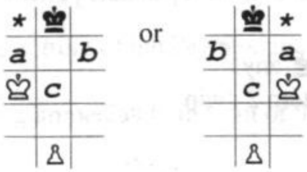


Figure 8.11: Case (d) of repeatable pattern 1.

Analysis tree: ♔ to ***c***, if ♚ to *, ♔ to ***b***, ♚ any, advance ♙, ♚ any
= repeatable pattern 1 (with ♙ further advanced): win
if ♚ other, ♔ to ***a***, ♚ any, advance ♙, ♚ any
= repeatable pattern 1 (with ♙ further advanced) win

Case (*e*):

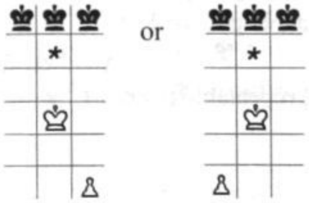


Figure 8.12: Case (e) of repeatable pattern 1.

Analysis tree: Advance ♔, ♚ any, advance ♙, ♚ any
= repeatable pattern 1 (with ♙ advanced): win

Case (f): Repeatable pattern 1, except cases (a-e)

Analysis tree: Advance ♙, ♚ any
= repeatable pattern 2 (with ♙ advanced): win

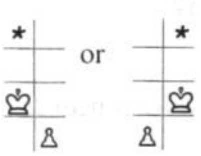
Justification for splitting into those cases (reasoning similar to this must be performed by a program to search move trees in pattern space): Case (a) is the highest possible pawn rank. It is examined first so that subsequent analysis can take advantage of the repeating pattern. Cases (b)-(e) are separated from case (f) by starting with repeatable pattern 1 ignoring ♚. The following analysis tree can then be obtained:

Advance ♙, ...
= repeatable pattern 2 (with ♙ advanced): win

In general, there are only four ways in which the black King's presence can invalidate black-king-less analysis: (i) Occupying a square that prevents any white move by obstruction; (ii) To be able to take the Pawn after a white move; (iii) To be in stalemate after a white move; (iv) If the final white move creates repeatable pattern 2, to be able to occupy the square * after it. When these four possibilities are applied to the analysis tree above, case (b) and cases (c)-(e) combined are distinguished from the residual case (f).

Splitting of the combined pattern (c)-(e) occurs when the white moves that are legal in all positions corresponding to the combined pattern (solidly legal) fail to lead to a win. The other three white moves must then be examined, and the patterns (c)-(e) result from dividing the combined pattern into pieces, in each of which one of the three moves is solidly legal.

8.8.3 Repeatable Pattern 2



♙ not on rook file;
♚ anywhere, except*
if ♙rank<5

Figure 8.13: Repeatable pattern 2.

Case (*a*): Repeatable pattern 2, as in Figure 8.13, with ♙ rank = 7

Analysis: = terminal pattern 2: win

Case (*b*):

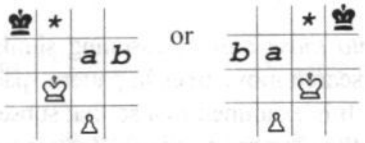


Figure 8.14: Case (b) of repeatable pattern 2.

Analysis: ♔ to ***a***, ♚ any, ♔ to ***b***, ♚ any
= repeatable pattern 1: win

Case (*c*):

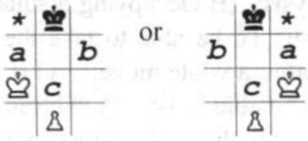


Figure 8.15: Case (c) of repeatable pattern 2.

Analysis: ♔ to *c*, if ♚ to *, ♔ to ***b***, ♚ any
= repeatable pattern 1: win
if ♚ other, ♔ to ***a***, ♚ any
= repeatable pattern 1: win

Case (*d*):

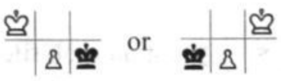


Figure 8.16: Case (d) of repeatable pattern 2.

Analysis: Advance ♙, ♚ any, advance ♔, ♚ any
= repeatable pattern 2 (with ♙ advanced): win

Case (*e*):

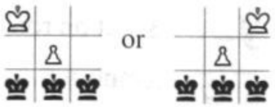
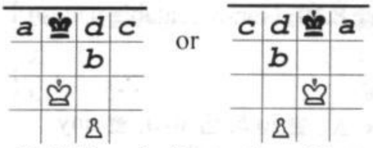


Figure 8.17: Case (e) of repeatable pattern 2.

Analysis: = case (a) of pawn-can-run pattern: win

Case (*f*):



Bold lines indicate edge of board.

Figure 8.18: Case (f) of repeatable pattern 2.

Analysis: Advance ♙,

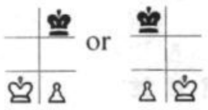
if ♚ to ***a***, ♔ to ***b***, ♚ any, ♔ to ***c***, ♚ any
= repeatable pattern 1: win

if ♚ to ***d***
= non-repeatable pattern 1: win

Case (*g*): Repeatable pattern 2, except cases (a)-(f)

Analysis: Advance ♔, ♚ any
= repeatable pattern 1: win

8.8.4 Non-Repeatable Pattern 1



♙ not on rook file;

♙ rank = 6

Diagram for analysis:

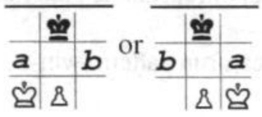
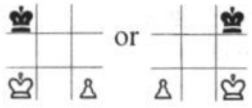


Figure 8.19: Non-repeatable pattern 1.

Analysis: Advance ♙, ♚ to ***b***, ♔ to ***a***, ♚ any
= terminal pattern 2 : win

8.8.5 Non-Repeatable Pattern 2



♙ not on rook file;

♙ rank = 5

Figure 8.20: Non-repeatable pattern 2.

Case (*a*):

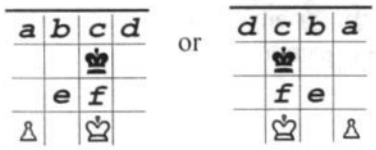
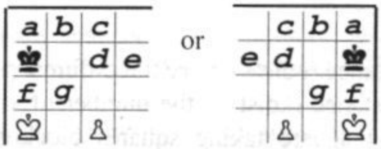


Figure 8.21: Case (a) of non-repeatable pattern 2.

Analysis: Advance ♙, if ♚ to ***b***, ♔ to ***e*** (call this pattern z),
if ♚ to ***a***
= non-repeatable pattern 1: win
if ♚ to *c*, advance ♙, ♚ any
= terminal pattern 1: win

if ♚ to *c*, ♔ to *f*, if ♚ to *b*, ♔ to *e*
= pattern z: win
if ♚ to *d*, advance ♙, ♚ any
= terminal pattern 1: win
if ♚ other, advance ♙, ♚ any
= terminal pattern 1: win

Case (b):



Bold lines indicate board edges.

Figure 8.22: Case (b) of non-repeatable pattern 2.

Analysis tree:

Advance ♙, if ♚ to *b*, ♔ to *g* (call this pattern y)
, if ♚ to *c*
= non-repeatable pattern 1: win
, if ♚ to *a*, ♔ to *d*, ♚ to original sq, ♔ to *e*
, ♚ any, advance ♙, ♚ any
= terminal pattern 2: win
if ♚ to *a*, ♔ to *f*, ♚ to *b*, ♔ to *g*
= pattern y : win

8.9 Initialising King-Move Sequences

Some classes of KPK positions seem to be simple to value because one King is clearly too far away to stop the other achieving its goals. The king-distance conditions derived below make this notion precise enough to define patterns and prove them correct.

The first step is to derive conditions for one King to reach a particular square despite interference from the other King. Conditions for reaching a given pattern can then be deduced. A decision cannot always be reached on the basis of distance alone. If the distances are nearly the same and the kings are either close or might become so, their ability to block each other's squares could favour either one, depending on the precise position. However, if $B \rightarrow S$ (the distance from one King to a target square) is sufficiently less than $W \rightarrow S$, (the distance from the other King to the same target square) or sufficiently greater, an immediate decision is possible.

Exact conditions for such decisions can be established with the aid of two measures of distance.

1. 'geometric' distance (gdist) = max(file difference, rank difference)
2. 'king-move' distance (kdist) = the number of king moves needed to reach the target square taking squares blocked by the Pawn into account, but not squares blocked or potentially blockable by the other King.

N.B. gdist always $\leq$ kdist.

There are two cases:

1. A sufficient condition that W (the King on move) can reach square T despite any move by B (the other King) is: kdist(W,T) > gdist(B,T).
2. A sufficient condition that B (not on move) can reach T, and, as a corollary, that W cannot, is: gdist(W,T) > kdist(B,T)+1.

It is possible to convince oneself of the truth of these conditions by informal reasoning, but a more-formal proof by induction offers greater protection against oversights:

Case 1: Let kdist(W,T) = n

a) If n = 0 then true since W on T already.

b) Suppose true for some n. Consider any square R for which kdist(R,T) = n+1, and any square for B such that gdist(B,T) > n+l. R must be adjacent to some square S for which kdist(S,T) = n. B cannot be adjacent to S since then gdist(S,T) $\leq n$ which would imply gdist(B,T) $\leq n$+l. Therefore B does not prevent W moving to S[2]. After B's reply to W's move to S, gdist(B,T) must

[2] This reasoning step would not be valid if Black's distance were calculated taking squares blocked by the Pawn into account, since B might be adjacent to S but have distance > 1 because it was unable to move there, e.g.

```
. . W
. S .
B . P
```

still be at least n+1 which is > kdist(S,T). By the induction hypothesis W can then reach T. Therefore true for n implies true for n+l.

Case 2:

a) True if kdist(B,T) = 0, since B already on T.

b) For kdist(B,T) = n, consider any square for W for which gdist(W,T) > n+1. After W's first move to any square S, gdist(S,T)>n. i.e. kdist(B,T) < gdist(S,T). Then by reversal of colours, case 1 implies that B can reach T.

From these single-square reachability conditions, king-distance conditions ensuring the reachability of repeatable patterns 1 and 2 may be deduced:

1. Let T = (either) square W of the white King in repeatable pattern 1. If kdist(W,T) < gdist(B,T) then W can reach its 'key' square in repeatable pattern 1. The only two ways this can fail to actually achieve the pattern are: (a) if Black can take the Pawn before W reaches its key square, and (b) if Black is stalemated. Kdist(B,P) > kdist(W,T) guarantees that Black cannot take the Pawn in the time that W needs to reach T. P rank < 6 guarantees that no stalemate is possible since all the stalemate positions have P rank = 6 or 7. Therefore:

kdist(W,T) < gdist(B,T) *and*
kdist(B,P) > kdist(W,T) *and*
P rank < 6 implies White can win.

2. Let T = (either) square W of the white King in repeatable pattern 2. If kdist(W,T) <gdist(B,T) then W can reach its key square. If kdist(B,*) > kdist(W,T) then B cannot defend by reaching *. If kdist(B,P) > kdist(W,T)–1 then Black cannot capture the Pawn (it must be done in one move less than kdist(W,T) since T is adjacent to P). If P rank < 6 no stalemate positions can occur. Therefore:

kdist(W,T) < gdist(B,T) *and*
kdist(B,*) > kdist(W,T) *and*
kdist(B,P) > kdist(W,T)–1 *and*
P rank < 6 implies W has a win.

If White's distance did *not* take blocked squares into account, W might be adjacent to S, but unable to go there. This illustrates the reason for having both *gdist* and *kdist* as measures of distance.

The single-square reachability conditions are only valid while P does not move. Every time P moves, kdist has to be redefined. This subtlety did not arise in the above king-distance conditions because they define positions in which White can reach a known win by king moves alone. The next king-distance condition defines positions in which B can reach P despite the efforts of W. Although primarily concerned with king moves, the analysis must also consider P moves. Rather than attempt to redefine single-square reachability, a direct proof by induction is given.

3. 'B rank $\geq$ P rank *and* gdist(W,P) > kdist(B,P)+l implies draw.'

Let kdist(B,P) = n.

a) True if $n = 0$. B has just taken P.

b) Suppose true for n.. Consider any square for B for which kdist(B,P) = $n + 1$, and any square for W such that gdist(W,P) > $n + 2$.

We now seek to prove that, after any white move, B has a reply that creates the kdist(B,P) = n case.

Neither a W or a P move (the double pawn advance is ignored) can reduce gdist(W,P) by more than 1, and kdist(B,P) = $n + 1$ implies that B has a move to a square S such that kdist(S,P) = n. So it is sufficient to show that W does not obstruct B to S after any W move and that kdist(B,P) is still $\leq n + 1$ after a P move.

(a) A W move to any square R: gdist(W,P) > $n + 2$ implies gdist(R,P) > $n + 1$. R cannot be adjacent to S since as kdjst(S,P) = n, gdist(S,P) $\leq n$ and therefore gdist(R,P) would be $\leq n + 1$.

(b) A P advance: Consider the geometry of the squares that P blocks from B. Let kd = kdist(B,P) and gd = gdist(B,P)

i) kd is either gd or gd + 1.
Hence, if kd after the P advance is to be > kd before, kd before must be gd and kd after must be gd + 1.

ii) If kd = gd + 1, B must be diagonally in front of P.
Hence, by geometry, if kd before = gd and kd after = gd + 1, gd after = gd before – 1.

iii) The P advance cannot increase gd as B rank $\geq$ P rank.

Therefore kdist(B,P-before) $\leq$ kdist(B,P-after).

Hence true for n implies true for $n + 1$.

8.9.1 Summary of King-Distance Conditions

(applicable to white-to-play positions only)

1. Win if:	kdist(W,W1) < gdist(B,W1) *and*
	kdist(B,P) > kdist(W,W1) *and*
	P rank < 6
2. Win if:	kdist(W,W2) < gdist(B,W2) *and*
	kdist(B,S) > kdist(W,W2) *and*
	kdist(B,P) > kdist(W,W2) – 1 *and*
	P rank < 6
3. Draw if:	kdist(B,P) < gdist(W,P) – 1 *and*
	B rank ≥ P rank

Figure 8.23: King distance conditions.

W, B and P denote the squares occupied by the white King, black King and white Pawn in the position under consideration. W1 and W2 are the squares W in repeatable patterns 1 and 2, when aligned so that P in the pattern coincides with P in the position. S is the square * in repeatable pattern 2.

8.10 The Difficulties of Retrograde Minimax with Patterns

This section discusses the difficulties that arise when attempting to perform retrograde minimax with patterns instead of individual positions. These difficulties would have to be overcome in order to achieve automation of the retrograde minimax process.

The first examples are based on the least ambitious description scheme for KPK (and hence the least difficult to reason about) one might consider. Each description gives the rank of each piece, but gives the files of the Kings relative to the Pawn, instead of absolutely. The Pawn is assumed to be on the queen side of the board (if we succeed in valuing all such positions, the values for positions with the Pawn on the king side can be obtained by mirroring.) The file of the Pawn is not specified, thus each description covers up to four positions. This choice originated from the observation that many patterns of pieces were consistently wins or losses irrespective of which file the Pawn was on. These descriptions of positions are referred to as VPF (for Variable Pawn

File) patterns. The most useful VPF patterns are those for which every matching position has the same value (consistent patterns). The following discussion shows how retrograde minimax tends to create inconsistent patterns, even when starting with consistent patterns.

Some VPF patterns have four positions corresponding to them (one for each possible file for the pawn). These are called here '*full*-patterns'. Others have only one to three. These are called here '*part*-patterns'.

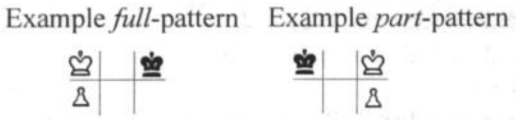
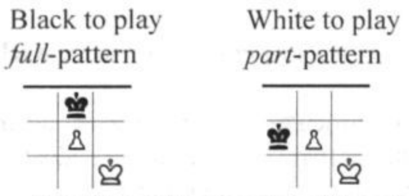


Figure 8.24: Example *full*-pattern and *part*-pattern.

Patterns adjacent to *part*-patterns are a problem, because they may have a move (to the *part*-pattern) which is legal in some occurrences of the pattern and not others. If this move is crucial (e.g. the only legal move) then the pattern may not have a common value over all its occurrences. Here is an example of a *part* pattern adjacent to a *full* pattern:



Bold line indicates edge of board.

Figure 8.25: Example *full*-pattern adjacent to *part*-pattern.

We wish to value the left-hand pattern, a *full*-pattern, by retrograde minimax (also called 'backing-up') from the right-hand pattern (amongst others). The right-hand pattern is consistent, but it is only a *part*-position. As a result, the left-hand pattern is not consistent, even though the pattern from which we are backing up is consistent. The right-hand pattern is single-valued, but because it is only a *part*-pattern, it leads to multiple-values for the pattern it backs up into.

Furthermore, in the retrograde minimax process, after valuing an inconsistent pattern, any VPF pattern adjacent to it may in turn be inconsistent. Therefore, it must be processed position by position instead of as a single pattern. This means any VPF pattern adjacent to either a *part*-pattern or to a multiple-valued pattern must be processed as individual positions instead of as a pattern[3]. Unfortunately, this includes most VPF patterns. Moreover, as a practical exercise in programming, the slight reduction in computation from the few VPF patterns that can be processed as single consistent entities is swamped by the overheads of handling patterns as well as positions.

The following examples illustrate the difficulties of manipulating more elaborate descriptions. They are based on a description scheme called here distance-based patterns. Each description consists of a logical combination (*and*, *or* and *not*) of predicates. Each predicate expresses a distance relation (=, > or <) either between squares, or between ranks and files.

Systematic valuing by backing up requires:

1. Generating descriptions one ply back (i.e. generating a description matching a set of positions such that each is one ply back from some position matching the starting description).
2. Generating and examining descriptions one ply forward again from the one ply back description (this will include the original description amongst others) to establish a minimax value for the backed up description if possible.

Unfortunately, this cannot be done by simple cycling through a small set of well-defined possibilities as it can be for position space.

Unlike position space and VPF-pattern space, descriptions are neither necessarily unique nor distinct. Also, it is not usually appropriate or feasible to divide up positions one ply distant into eight descriptions corresponding to the different directions of king moves.

Moreover, only descriptions with consistent values are useful. (By consistent is meant that all positions matching the description have the same minimax

[3] This might not be the case if suitable frame axioms about the relationship of various moves to possible changes in values could be found, but this seems very difficult indeed.

value.) Some means has to be found to restrict the generation of descriptions to consistent ones or detect and divide up inconsistent ones.

These problems can best be illustrated with an example. A description of White 1-ply wins is:

WTP and PR=7 and W$\rightarrow$Q = 1 (White to play, pawn rank = 7 *and* dist(white King, queening square) = 1)

or WTP and PR=7 and B Q > 1 and W $\rightarrow$ Q > 0

BTP and WTP mean Black to play, and White to play, respectively.

Generating a description of black 2-ply losses might proceed by generating first a description of losses one ply back from the first component (PR = 7 and W $\rightarrow$ Q = 1) of the white 1-ply wins and then generating a description of losses one ply back from the second component.

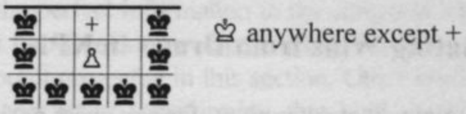
The first of these is actually feasible with this description and might go as follows:

> Black moves cannot affect this description component except for changing the side to play and so any position matching BTP and PR = 7 and W $\rightarrow$ Q = 1 is a position one black ply back. Each of these will be a loss if and only if all WTP positions one black ply forward are white wins. All black moves lead straight back into the original description and so "BTP and PR = 7 and W $\rightarrow$ Q = 1" positions are indeed 2-ply losses.

However, the second description component is a different matter. One set of positions one black ply backwards is obtained by changing B $\rightarrow$ Q > 1 into B $\rightarrow$ Q > 2. This set can be deduced to be consistent and a 2-ply loss for Black, since any black forward move still leaves B $\rightarrow$ Q > 1. The remainder of the patterns one black ply back provides all the difficulty.

The troublesome pattern one black ply back from component 2 is (BTP and PR = 7 and B $\rightarrow$ Q = 1 and W $\rightarrow$ Q > 0). This set is not consistent and it does not seem feasible to divide it into consistent subsets by reasoning about distances, or indeed by any process of reasoning sufficiently well-defined to program.

The positions may be illustrated by diagrams as follows.



Bold line indicates edge of board.

Figure 8.26: An inconsistent pattern.

Of these, by hand examination, the following positions are losses:

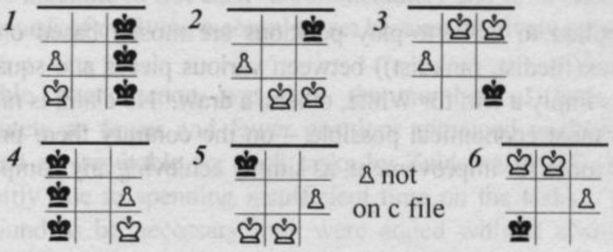


Figure 8.27 Cases within the inconsistent pattern.

These may be described using distances by:
BTP and PR=7 and W $\rightarrow$ Q > 0 and B $\rightarrow$ Q = 2 and W $\rightarrow$ * = 1 and B $\rightarrow$ * = 1 and
B $\rightarrow$ P > 1 where * = square to right of Pawn
and
BTP and PR=7 and W $\rightarrow$ Q > 0 and B $\rightarrow$ Q = 2 and W $\rightarrow$ S = 1 and B $\rightarrow$ S = 1 and B $\rightarrow$ P > 1 and B $\rightarrow$ A8 > 0 where S = square to left of Pawn, and A8 = 8th rank on Rook's file

Further positions one black ply back are obtained with B $\rightarrow$ Q = 1, but from hand examination none are black 2-ply losses.

The problem is that these descriptions were generated *ad hoc* as the need for them arose. No systematic deductive method of obtaining them is apparent.

Going further back the descriptions required to generate consistent subsets become more complicated still, essentially for reasons similar to those above. In general the requirement of consistency appears to lead to an unavoidable proliferation of predicates.

8.11 Discriminating Wins from Draws in KPK

A FORTRAN subroutine that determines, for any chess position with just King and one Pawn versus King, whether it is a win or a draw was developed. White-to-play positions are evaluated directly, by a series of tests on the ranks and files of the pieces. Black-to-play positions are evaluated by a 1-ply lookahead. The subroutine was tested against a complete table of KPK values (Clarke 1977) and is correct in all cases.

The tests applied to white-to-play positions are mostly based on geometric distances (max(filedist, rankdist)) between various pieces and squares. Some combinations imply a win for White, others, a draw. No claim is made that the tests are the most economical possible – on the contrary there probably is a great deal of room for improvement, as simply achieving any complete set was the goal.

This work grew out of the research into description generation and deduction in king and pawn endings. While studying the properties of geometric distances as a possible basis for position descriptions that might be suitable for mechanical generation, it was appropriate to see how convenient such descriptions were for classifying KPK positions, regardless of how the descriptions might be obtained. It was not necessary to create the subroutine described here for this purpose but having got part way surprisingly quickly it seemed worthwhile to finish it. Although KPK information had been programmed by several researchers (Bramer 1977, Harris 1977, Piasetski 1977, Tan 1972) and other chess endgame information by Bramer (1975, Huberman (1968), Michie (1977), Newborn (1977), Tan (1973, 1974, 1977), and Zuidema (1974), few were correct in all cases. A program known to be correct in all cases would be useful for comparison and reference.

No attempt was made to keep to 'mechanizable' descriptions when extending the classification rules to be correct on all KPK positions. The technique used was to compare systematically the current routine with the complete KPK table until several misclassifications had been accumulated, then study them and devise new or modified rules which resulted in correct classification for the wrong ones plus as many other cases as could be intuitively generalised from these examples. The modified routine was then tested, etc.

The feedback from the perfect information in the complete KPK table resulted in very quick (about two weeks) development of the routine. This is perhaps the most important point presented in this section. Other workers have spent a great deal of time and effort transforming this kind of information into a computer program. It may be that a large portion of this effort lies in a possibly unsuspected quarter, viz. being their own 'devil's advocate' when considering the correctness or otherwise of parts or proposed parts of their algorithm. I found it very much easier to suggest new classification rules (which might or might not be correct) that to test them for correctness. With a rapid-response machine to test each rule immediately and print exceptions and omissions, the task of evolving a complete set became relatively easy.

One noticeable phenomenon was that the number of tests increased disproportionately as fewer and fewer positions remained unclassified. This was anticipated as inevitable for such tasks by Zuidema (1974), but it may have been partly due to spending insufficient time on the tasks. When new rules were found to be necessary they were added without always looking examining existing rules to see if they were made redundant by the new rules, or looking for ways to combine them into more economical.

The Appendix gives a listing of the KPK routines, together with an explanation of how it can be used. Also in the Appendix is a concise representation of the decision rules, given as a decision table. There are 48 rules (or tests), which are applied in sequence 1-48. If one is found to be applicable it yields a value W or D (win or draw) and immediate exit from the routine. If no applicable rule is found the default is draw. A rule is applicable if and only if every condition is met. In other words, it is the logical 'and' of the conditions and there are no 'or'ed conditions. This voluntary restriction facilitated recording the rules as a decision table during development, which was convenient as it required little writing, was easy to alter, and compact enough to enable the rules to be viewed *en bloc*.

Figure 8.28 lists the number of configurations recognised by each rule but not by those proceeding it. N.B. The rules are not in chronological order of generation. New rules and modifications were sometimes accompanied by shuffling the order - one reason being to put simple and quick tests before more complex ones, another being to correct rules by testing for exceptions beforehand.

rule	*configs*	*rule*	*configs*	*rule*	*configs*	*rule*	*configs*	*rule*	*configs*
1	2	11	1	21	216	31	3469	41	448
2	3	12	24	22	315	32	63	42	39
3	1344	13	3	23	40	33	2527	43	342
4	3070	14	237	24	234	34	36	44	117
5	1789	15	235	25	150	35	1531	45	12
6	5	16	303	26	24	36	840	46	2
7	54336	17	27	27	48	37	171	47	12
8	7374	18	54	28	4783	38	63	48	9
9	8	19	1	29	1155	39	9		
10	3	20	5	30	387	40	60		

Figure 8.28: Number of configurations detected by each rule.

A subroutine that plays KPK can readily be constructed using KPKWV and KPKBV to provide the win/draw information. It is not quite enough however to ensure that the move chosen is a win if one exists. The choice must ensure progress towards the win, as procrastination can lead to a draw by repetition or by the 50-move rule. This is easy for KPK, and the following almost suffices for the K+P side:

> If there is only one winning move, make it.
> If advancing the Pawn wins, advance it.
> Otherwise, select the king move that leaves the King furthest up the board.
> If that still leaves a choice, choose the move that leads to the least difference between king file and pawn file.
> If still a choice, choose the one that leaves the King nearest the edge of the board.

Similarly, reasonably sensible play for the lone King can be achieved by aiming to keep as close to the Pawn as possible when faced with a choice.

These heuristics do not produce optimal play in the sense of always taking the fewest moves to win (or delaying the loss as long as possible), but they are intended to ensure a win is achieved whenever one is possible and sensible-looking play. I reserve the word 'correct' for such programs. It seems computationally feasible to establish whether or not a given program for KPK is correct by systematically marking all positions that lie on winning sequences chosen by the algorithm in a manner analogous to the standard retrograde minimax method, which marks minimax values.

It should perhaps be mentioned that to achieve correct and good-looking play for KRK (Bramer 1975, Huberman 1968, Michie 1977, Zuidema 1974) appears to require more complex heuristics than for KPK.

The difficulty in KPK resides mostly in deciding whether a win is possible, whereas in KRK almost all positions are wins and all the difficulty lies in choosing a move that wins rapidly.

This Chapter has been a case study for handling minimax with patterns instead of individual positions. Sections 8.3 to 8.9 examine in detail a set of patterns that can be proved correct by forward search in patterns space instead of position space. The task of handling the patterns is substantial even after the patterns are invented and made precise. Section 8.10 transfers insights to the original goal of this work, namely: retrograde minimax with patterns instead of positions. The prospects for automating this seem poor because *ad hoc* invention of new patterns with the aid of human intuition seems to play a vital part. Finally, section 8.11 reported on the successful attempt to produce a precise pattern set for the KPK endgame, which has been useful for comparison and reference. The Chapter has addressed the research question of this thesis by increasing our understanding of minimax search when applied to pattern space instead of position space.

rule	configs	rule	configs	rule	configs	rule	configs	rule	configs
1	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
2	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
3	11344	13	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
4	19070	14	9257	24	1734	34	26	44	117
5	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
6	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
7	54336	17	827	27	448	[illegible]	[illegible]	[illegible]	[illegible]
8	17374	18	584	28	[illegible]	38	563	48	9
9	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
10	[illegible]	[illegible]	[illegible]	30	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Figure 8.28: Number of configurations [illegible] by each rule.

A subroutine that plays KPK can easily be developed using KPK-[illegible] and KPK-[illegible] to provide the win/draw information. [illegible] however to ensure that the move chosen is a [illegible] progress towards the win, [illegible] by repetition or by the 50-move rule. [illegible] for [illegible] KPK:

If there is only one winning move, make it.
If advancing the Pawn wins, advance it.
Otherwise, select the king move that leaves the King furthest up the board.
If that still leaves a choice, choose the move that leads to the least
difference between King file and pawn file.
If still a choice, choose the one that leaves the King nearest the edge of the
board.

Similarly, reasonably sensible play for the lone King can be achieved by aiming to keep as close to the Pawn as possible when faced with a choice.

These heuristics do not produce optimal play in the sense of always taking the fewest moves to win (or delaying the loss as long as possible), but they are intended to ensure a win is achieved whenever one is possible and sensible-looking play. I reserve the word 'correct' for such programs. It seems computationally feasible to establish whether or not a given program for KPK is correct by systematically marking all positions that lie on winning sequences chosen by the algorithm in a manner analogous to the standard retrograde minimax method, which marks minimax values.

Chapter 9

The Choice of Search Envelopes[1]

In nearly all chess-playing programs, numerical estimates from heuristic evaluations are used with minimax lookahead. The lookahead produces a better value, or better move choice, than the direct evaluations would. Our investigations into minimax theory have uncovered mechanisms by which lookahead obtains its benefits: the mechanisms are intimately related to the structure inherent in the underlying problem. Search algorithms that respond to this underlying structure can be orders of magnitude more cost-effective than search algorithms using only alpha-beta. In particular, search pruning techniques previously described as heuristics, or ad-hoc procedures, can be re-interpreted as instances of general algorithms conforming to a theoretical model.

9.1 Heuristics

Most chess programs, and in particular the most successful ones, such as CHESS 4.5: (Slate and Atkin, 1977), BELLE (Condon and Thompson, 1982), CRAY BLITZ (Hyatt, Gower and Nelson, 1990) and DEEP BLUE (Seirawan, 1997) choose their move on the basis of a large minimax lookahead using an evaluation function that delivers a numerical score for any position. The minimax process selects, from the set of evaluations at maximum depth, one value to be 'backed-up' to the root node. (When move choice is required, values are obtained for the positions resulting from each of the available moves.) It is assumed, and well established empirically, that the backed-up values are better than direct evaluations. However, the theoretical justification

[1] This Chapter is an edited version of Beal (1984), Using Numerical Heuristics Effectively. Thanks are due to M. Somalvico and B. Pernici (Eds.) for permission to use the contents in this thesis.

particular value from the set of terminal evaluations as the backed-up value: each of the terminal evaluations is individually unreliable - why does the minimax process give us any more confidence in the one selected? Figure 9.1 illustrates the fact that the value backed up to the root is one of the terminal values.

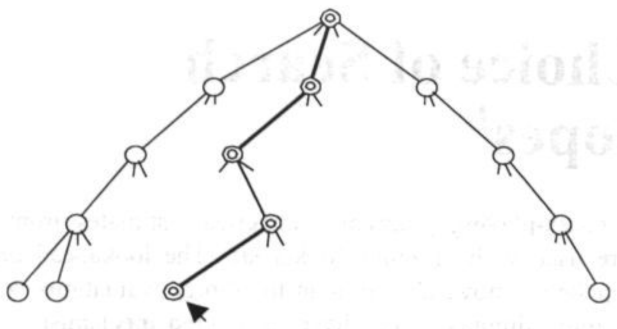


Figure 9.1: The selected node.
Why is the value backed up to the root more reliable than the others?

The chess programs use the alpha-beta pruning rule plus various refinements of the search procedure in order to get maximum benefit from alpha-beta cut-offs. It means that only some of the game tree is scanned. However, it does not affect the value the search obtains, or the move chosen (assuming that the move search order at every node remains the same[2]). So we can note that, although the alpha-beta technique is vital for efficiency, when discussing why the move was chosen and whether it was a good choice, we can refer to the complete tree and forget the alpha-beta pruning.

In addition, many chess programs perform some kind of heuristic pruning, e.g., a forward pruning of moves that do not appear to have sufficient merit. Heuristic pruning can always be regarded as defining some nodes to be terminal even though they are not at maximum depth. The most successful chess programs (CHESS 4.5, BELLE, CRAY BLITZ and DEEP BLUE) do not use heuristic pruning: Earlier versions of CHESS 4.5 used it, but then it was abandoned in favour of full-width search.

[2] If the move ordering is not the same, tie breaking between moves that obtain identical best minimax values may differ, but this does not disturb the main conclusions of the analysis in this section.

Since most of chess is far beyond complete calculation, choosing a move is a race to perform as much useful computation as possible within a time limit. The key question is: "what constitutes good value for money in a minimax lookahead?" The alpha-beta pruning technique is obviously a gain in value for money: it reduces the search effort without changing the result. Assessing heuristic pruning is more complicated. The pruning may change the result of the search as well as save tree-search time. We might rate it a gain in value for money if the search saving was sufficient to compensate for occasional worsening of move choice. In practice, we might not want to save search time: the time available for choosing moves is more or less fixed. The saving from prunes at one place in the tree will be used to enable deeper search in other places. So, in value for money terms: "does the deepening of search in those places improve the move choice more than occasional unsound prunes worsen it?"

Our results in minimax search theory, and the results of others (e.g., Bratko and Gams, 1982; Nau, 1982, 1983) aimed at answering the original question "why are backed-up values more reliable?" have also given new insight into heuristic pruning procedures.

9.2 From Minimax Search Theory to Search Envelopes

Chapter 2 analysed a preliminary model of minimax and discovered that, although the model did not appear unrealistic, any minimax problem conforming to it would give worse, not better, values with minimax lookahead. Nau (1981), in independent investigations, gave the name 'pathology' to this phenomenon of minimax lookahead making matters worse instead of better. He exhibited a simple game for which an obviously reasonable (and effective) evaluation function exists, yet minimax lookahead in this game produces worse values than direct evaluations.

Three independent papers in 1982 contributed to resolving this conflict with practical experience.

(1) Bratko and Gams (1982) published the results of analysing the preliminary model in more detail. They found that lookahead was still deleterious in several extensions to the model, but noted that more-reliable values could be created in local subtrees. They speculated that such more-reliable values could act as 'stabilising seeds' in large searches.

(2) Nau (1982) presented a modification of his pathological game which is created with the aid of values randomly assigned along the move sequence leading to each position, instead of directly to terminal positions. Since nearby positions share much of that move sequence, they are more likely to have the same value than distant nodes. In this game, there is no pathology.

(3) Beal (1982) described an extension to the earlier work, intended to model, in the simplest possible way, the reason for minimax being successful in practical game-playing.The model introduces non-uniformity in the otherwise random distribution of terminal values for the game tree. It assumes that, at any level, there is a tendency for values which are the same to cluster. The tendency is parameterised from 0 (no clustering = original model) to 1 (total clustering). The analysis shows that the greater the clustering, the greater the benefit of lookahead. It is suggested that a tendency to cluster mirrors the structure of real games, in which (a) terminal values are determined by board positions and (b) positions only change slightly with each move, therefore adjacent positions tend to have the same value.

Chapter 3, section 3.1 found that in the king and pawn against king endgame, for which complete data is available, substantial clustering occurs. If the branching factor and clustering parameter for the model are set to match KPK, then lookahead in the model is well into the beneficial range.

The clustering phenomenon produces its benefit for heuristic search at nodes where the side to play can win and has a choice amongst many moves which do so. The essence of the situation is that even if one or some of the winning moves are incorrectly valued, the chances are that a heuristic tree search will detect at least one as a win. As the KPK statistics show, such clusters can occur sufficiently frequently in games to make lookahead have an increasing probability of finding correct values with increasing depth. Figure 9.2 shows an example of a fragment of tree in which the backed-up value returned to the root is reliable (because heuristic errors are unlikely to propagate). The clusters are shown as triangles. The other branches are assumed to lead to nodes which are not clusters (i.e., nodes that have a mixture of successful and unsuccessful descendants).

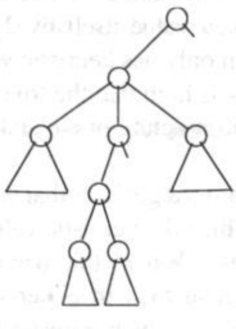


Figure 9.2: A cluster structure.

The minimax models use 'probability of error' as a figure of merit for evaluations. The benefit of lookahead is assessed in terms of the probability of error of the backed-up value, compared to the probability of error in direct evaluations. Thus the average effect of searches to given depths can be analysed.

The same technique could, in principle, be used to compare different heuristic pruning methods. The expected number of nodes in the search, together with the expected reduction of probability of error in the backed-up value, could be used to compute the 'value for money'. Unfortunately, typical evaluation functions, and typical heuristic estimators for "whether a move should be pruned" are elaborate, and assessing their effect is impractical.

However, the minimax models reveal one kind of node at which backed-up values become more reliable. It is possible to define search algorithms which, in addition to computing the backed-up value, also detect clusters. Going further, it is possible to define search algorithms which extend the search a few nodes at a time, examining the values obtained to see if the search has detected a reliable value at that point, and if so, stopping. Such an algorithm will search up to a fringe of more reliable values. Within the fringe is an irregular subtree of the complete game tree. This subtree need not be completely scanned, as alpha-beta pruning can be employed so that the usual reduction of branching factor provided by alpha-beta can be enjoyed within the subtree.

Chapter 3 on minimax models presented an algorithm called locked-value search which detects when sufficient clusters have been found to create a reliable (locked) value. It searches within a subtree delimited by locked values (although it overshoots the locked value itself by 2 ply in order to detect the clusters). Although the algorithm only has heuristic values available, when they indicate a cluster, the probability is high that the true values are clustered. Thus the algorithm is responsive to the structure of the underlying minimax tree.

Chapter 2 gave a simpler alternative algorithm that searches for a different kind of structure, namely, nodes at which the heuristic value agrees with the backed-up value from a 1-ply lookahead. Where this occurs one may have slightly more confidence in the joint value than in either of the two distinct values otherwise. A search up to a fringe of such 'consistent' values can be defined, and is given the name consistency search.

Both these algorithms respond to structure in the pattern of values obtained as the search proceeds. They are examples of heuristic pruning in the sense that nodes not at maximum depth are treated as terminal. However, they do not use any additional heuristics (i.e. beyond what is in the position evaluator) to do the pruning. They can be regarded as a conventional alpha-beta search within a search envelope defined by the consistent (or locked) values.

9.3 Search Envelopes

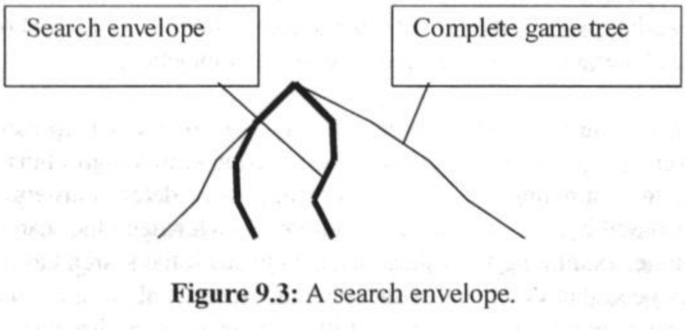


Figure 9.3: A search envelope.

Figure 9.3 illustrates that heuristic pruning rules can be regarded as defining a subtree of the full game tree, rooted at the same position, but narrower and (usually) irregular in shape. The leaf nodes of this subtree are the *search envelope*. The envelopes for consistency search and locked-value search are better value for money than a full-width envelope of the same number of nodes. The envelope determines the move chosen, and the selective envelopes have a greater chance of returning a reliable value, and hence a good move. Alpha-beta can be used within any envelope, and the combination of a good envelope and alpha-beta will be much more cost-effective than alpha-beta alone.

9.4 Envelopes for Well-Known Algorithms

A good choice of search envelope is critical to the achievements and the efficiency of the lookahead search process. Experiments by the author many years ago found that to achieve approximately the same 'probability of error' as a capture search in a selection of middle-game positions, required approximately 4 plies of full-width lookahead on the same evaluation function (i.e. material count only). Both the full width and the capture-tree searches were programmed to obtain good move ordering and hence near-optimal alpha-beta cut-offs. The full-width search required of the order of 50 times as many nodes to be scanned. Thus, viewing the capture-tree search now as a technique for choosing a search envelope which is better value-for-money than the full-width one, the improvement in cost-effectiveness is comparable to inserting alpha-beta into plain minimax.

Another practical algorithm is 'marginal forward pruning' (Slagle, 1971). This has been tried in many chess-playing programs, in particular by Kent and Birmingham (1977). The essence of the technique is to prune nodes which have a static evaluation less than the current value for alpha. Of course, search below that node may produce any value, so the technique is risky, and its intuitive justification is debatable. Nevertheless it seems in practice to be a useful compromise between full-width search and not searching at all. It turns out that applying consistency search to a complex evaluation function (i.e., much more elaborate than just material count) results in a very similar algorithm. If the evaluation produces values from a large or continuous range, it is natural to define values as consistent if they differ only by a small constant: this is the 'margin' in marginal forward pruning. The difference is: in consistency search the comparison should be with the static value of the parent, not the alpha value. In very many positions, the value of the parent will be limited by the current bound and the algorithms will coincide, but in other positions the difference will be significant.

The Kent and Birmingham paper reported that razoring produced an order of magnitude reduction in the number of nodes in a four-ply search. Most of the moves stayed the same, and it still found good moves even where they were different from the full-width choice.

More recently, Heinz (1998) reported reductions of 10-20% in tree size, while incurring hardly any loss of tactical strength, for extended futility pruning, which is similar to razoring but applied only at nodes within 2 plies of the horizon.

The envelope defined by razoring appears to be a more cost-effective one than full-width. Modifying it to be a consistency search may well be a further improvement. In any case, the question "what is the search envelope?" seems to be the right one to ask. Extensive research on the alpha-beta technique (e.g., by Fuller, Gaschnig and Gillogly (1973); Knuth and Moore (1975); Newborn, (1977); Baudet (1978)) has so far been concerned with fixed-depth full-width searches. It is known that alpha-beta is asymptotically optimal in that case (Pearl, 1980), and it is reasonable to guess that the same holds for arbitrary envelopes. The remaining question of which envelope is optimal has yet to be answered.

The interesting and perhaps unexpected insight resulting from the effort on modelling minimax is that definitions of good envelopes might come from game-independent techniques, and not necessarily from heuristics specific to a particular game.

The emphasis in AI is on knowledge-rich systems rather than on ones using extensive search. I do not intend to divert that emphasis. However, where knowledge is used in conjunction with search, it is important that the search be cost-effective if the knowledge is to yield its maximum benefit. The research on minimax models should be seen as establishing the background in which chess-specific knowledge within evaluation functions can be as effective as possible.

Chapter 10

A Generalised Quiescence Search Algorithm[1]

This Chapter describes how the concept of a null move may be used to define a generalised quiescence search applicable to any minimax problem. Experimental results in the domain of chess tactics show major gains in cost effectiveness over full-width searches, and it is suggested that null-move quiescence is as widely useful as the alpha-beta mechanism. The essence of the mechanism is that null moves give rise to bounds on position values which are more reliable than evaluations. When opposing bounds touch, they create a single value which is more reliable than ordinary evaluations, and the search is terminated at that point. These terminations are prior to any alpha-beta cut-offs, and can lead to self-terminating searches.

10.1 The Idea of the Null Move

The null move means changing who is to move without any other change to the game state. Some games, such as go, allow the null move as a legal move; others such as chess, do not. In the overwhelming majority of positions in either game, the null move would be a poor move, because there would be moves that did something beneficial for the side to play. This is true even if the side to play is losing, because even then the losing side is expected to be trying to minimise the loss, or delay it. Positions where the best possible outcome is obtained by the null move are really very rare. In chess they occur sometimes in the late stages of the game and, although they comprise a negligible fraction of positions encountered in practice, they have a special name, *zugzwang*. In a zugzwang position, every piece is either blocked or 'tied down' by some

1 This Chapter is an edited version of Beal (1990), A Generalised Quiescence Search Algorithm, in *Artificial Intelligence Journal*, Vol 43 pp.85-98. Thanks are due to Elsevier Science for permitting its use here.

null move is never useful during the main part the game, but as the game reaches the very end, both players are approaching situations where they would lose rather than gain by playing another stone. When a player thinks that there is no further gain for him, that player will 'pass' (= null move) and when both players pass consecutively, the game is over.

Since the null move is so rarely a good move (and not even legal in chess), why should it be included in minimax lookahead?

The answer lies in the structural properties of the computation that minimax lookahead is making, and has little to do with whether the null move is legal or not. It is also independent of the alpha-beta algorithm used to speed up minimax.

10.2 What Computation is Minimax Making?

Since CHESS 4.5 (Slate and Atkin, 1977), most chess programs, particularly the top performing ones, have used a search regime roughly characterized by: full-width search to a fixed depth, followed by a quiescence search. A typical quiescence search would be: captures and checks at ply 1, captures only after that.

Although the search is described as full-width, it is taken for granted that alpha-beta will be used. Essentially, the description above defines a tree to be searched, within which an alpha-beta search will look at as many moves as necessary. It is well-known that an alpha-beta search will give the same result as the vastly less efficient look-at-all-moves minimax.

The *computation* performed at each move is that of choosing a best move based on the tree to be minimaxed (a subset of the complete game tree), and the evaluation function values at all the tree;s endpoints. The factor being singled out for attention here is the tree to be searched, i.e., its shape and size.

Clearly, CHESS 4.5-style programs perform a different computation if the depth changes, or if changes are made in the rules governing which moves comprise the main searh or quiescence trees. The key questions would be "Is the computation any better?" and "How long does it take now?". In chess, except for certain relatively simple endgames, we have never been interested in any specific computation for move choice, but in getting "value for money", that is,

quality of move choice for time spent.
The points of the last paragraph are stressed because, unlike alpha-beta which gives drastically reduced computation times for the *same* computation, the null-move algorithms to be described produce a drastic reduction in computation time, but on a *different* computation. This makes it harder to judge the results.

In any case, we reached the most important question of all: "What shape and size should the search tree be or, in other words, which moves should be considered for searching?"

Note that alpha-beta should not be mentioned in the answer. Although alpha-beta indicates moves not worth searching, the context is different. Here the question is being asked of the tree *within which* alpha-beta is to operate.

The question is, of course, very old, perhaps the oldest in the 50-year history of computer chess. It is the question of selective search.

10.3 Attempts at Selective Search

Shannon's legendary paper of 1950, "Programming a Computer to Play Chess", described how minimax search should be of variable depth, only stopping at quiescent positions, and rejecting some moves at interior nodes of the tree. He called such strategies type-B, and suggested a function similar in spirit to CHESS 4.5-style quiescence search rules for stopping, but did not suggest an actual function for rejecting moves at higher nodes.

Almost all the early chess programmers did indeed try to restrict the search to a subtree of moves that include 'important' moves and exclude 'irrelevant' moves. The Bernstein program of around 1957, perhaps the most complete and effective of the very early chess programs, limited the search tree to the "best 7" at every node, using a sequence of plausible move generators to produce the best 7. Ten years later, the Greenblatt program (Greenblatt, Eastlake and Crocker, 1967) became the best program of its day using carefully chosen quiescence rules aming at tactical solidity.

Then, in 1973, CHESS 4.0 (later transmuted into CHESS 4.5 (Slate and Atkin, 1977)) made the successful exchange of speed and efficiency for selection, and set the pattern for the next fifteen years. There was however, a vitally important qualification of the new pattern of do-it-all-to-fixed-depth-and-do-it-fast.

Namely, the quiescence search. The endpoints of the fixed depth search were not places to apply the evaluation function, but places to start operating s simple and successful selective search, namely, the capture tree (augmented by one or two plies of checks if available). There were also, of course, the vital mechanisms of iterative deepening, transposition tables, alpha-beta by then taken for granted, and many ingenious programming techniques.)

Recently, Heinz (1997) has given a good summary of current practice in modern chess programs with an overview of the mechanisms used in his program DARKTHOUGHT.

There have been much more radical attampts to find better selective search mechanisms. Harris (1974) proposed a bandwidth search centered on promosing lines of play. The authors of the Russian program KAISSA (Adelson-Velskiy, Arlazarov and Donskoy, 1975), mention a variety of heuristics, including allowing the play of a null move when ahead in material. Berliner (1979) proposed the B* algorithm which uses separate optimistic and pessimistic estimates of position value at each node. Palay (1983) extended and developed B* to use probability distributions rather than optimistic-pessimistic ranges. Subsequently, McAllester (1985) presented an elegant method, called conspiracy numbers, based on counting critical positions.

In addition, there have been many attempts to use extensive domain-specific knowledge to create effective selective searches. Two major attempts that focussed on tactical play were Berliner's (1974) CAPS-II and Wilkin's (1979) PARADISE.

However, none of these methods have proved effective enough to become widespread.

10.4 Quiescence Search

Quiescence searches are, of course, selective searches. They derive from the idea of expanding the search just enough, and only just enough, to avoid evaluating a position where tactical disruption is in progress.

Chess players can identify many types of 'tactical disruption'. The simplest notion, which leads to the idea of a capture tree, is to say that tactical disruption is present if an immediate capture is available. More sophisticated definitions would take account of checks, forks, trapped pieces, and other tactical features, leading to more elaborate (and larger!) quiescence searches.

It is tempting, but not quite accurate, to think that a quiescence search could be defined by giving the rules for selecting the moves that make up the quiescence search tree. (If the rules produced no moves at a particular position the position would be terminal node of the tree. Alpha-beta would be used within the tree.) This description sounds complete if rather condensed, but it is not quite right. To see what is missing, consider the position of Figure 10.1.

The position in Figure 10.1 is not terminal. White has a capture available. However, it is a disadvantageous capture and White ends up losing two Pawns worth of material. This value (–2 Pawns) is the value returned by a quiescence search according to the definition just given.

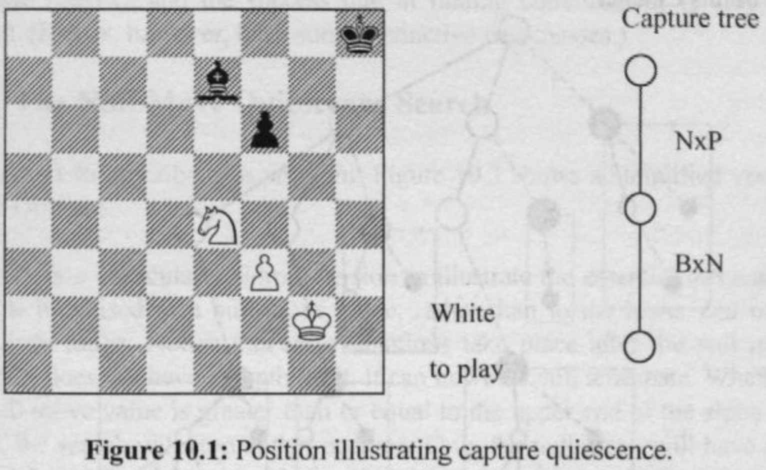


Figure 10.1: Position illustrating capture quiescence.

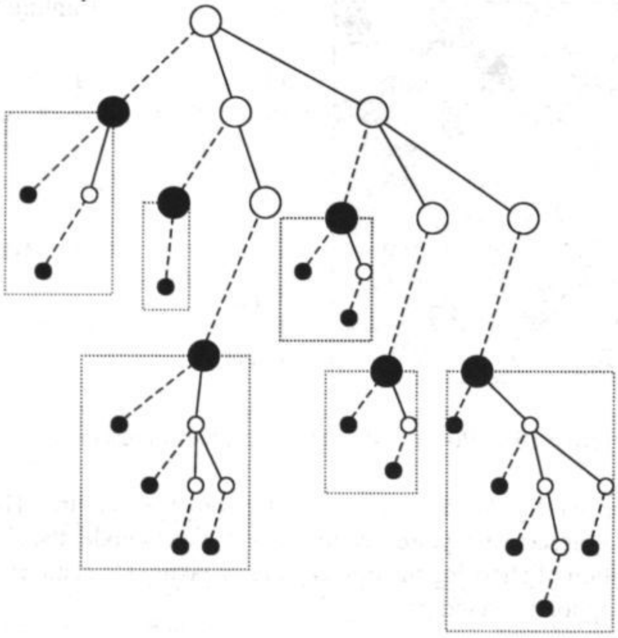
Clearly, this is wrong. White need not enter into the capture. The correct definition of a quiescence capture tree includes: "at each node, the side to play is given the option of choosing the best capture *or* taking the static evaluation." White has the option to 'stand pat'.

This option to 'stand pat' can equally well be viewed as an option to choose the evaluation after a null move, rather than the static evaluation now, since the material does not change with a null move. Although involving the null move seems an unneccessary complication at first sight, regarding the value as arising from playing the null move enables capture search to be seen as merely one special case of a general algorithm.

10.5 Game Independence

An interesting property of the concept of a capture tree is that definitions can be given in a game-independent way. By game-independent is meant that all game-specific knowledge is packaged in an evaluation function (in this case statically counting the values of pieces on the board), and then game-independent rules define the search regime. In Chapter 2, we saw that the game-independent rules for consistency search produce a capture tree from the material balance evaluation in chess.

However, there is another game-independent search regime, null-move quiescence search, that also produces a capture tree when given the material balance evaluation. That other regime is null-move quiescence, and that is the subject of this Chapter.



Dark circles are terminal nodes of quiescence searches
The second-order search is highlighted by larger nodes

Figure 10.2: Illustrative shape of search tree for second-order null-move quiescence.

Null-move quiescence becomes particularly interesting when it is 'bootstrapped'. Because it can be applied to *any* evaluation function, it can be applied as a second-order search to the values obtained by the first-order quiescence searches. The idea here is that just as 'counting material' can be packaged in a black box and called an evaluation function, so a result obtained by a null-move search can be packaged in a black box and called an evaluation function. Thus, the rules can be applied a second time at a higher level. Figure 10.2 illustrates the effect diagrammatically.

Applying this to material balance in chess, the second-level quiescence search, although it still contains no chess knowledge beyond counting up material, selects moves such as checks, moves out of check, attacks on pieces, defences, blocks, and other types of sharp tactical moves that chess players concentrate on. Perhaps is only an average of about four (compared to an average of about 35 legal moves), and the success rate in finding combinations is very high indeed. (It does, however, have some distinctive weaknesses.)

10.6 The Null-Move Quiescence Search

It is easiest to describe as a program. Figure 10.3 shows a simplified version, called QUIESCE.

QUIESCE is a particularly simple version to illustrate the essential mechanism. *bestv* is initialised to a null-move value, rather than to the lower end of the alpha-beta range. Notice that *all* evaluations take place after the null move. QUIESCE does not have a depth limit. It can however still terminate. Whenever the null-move value is greater than or equal to the upper end of the alpha-beta range, the search will stop at that position. Only the null move will have been explored, and that is evaluated directly. QUIESCE can be, and often is, a self-limiting, self-terminating search.

Also note that the function *evaluate* could itself be a search. Thus to obtain a second-order null-move quiescence search, QUIESCE would be defined with *evaluate* = material balance, and QUIESCE2 would be defined with *evaluate* = QUIESCE1.

```
QUIESCE(lower, upper) integer lower, upper;
{ integer bestv;
  makenull; bestv ← evaluate(-upper, -lower); unmakenull;
  foreach move m do
  {  if(bestv >= upper) return(bestv);
     make(m); v ← -QUIESCE (-upper, -bestv); unmake(m);
     if (v > bestv) bestv ← v;
  }
  return(bestv);
}
```

Figure 10.3: QUIESCE: the simplest version of null-move quiescence search.

The reader may have already noticed that QUIESCE, when using material balance as the evaluation function, will search all (or potentially all) moves at nonterminal nodes – not merely capture moves – despite the earlier claim that this quiescence mechanism would produce a capture tree. What will happen is that noncaptures will indeed be searched by QUIESCE, but will always terminate with a cut-off after the null move at the next ply. The single ply of search-and-reject behaviour for all noncaptures creates a 1-ply 'fringe' around a capture tree skeleton. The fringe could be optimised away in an implementation specialised to a capture search. The essential requirement for optimising away the fringe for QUIESCE is that an incremental change to the evaluation function can be computed without actually making the move. Any evaluation function which meets this requirement could be similarly optimised.

A more practical version of null-move quiescence would use iterative deepening, transposition tables, and ordering heuristics, as used with full-width minimax searches.

The question of a depth limit raises more subtle issues that it might seem. If the search is truncated at a depth limit, the result should be not a single value, but a triple. This is explained later.

10.7 Performance of the Generalised Null-Move Quiescence Search

Null-move quiescence is a general mechanism, capable of operating with any evaluation function. However, it is particularly effective at solving tactical problems in chess. First-order quiescence on material scores (capture trees) is well-known to be cost-effective. The performance of second-order quiescence on material was tested on the 300 tactical positions in the Reinfeld (1945) book, *Win at Chess*. This test set has been used on other chess programs and algorithms.

One modification was made to the definition of second-order quiescence. That was to enhace the value returned from the first-order quiescence (i.e., the result from a capture tree) with the ability to see checkmates. This is an ad hoc modification, but is a cost-effective compromise between putting the knowledge about checkmate in with the material balance evaluation (where it might be thought to belong, but where it is very costly in program time) and leaving it out altogether.

With this modification, material double-quiescence solved 276 of the 300 positions. Only positions where the right move was found for the right season, or the program demonstrated a flaw in Reinfeld's published solution, were counted as correct. This result can be compared with BELLE (Thompson and Condon, 1982), which was reported by Palay (1983) to only solve 273 within 3-minite per position limit.

The significance of this result is not so much that the score happens to be highre than 1983 BELLE, but the 'value for money' that it represents. The score itself *is* good considering the paucity of knowledge used, but the time taken is the real result. The result below show that BELLE's score can be obtained for approximately one-fifteenth of the effort.

The actual execution times (in Z8000 microcomputer seconds) are given in Table 10.1. "F" entries indicate that no solution was found because the material-only evaluation function has insufficient knowledge to solve the problem. These may be compared with BELLE times by dividing by 100, as BELLE looks at approximately 100 times as many positions per second (about 160,000 versus about 1500 for the Z8000 program). Of the solved positions, the longest took 16 seconds of BELLE-equivalent time. 273 positions (BELLE's score) can be obtained with a time limit of 12 BELLE-equivalent seconds per position, thus consuming only a fifteenth of BELLE's effort.

The comparison is on the basis of giving each program the full 180 (or 12) seconds on every position. Both programs need far less on most positions as Table 1 shows for the Z8000 program (remember that 1200 Z8000 seconds are needed for 12 BELLE-equivalent seconds). 235 positions were solved in less than 1 second of BELLE-equivalent time although, as times to solution are not given in Palay (1983), this cannot be compared with BELLE. The deepest search needed was for position 96 (= number 16 in test set 5), which went to 16 ply plus captures.

Apart from the see-checkmates enhancement, the experimental implementation differed from QUIESCE only in that special-purpose capture tree code was used for the first-order search, and that the second-order QUIESCE used iterative deepening with move ordering and already-calculated values obtained from the previous iteration. The program was written in assembly language, derived from a tournament chess program.

	Test sets (called quizzes in Reinfeld)														
Posn	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	3	42	F	1	7	F	2	57	31	1	1	29	213	124	35
2	F	F	2	29	20	1	5	4	19	7	1	53	8	1629	4
3	70	7	1	6	130	11	4	6	F	21	6	223	F	54	28
4	3	1	1	3	1	15	66	14	8	1	1205	17	32	70	315
5	1	1	1	23	1	30	18	22	15	153	91	2	789	1151	890
6	3	1	F	48	F	21	1	F	21	15	5	47	1	313	2
7	1	1	1	2	F	1	12	25	42	18	637	3	F	5	46
8	1	3	32	2	61	29	8	28	9	3	35	32	F	9	24
9	17	2	62	29	3	5	18	10	16	5	18	F	270	482	14
10	40	7	3	3	17	1	66	8	3	F	73	F	51	151	23
11	17	301	9	615	90	63	22	15	212	1	18	136	F	71	500
12	1	8	4	2	45	3	3	1	2	5	24	78	177	24	28
13	4	4	3	7	F	12	52	17	3	203	F	46	7	85	53
14	8	2	1	1	2	23	8	1	4	62	43	173	62	9	20
15	1	7	10	21	1	11	6	F	4	29	47	91	37	40	3
16	4	1	5	258	598	33	5	1	37	318	26	16	142	30	162
17	17	1	2	10	2	1	60	244	5	1	172	1241	45	32	F
18	34	1	27	43	7	32	11	3	192	20	5	35	20	47	5
19	4	1	6	17	2	10	40	5	4	10	2	F	F	29	F
20	1	1	1	107	F	37	5	1	730	32	776	649	114	886	23

Table 10.1: Times (seconds on Z8000) to solution for the 300 Reinfeld positions. Positions 92 (= number 12 in test set 5), 157, 210, 249, 264 and 296 showed Reinfeld's solution to be incorrect and were scored as "correct by demonstrating flaw". Positions 116, 129, 130, 149, 204, and 223 produced Reinfeld's solution move with lesser gain shown. These were also scored as "correct by demonstrating flaw".

10.8 Comparisons with Other Programs

As a comparison with knowledge-based programs, PARADISE (Wilkins, 1979) scored 89 out of the first 100 (compared to 92 for QUIESCE), and took about ten times as long. This is not really a meaningful comparison though, and Wilkins' work was actually a tour de force in the difficult research area of creating programs that use knowledge instead of search.

From private discussions, it seems that a few other programs have used the null move as an additional move in ordinary minimax. The idea is that in positions where the side to play is significantly ahead, the null move, although not the best, will nevertheless be sufficient to produce a cut-off, and, being a quieter move, may have a smaller subtree than the best move. This was the use mentioned in the KAISSA (Adelson-Velskiy, Arlazarov and Donskoy, 1975). This technique may have some benefit in full-width searches, but it involves doing a normal depth search below the null move, rather than direct evaluation as in QUIESCE. Closer to QUIESCE was the program MERLIN (Kaindl, 1983), which used the null move to find threats.

The work that is closest in spirit to QUIESCE is the B* algorithm of Berliner (1979), and the experiments performed by Palay (1983) on probabilistic modifications of B*. In B*, it is necessary to obtain upper and lower bounds for all positions in the tree. Berliner's paper on B* suggested that the bounds would come from evaluation knowledge. Palay experimented with a method which obtained bounds from a shallow (2-ply) search from the current position. The 2-ply search made two moves in succession for one side or the other, and thus provided biased results which were taken as bounds for each side. The main search would then proceed deeper from the current position, within the bounds found from the 2-ply search.

Conceptually, this has something in common with QUIESCE although Palay's algorithm is considerably more complicated. With an effort limit of 4 hours of VAX 11/780 timer per position Palay's program solved 245 of the Reinfeld problems. (4 hours of VAX 11/780 time is very roughly equivalent to 3 minutes of BELLE time.)

10.9 Reasons why Null-Move Quiescence is Effective

Null-move quiescence takes a given evaluation function, applies a search regime to it, and returns a value from the search which is more reliable than the given evaluation function. As can be seen from an inspection of QUIESCE, the value returned always originates as a null-move value.

Why should null-move values be more reliable than ordinary values? This is a key question closely related to the research questions of this thesis.

The answer is, "They are not." It is their role as *bounds* that is more reliable. That is, the statement "the value at position P is *at least* X", where X = *evaluate*(*nullmove*(P)), is much more reliable than the statement "the value at P is X", or "the value at P is *evaluate*(P)". In general, as the search proceeds, tighter and tighter bounds can be acquired, until a lower bound meets an upper bound. At this point, the search closes off with a single *reliable* value.

Thus the value that QUIESCE returns is not 'just' a null-move value, but is the result of two 'opposing' null-move values touching. Figure 10.4 illustrates the smallest possible tree,

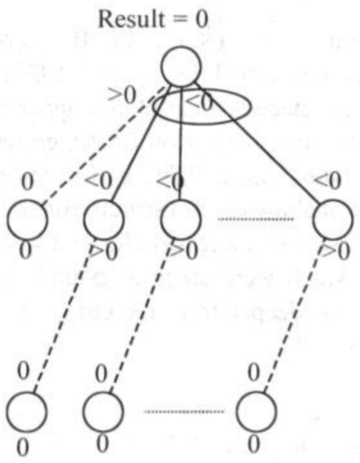


Figure 10.4: Smallest possible tree for QUIESCE. *Negamax convention*: numbers below node are from the point of view of player making move below; numbers above are from the point of view of players making move above. Dashed lines indicate null moves.

More typically, the top-level null-move value will not establish a closing bound. In this case, the depth-1 searches will not all consist of a single null-move evaluation, but some will expand to another ply of search.

A simple model of minimax (Beal, 1982) showed that the benefit of minimax lookeahead in random trees conforming to an overall clustering tendency increased with the clustering, and that the degree of clustering in a chess endgame (KPK) was amply sufficient for beneficial (rather than pathological lookahead behaviour.

The concepts of that model can be used to consider what the benefits of null-move quiescence lookahead might be. The result, not presented in detail here, is that if the null-move value was assumed to take values unrelated to other local values, then the error propagation properties were very similar to full-width lookahead. This would mean that the expected reliability of QUIESCE would be low, since the null-move values are raw evaluations, not deep searches, and can occur at any depth.

Perhaps it is not surprising though, that null-move quiescence would be useless if the null-move value was unrelated to other local values.

If instead a positive assumption is made about the properties of the null move, namely, that it is highly unlikely for a null-move value to be higher than the best of other moves, then the error-reduction factor changes from approximately

$$\frac{b}{1-k}\sqrt{k(k-f+kf)}$$

as derived in Chapter 3, to approximately zero. In other words that, with that kind of model, a null-move quiescence search removes all first-order error terms.

Of course, this all depends on the assumption that the null-move value is not higher than the best of the real moves. This has to be assumed separately both for true games values and the heuristic values being searched. The assumption about true values is an assumption that this position is not zugzwang. The assumption about the heuristic values amounts to an additional assumption that the evaluation function is really measuring something related to the game.

Both these assumptions seem to be reasonable in practice, and their thruthfulness could perhaps be measured for evaluation functions in chess, although this has not been attempted.

10.10 Depth-Limited Versions of QUIESCE, Fat Values and Nested Minimax

As the search proceeds, null-move values will narrow the range of possible values. These values will, of course, be passed around the tree for use by the alpha-beta mechanism. However, it should be noted than the bounds arising from null moves are *not* alpha-beta 'bounds'. Null-move values produce a bound on the actual value of the current position. Alpha-beta 'bounds' are a limitation on the range of interest we currently have in the actual value.

This distinction becomes important if the search is terminated prematurely (that is, by a depth limit). In this case null-move bounds can be returned as part of the top-level result (whereas it would not make sense to return alpha-beta 'bounds' as results). In general, the result will be a range (more picturesquely called a 'fat value') which is reliable plus an unreliable value wihtin it. At the terminal depth, only an unreliable point value is available, but as values are backed up, they can be combined with null-move values. The result of this combination is a reliable fat value with un unreliable point value located within it. The '<reliable range> + unreliable value inside' pattern gives rise to the concept of 'nested evaluations' or 'nested minimax' (*cf.* Chapter 4).

```
QUIESCE-D(lower, upper, d) integer lower, upper, d;
{ integer bestv;
  if(d = 0)  { v ← evaluate(lower, upper); return(-INF, v, INF); }
  LOWER ← -INF; UPPER ← -INF;
  makenull;  bestv ← -evaluate(-upper, -lower);  unmakenull;
  if(bestv > lower) LOWER ← bestv;
  foreach move m do
  {  if (bestv >= upper) return(LOWER, bestv, INF);
     make(m); Lm, vm, Um ← QUIESCE-D(-upper, -bestv, d-1); unmake(m);
     L ← -Um;  ← - vm;  U ← -Lm;
     if(L > LOWER) LOWER ← L;
     if(v > bestv) bestv ← v;
     if(U > UPPER) UPPER ← U;
  }
  return(LOWER, bestv, UPPER);
}
```

Figure 10.5: Depth-limited version of null-move quiescence search.

Figure 10.5 shows QUIESCE-D, a depth-limited version of QUIESCE. It handles two distinct reliability levels and therefore produces evaluations with one level of nesting. Two applications of QUIESCE-D, in the manner of Figure 10.2, would produce three reliability levels, and hence an evaluation with two levels of nesting. Nested evaluations would also arise from a minimax search that had recourse to a special-purpose evaluation that only became applicable occasionally.

Multiple applications of QUIESCE, or a variety of special-purpose evaluation mechanisms, could in principle produce evaluations with any number of levels of nesting. An example of the procedure for minimaxing with double nested evaluations is given in Chapter 6.

The QUIESCE of Figure 10.3 could, of course, be depth-limited without attempting to distinguish reliable values which arose from null-move termination from unreliable values obtained at the depth limit. This amounts to throwing away the fat value part of the result, and although the resulting algorithm is simple and more familiar, it loses valuable information. For example, an iterative search could be stopped when a definite result is obtained. Also, incomplete searches can yield valuable fat value information. Figure 10.6(a) shows an example where the final value is not known, but the move choice is already definite. Another interesting situation arises when the known values offer a choice between a safe move, [0, 0], and an uncertain move that may bring gain or loss, (–2, 3], as in Figure 10.6(b).

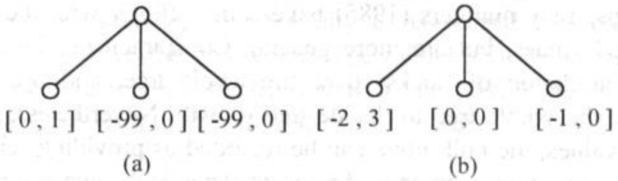


Figure 10.6: Move choices with fat values.

10.11 Comparison with Other Search Techniques

A good result on tactical problems in chess is relatively easy to achieve. There may well be positions and domains in which QUIESCE is not very successful. However, its simplicity and generality, and cost effectiveness in chess tactics suggest that it may be a better algorithm to start from than full-width searching (which has its own disadvantages). Its generality means that it could be applied to any minimax problem, thus making it a technique as widely applicable as alpha-beta. If it should turn out that material balance in chess was typical, rather than exceptional, then null-move quiescence would be as important as alpha-beta to cost-effective searches.

Improvements and extensions to QUIESCE can include not only iterative deepening, transposition tables, and move-ordering heustics, which are currently standard technology in full-width searches, but also additional interior evaluations to test for zugzwang, additional interior evaluations known to be of higher reliability in special situations, and mechanisms controlling the effort spent at different levels of the quiescence hierarchy.

The logic of the null move can be related to two other search algorithms. If the null move has a particular value, it is probably the case that several moves have at least that value. If one cares to assume any particular average number, say 6, then a closed-off null-move search is equivalent to a locked value of degree 6. Locked-value searches (*cf.* Chapter 3) are searches which continue until they strike small local configurations where more than *N* (the lock number) have to change to change the value of the configuration. More flexibly, each null-move bound can be regarded as preparing one side of an eventual double-sided lock on a deeper position.

McAllester's conspiracy numbers (1985) have some affinity with the earlier concept of locked values, but are more general. Conspiracy numbers allow incremental accumulation of 'locks' over the whole tree, and use global information over the whole tree to decide tree growth. Nevertheless, just as with the locked values, the null move can be regarded as providing (cheaply) the equivalent of a conspiracy number of 6 (or whatever is the average number of branches that obtain at least the value of the null move). This idea may improve the economics of conspiracy number algorithms.

Finally, this Chapter has covered important ground in addressing the research questions of this thesis. Sections 10.1 and10.2 described how the null move can be used to create a selective search. Sections 10.5 and 10.6 showed that not only is the selectivity game-independent, but also that it can be used to create hierarchies of increasing sophistication of search. Section 10.7 provided experimental evidence for the advantages of null-move quiescence. Section 10.9 provided an intuitive explanation for the benefits, and Section 10.10 made connection with Chapters 6 and 9 by showing how nested minimax is an appropriate method to administer the values discovered by null-move quiescence search. In effect Chapter 10 is the answer to the main research question of this thesis "does a better understanding of minimax search lead to improved search algorithms?"

Chapter 11

Conclusions

11.1 Deriving Effective Search Algorithms from Minimax Analysis

A major aim of modelling minimax trees and minimax search behaviour is to discover insights that can be used to construct more effective search algorithms. The model of Chapter 3 illustrates that game trees have stable regions and unstable regions. Stability is associated with nodes at which multiple moves give the best value. The model shows that the presence of such nodes causes fixed-depth search to improve the reliability of the backed-up value with each additional ply of search.

It is clear that search will be more efficient if it spends more of its time in unstable regions that affect the decision (where additional search matters) and less in stable regions (where additional search delivers unchanged values).

Thus, the minimax models and analysis lead to a coherent perspective for various domain-independent search algorithms that could be called *stability-sensing*. From this perspective we can regard different algorithms as trying to achieve stability-sensing by different means. The singular extensions algorithm (Anantharaman, Campbell and Hsu 1990) is based on an attempt to discover nodes at which stability is absent. If the attempt used full-depth searching to discover nodes with singular moves, the exponential cost would render the algorithm ineffective. But with a reduced-depth search to identify probably-singular nodes, the algorithm was reported to show a benefit. The concept of locked-value search in Chapter 3 used full evaluations to identify locked-values, and did not yield a cost-effective algorithm. The conspiracy-number algorithm (McAllester, 1988) is an elegant way to identify stable sub-trees, rather than individual stable nodes. However, it also fails to be where a version called proof-number search has been found to be effective (Allis, van der Meulen and van den Herik, 1994).

Null moves have been found to provide the most cost-effective stability-sensing mechanism in practice. The null move score is a lower bound on the actual score (except at zugzwang positions which are rare), and it is common for several real moves to achieve at least the null move score. Hence, null-move cut-offs are typically cutting off at stable nodes, and thus performing a shallower search at stable nodes (because the null-move search is conducted with reduced depth), than at unstable nodes, where the search goes to full depth.

11.2 Comparison of Null-Move Quiescence with Null-Move Depth Reductions

Chapter 10 introduced the theoretical idea of null-move quiescence, and showed that it has close correspondence with at least two successful algorithms for tactical search in chess. It was noted, however, that in practice a depth limit would be required, because the search could easily become too large. Null-move quiescence ceases to be cost-effective if the average size of the tree *after* searching below the null move becomes too large in relation the average size of the tree *below* the null move. (Because then the chance of an error in the null-move valuation dominates the overall chance of error, and the search effort in the rest of the tree does very little to reduce the overall chance of error.) This typically happens if the evaluation function is fine-grained (in contrast to material evaluations in chess which are very coarse-grained), or if the average branching factor of the null-move quiescence tree is a large fraction of the full move tree. In such cases, it is necessary to place a depth limit on the null-move quiescence search, and this would be the typical situation for most game-playing programs.

As an alternative to this, there is a simpler null-move algorithm. Goetsch and Campbell (1991) and Donninger (1993) report results. In this version, the search below the null move is not a fixed-depth search, but is depth reduced. Hence, as the overall search is iteratively deepened, the search below the null move also deepens. This contrasts with null-move quiescence, which would wait until the (depth-limited) quiescence search is complete before increasing the search depth below the null move. Depth-limited quiescence searches have a minor advantage in that there is less re-computation of the null move values which slightly increases search efficiency. However, the depth reduction algorithm discovers any incorrect null move values at shallower depths, is simpler to implement, and provides what can be regarded as a series of smoothly increasing levels of quiescence with increasing depth of the tree.

11.3 Retrograde Minimax with Patterns

The study of retrograde minimax with patterns for KPK reveals that although some patterns are easy to invent, and some of the reasoning steps required to perform retrograde minimax analysis are easy to perform, others are surprisingly hard. It is especially difficult to create mathematically rigorous proofs of pattern correctness, when working with retrograde minimax. Chapter 8 provides complete detail for an example set of KPK patterns, and the length of that material is perhaps discouraging for approaches that require individually crafted proofs for every pattern.

11.4 Conclusions on the Problem Statement

Chapters 2 and 3 answer research question 1, "can we increase our understanding of minimax search?" Sections 2.2 and 3.5 offer partial answers to research question 2 "does a better understanding of minimax models and minimax search lead to better search algorithms?" Chapters 4 and 5 offer specialised contributions to minimax search, showing how understanding of the technical details of the search process leads to algorithms that administer the search effectively. The algorithm presented in Chapter 4 preserves all the information obtained in searches which utilise evaluations of differing reliability, and Chapter 5 presents an algorithm that encodes the information obtained from conventional simple searches in an economical way. Chapter 6 gives examples of several different ways in which evaluations of differing reliability can arise. Chapter 7 illustrates another important way in perfect values intermix with heuristic values in practice, and reports on experimental results. Chapter 9 contributed to answering the research questions of this thesis by clarifying the concept of search envelope as distinct from the set of horizon nodes of the search. This led to Chapter 10 which discusses a very successful game-independent selective search technique, namely null move quiescence. Chapter 10 provides a decisively positive answer to research question 2, and makes the largest research contribution of this thesis.

11.5 Future Work

The application of minimax models to practical minimax lookahead searches provides only a partial explanation for the role of typical quiescence searches as implemented in many games programs in practice. The main search is usually conducted with null-move reductions, which partly satisfies the need of the overall search to go deeper in unstable, or non-quiescent, parts of the tree. But the last few plies of search are typically performed with a special-purpose quiescence search involving more than null-move quiescence. Typically, only moves which change the evaluation by large amounts are allowed to enter the search. The effect of this is similar to that obtained by forcing the evaluations to be low-resolution, or coarse-grained. This typically produces a very low branching factor null-move quiescence tree. Such a search seems to be highly cost-effective when the available time for search is so low that only a few nodes can be searched. It would be interesting to try to model the behaviour of searches with varying evaluation resolution, to explain, and perhaps ultimately improve, the performance of practical game algorithms.

It is also highly desirable to take the analysis of the null move algorithms a stage further, and construct parameterized models that model the error characteristics of realistic searches to arbitrary depths. Scheucher and Kaindl (1998) show that the probability of large errors decreases with increasing depth of search, and that the average size of errors decreases with increasing depth of search. A model that related such a formula for errors to the probability of null-move evaluations being incorrect could lead to better null-move algorithms. The advantage would come from adjusting the depth reductions in different parts of the tree to reduce the chance of overall error.

It is hard to see how to advance the state of the art in retrograde minimax using patterns. Clearly there is a need for better formal systems in which to perform the analysis. Even without that, it might be appropriate to examine more-complex endgames. But there is one less obvious aspect that I think is also worth investigating. That is the extent to which unpredictable (as opposed to logically deducible) pattern fusions occur during pattern handling. For example, and leaving out some detail, if a position with king diagonally in front of Pawn is a win by manoeuvre A, and the position with the King directly in front is a win by manoeuvre B, then it is possible form a compact description which says that if the King is one rank ahead of the Pawn, the position is a win. Such description fusions arise by serendipity, as the result of case analysis, and typically are not deducible from the component patterns by conventional logic. It would be interesting to know if descriptions that are only accessible by case analysis are essential for compact pattern sets. With the large amounts of computer power now cheaply available, it might be possible to explore sufficient pattern sets to illuminate the role of description fusions.

References

Adelson-Velskiy, G.M., Arlazarov, V.L., and Donskoy, M.V. (1975). Some Methods of Controlling the Tree Search in Chess Programs. *Artificial Intelligence*, Vol. 6, pp. 361-371. ISSN 0004-3702.

Allis, L.V., Herik, H.J. van den, and Herschberg, I.S. (1991). Which Games Will Survive? *Heuristic Programming in Artificial 2: the Second Computer Olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 232-243. Ellis Horwood Ltd., Chichester, England. ISBN 0-13-382615-5.

Allis, L.V., Meulen, M. van der, and Herik, H.J. van den (1991). Databases in Awari. *Heuristic Programming in Artificial Intelligence 2: the Second Computer Olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 73-86. Ellis Horwood Ltd., Chichester, England. ISBN 0-13-382615-5.

Allis, L.V., Meulen, M. van der, and Herik, H.J. van den (1994). Proof-Number Search. *Artificial Intelligence*, Vol. 66, No. 1, pp. 91-124. ISSN 0004-3702.

Arlazarov, V.L. and Futer, A.L. (1979). Computer analysis of a rook end game. *Machine Intelligence*, Vol. 9 (eds. J.E. Hayes, D. Michie, and Mikulich). Ellis Horwood Ltd, Chichester, England. ISSN 0162-8828.

Baudet, G.M. (1978). On the Branching Factor of the Alpha-Beta Pruning Algorithm. *Artificial Intelligence*, Vol. 10, pp. 173-199. ISSN 0004-3702.

Beal, D.F. (1977) *Discriminating wins from draws in KPK.* Technical Report, Dept. of Computer Science, Queen Mary College, London University, London, England.

Beal, D.F. and Clarke, M.R.B. (1980). The Construction of Economical and Correct Algorithms for King and Pawn against King. *Advances in Computer Chess 2* (ed. M.R.B. Clarke), pp. 1-30. Edinburgh University Press, Edinburgh. ISBN 0-85224-3774.

Beal, D.F. (1980). An Analysis of Minimax. *Advances in Computer Chess 2* (ed. M.R.B.Clarke), pp. 103-109. Edinburgh University Press, Edinburgh. ISBN 0-85224-377-4.

Beal, D.F.(1982). Benefits of Minimax Search. *Advances in Computer Chess 3* (ed. M.R.B.Clarke), pp. 17-24. Pergamon Press, Oxford. ISBN 0-08-026898-6.

Beal, D.F. (1983). Using numerical heuristics effectively. *Proceedings l'Intelligenza Artificiale ed Il Gioco Degli Scacchi* (eds. B. Pernici and M. Somalvico), pp. 131-136.

Beal, D.F. (1984). Mating Sequences in the Quiescence Search. *ICCA Journal*, Vol. 7, No. 3, pp. 133-137. ISSN 0920-234X.

Beal, D.F. (1984). Mixing Heuristic and Perfect Evaluations: Nested Minimax. *ICCA Journal*, Vol. 7, No. 1, pp. 10-15. ISSN 0920-234X.

Beal, D.F. (1985) (ed.). *Advances in Computer Chess 4*. Pergamon Press, Oxford. ISBN 0-08-029763-3.

Beal, D.F. (1986). Selective Search Without Tears. *ICCA Journal*, Vol. 9, No. 2, pp. 76-80. ISSN 0920-2234X.

Beal, D.F. (1989). Experiments with the Null Move. *Advances in Computer Chess 5* (ed. D.F. Beal), pp. 65-79. North Holland, Amsterdam, The Netherlands. ISBN 0-444-87159-4. A revised version is published (1990) under the title A Generalised Quiescence Search Algorithm. *Artificial Intelligence*, Vol. 43, No. 1, pp. 85-98. ISSN 0004-3702.

Beal, D.F. (1989) (ed.). *Advances in Computer Chess 5*. North Holland, Amsterdam. ISBN 0-444-87159-4.

Beal, D.F. (1990). A Generalised Quiescence Search Algorithm. *Artificial Intelligence*, Vol. 43, No. 1, pp. 85-98. ISSN 0004-3702.

Beal, D.F. (1991) (ed.). *Advances in Computer Chess 6*. Ellis Horwood Ltd., Chichester, England. ISBN 0-13-006537-4.

Beal, D.F. and Smith, M.C. (1994). Random Evaluations in Chess. *ICCA Journal*, Vol. 17, No. 1, pp. 3-9. ISSN 0920-234X. Also published in

Advances in Computer Chess 7 (eds. H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk), pp. 273-283. University of Limburg, Maastricht, The Netherlands. ISBN 90 6216 1014.

Beal, D.F. (1995). An Integrated-Bounds-and-Values (IBV). *ICCA Journal*, Vol. 18, No. 2, pp. 77-81. ISSN 0920-234X..

Beal, D.F. and Smith, M.C. (1995). Quantification of Search-Extension Benefits. *ICCA Journal*, Vol. 18, No. 4, pp. 205-218. ISSN 0920-234X.

Beal, D.F. and Smith, M.C. (1996). Multiple Probes of Transposition Tables. *ICCA Journal*, Vol. 19, No. 4, pp. 227-233. ISSN 0920-234X.

Beal, D.F. and Smith, M.C. (1997). Learning Piece Values Using Temporal Differences. *ICCA Journal*, Vol. 20, No. 3, pp. 147-151. ISSN 0920-234X.

Bell, A.G. (1978). *The Machine Plays Chess*? Pergamon Press, Oxford-New York-Toronton-Sydney-Paris-Frankfurt.

Berliner, H.J. (1974). *Chess as Problem Solving: The Development of a Tactics Analyzer*. Ph.D. Thesis. Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA.

Berliner, H. (1979). The B* Search Algorithm: A Best-first Proof Procedure. *Artificial Intelligence*, Vol. 12, pp. 23-40. ISSN 0004-3702.

Bernstein, A. (1958). A Chess-Playing Program for the IBM 704 Computer. *Proceedings of the Western Joint Computer Conference*, pp. 157-159.

Birmingham, J.A. and Kent, P. (1977). Tree-Searching and Tree-Pruning Techniques. *Advances in Computer Chess 1* (ed. M.R.B. Clarke), pp. 89-97. Edinburgh University Press, Edinburgh. Reprinted (1988) in *Computer Chess Compendium* (ed. D.N.L. Levy), pp. 123-128. B.T. Batsford, London. ISBN 0-85224-292-1.

Borel, E. (1921). La théorie du jeu et les équations intégrales à noyau symétrique. *Comptes Rendus de Académie des Sciences*, Vol. 173, pp. 1304-1308. English translation by L.J. Savage (1953), under the title: the theory of play and integral equations with skew symmetric kernels. *Econometrica*, Vol. 21, pp. 101-115.

Bramer, M.A. (1975). *Representation of Knowledge for Chess Endgames*. Report, Open University, Milton Keyness.

Bramer, M.A. (1977a) King and pawn against king: using effective distance. Report; Open University, Milton Keynes. (1977b). *Representation of knowledge for chess endgames: towards a self-improving system*. PhD Thesis; Open University, Mdton Keynes.

Bratko, I. (1978) Proving correctness of strategies in the AL1 assertional language. *Information Processing Letters*, Vol. 5, pp. 223-30.

Bratko, I. and Gams, M. (1982). Error Analysis of the Minimax Prinicple. *Advances in Computer Chess 3* (ed. M.R.B. Clarke), pp. 1-15. ISBN 0-08-026898-6.

Chinchalkar, S. (1996). An Upper Bound for the Number of Reachable Positions. *ICCA Journal*, Vol. 19, No. 4, pp. 181-183. ISSN 0920-234X.

Clarke, M.R.B. (1977) A quantitative study of king and pawn against king. *Advances in Computer Chess 1* (ed. M.R.B. Clarke), pp. 108-118. Edinburgh University Press. ISBN 0-85224-292-1.

Condon, J.H. and Thompson, K. (1982). BELLE Chess Hardware. *Advances in Computer Chess 3* (ed. M.R.B. Clarke), pp. 45-54. ISBN 0-08-026898-6.

Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R. (1972) *Structured Programming*. Academic Press.

Donninger, Chr. (1993). Null Move and Deep Search: Selective-Search: Selective-Search Heuristics for Obtuse Chess Programs. *ICCA Journal*, Vol. 16, No. 3, pp. 137-143. ISSN 0920-234X.

Fuller, S.H., Gaschnig, J.G., and Gillogly, J.J. (1973). *Analysis of the Alpha-Beta Pruning Algorithm*. Department of Computer Science, Carnegie-Mellon University.

Fürnkranz, J. (1996). Machine Learning in Computer Chess: The Next Generation. *ICCA Journal*, Vol. 19, No. 3, pp. 147-162. ISSN 0920-234X.

Gillogly, J.J. (1972). The Technology Chess Program. *Artificial Intelligence*, Vol. 3, pp. 145-163. ISSN 0004-3702.

Goetsch, G. and Campbell, M.S. (1991). Experiments with the null-move heuristic. *Computers, Chess, and Cognition* (eds. T.A. Marsland and J. Schaeffer), pp. 154-168. Springer-Verlag, New York. ISBN 0-387-97415-6.

Greenblatt, R.D., Eastlake, D.E., and Crocker, S.D. (1967). The Greenblatt Chess Program. *Fall Joint Computer Conference*, Vol. 31, pp. 801-810.

Gries, D. and Misra, J. (1978). A linear sieve algorithm for finding prime numbers. *CACM* 21, Vol. 12, pp. 999-1003.

Harris, L.R. (1974). The Heuristic Search Under Conditions of Error. *Artificial Intelligence*, Vol. 5, pp. 217-234. ISSN 0004-3702.

Harris, L.R. (1977). Listing of Algol 60 program circulated privately.

Heinz, E.A. (1997). How DARKTHOUGHT Plays Chess. *ICCA Journal*, Vol. 20, No. 3, pp. 166-176. ISSN 0920-234X.

Heinz, E.A. (1998). Extended Futility Pruning. *ICCA Journal*, Vol. 21, No. 2, pp. 75-83. ISSN 0920-234X.

Heinz, E.A. (1998). Efficient Interior-Node Recognition. *ICCA Journal*, Vol. 21, No. 3, pp. 157-168. ISSN 0920-234X.

Heinz, E.A. (1998). DARKTHOUGHT Goes Deep. *ICCA Journal*, Vol. 21, No. 4, pp. 228-244. ISSN 0920-234X.

Herik, H.J. van den (1980). Goal-directed Search in Chess End Games, Delft Progress Report, Delft University, Vol 5, No. 4, pp. 253-279. ISSN 03404-985x.

Herik, H.J. van den (1983). *Computerschaak, Schaakwereld en Kunstmatige Intelligentie*. Academic Service, Den Haag. ISBN 90-6233-093-2.

Herik, H.J. van den (1985). The Construction of an Omiscient Endgame Data Base. *ICCA Journal*, Vol. 8, No. 2, pp. 66-87. ISSN 0920-234X.

Herik, H.J. van den and Herschberg, I.S. (1986). Omniscience, the Rulegiver? *Proceedings of L'Intelligenza Artificiale Ed Il Gioco Degli Scacchi, III Convegno Internazionale* (eds. B. Pernici and M. Somalvico), pp. 1-17.

Herik, H.J. van den (1997). Two Decades. *ICCA Journal*, Vol. 20, No. 4, pp. 213-214. ISSN 0920-234X.

Hyatt, R.M., Gower, A.E., and Nelson, H.L. (1984). CRAY BLITZ. *Advances in Computer Chess 4* (ed. D.F. Beal), pp. 8-18. Pergamon Press, Oxford. ISBN 0-08-029763-3.

Junghanns, A. (1994). Fuzzy Numbers as a Tool in Chess Programs. *ICCA Journal*, Vol. 17, No. 3, pp. 141-148. ISSN 0920-234X.

Junghanns, A. (1998). Are there Practical Alternatives to Alpha-Beta?. *ICCA Journal*, Vol. 21, No. 1, pp. 14-32. ISSN 0920-234X.

Kaindl, H. (1983). Searching to Variable Depth in Computer Chess. *Proceedings of IJCAI-83*, pp. 760-762. Karlsruhe, FGR.

Kaindl, H. (1990). Tree Searching Algorithms. *Chess, Computers, and Cognition* (eds. T.A. Marsland and J. Schaeffer), pp. 133-158. ISBN 0-387-97415-6.

Knuth, D.E. and Moore, R.W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, pp. 293-326. ISSN 0004-3702.

Marsland, T.A. (1986). A Review of Game-Tree Pruning. *ICCA Journal*, Vol. 9, No. 1, pp. 3-19. ISSN 0920-234X.

Marsland, T.A. (1993). Single-agent and Game-tree Search, *Encyclopeadia of Computer Science and Technology* (eds Kent and Williams), Vol. 27, pp. 317-336, Marcel Dekker Inc. NY. ISBN 0-8247-2280-9.

McAllester, D.A. (1985). *A New Procecdure for Growing Mini-Max Trees*. Technical Report, Artificial Intelligence Laboratory, MIT, Cambridge, MA.

Michalski, R.S. and Negri, P.G. (1977) An experiment on inductive learning in chess end games. *Machine Intelligence*, Vol. 8 (eds Elcock and Michie), pp. 175-92. Ellis Horwood/Wiley.

Michie, D. (1976) An advice-taking system for computer chess. *Computer Bulletin* (ser. 2) 10, pp. 12-14.

Morrison, P. and Morrison, E. (1961*). Charles Babbage and his Calculating Engines*. Selected writings by Charles Babbage and others. Dover Publ. Inc., New York.

Nau, D.S.(1980). Decision quality as a function of search depth on game trees. *Technical report* TR-866. Computer Science Dept. University of Maryland.

Nau, D.S. (1981). Pearl's game is Pathological. *Technical report* TR-999. Computer Science Dept. University of Maryland.

Nau, D.S. (1982). An Investigation into the Causes of Pathology in Games. *Artificial Intelligence*, Vol. 29, pp. 257-278. ISSN 0004-3702.

Nau, D.S. (1983). An investigation into the causes of Pathology in Games. *Artificial Intelligence*, Vol. 21, pp. 257-278. ISSN 0004-3702.

Nau, D.S. (1983). Pathology on Game Trees Revisited, and an Alternative to Minimaxing. *Artificial Intelligence*, Vol. 21, pp. 221-244. ISSN 0004-3702.

Nelson, H.L. (1985). Hash Tables in Cray Blitz. *ICCA Journal*, Vol. 8, No. 1, pp. 3-13. ISSN 0920-234X.

Neumann, J. von (1928). Zur Theorie der Gesellschaftsspiele. *Math. Ann,*. Vol.100, pp. 295-320. Reprinted (1963) in 'John von Neumann Collected Works' (ed. A.H. Taub), Vol. VI, pp. 1-26. Pergamon Press, Oxford-London-New York-Paris.

Neumann, J. von and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Second Edition 1947. Princeton University Press, Princeton.

Newborn, M.M. (1977). The Efficiency of the Alpha-Beta Search on Trees with Branch-dependent Terminal Node Scores. *Artificial Intelligence*, Vol. 8, pp. 137-153. ISSN 0004-3702.

Palay, A.J. (1983). *Searching with Probabilities*. Report CMU-CS-83-145. Carnegie-Mellon University, Pittsburgh, PA.

Pearl, J. (1980). Asymptotic Properties of Minimax Trees and Game-Searching Procedures. *Artificial Intelligence*, Vol. 14, pp. 113-138. ISSN 0004-3702.

Pearl, J. (1980). A Simple Game-Searching Algorithm with Proven Optimal Properties. *Proceedings of the First National Conference on Artificial Intelligence*. Stanford.

Pearl, J. (1983). On the nature of pathology in game searching. *Artificial Intelligence*, Vol. 20, No. 4, pp. 427-453. ISSN 0004-3702.

Quinlan, J.R. (1979). Discovering Rules by Induction from Large Collections of Examples. *Expert Systems in the Micro-electronic age* (eds. R.S. Michelski, J.G. Carbonell, and T.M. Mitchell).

Reinfeld, F. (1958). *Win at Chess*. Dover Publications, New York.

Schaeffer, J. (1997). *One Jump Ahead*. Springer-Verlag New York, ISBN 0-387-94930-5.

Scheucher, A. and Kaindl, H. (1998). Benefits of using multivalued functions for minimaxing. *Artificial Intelligence*, Vol. 99, No. 2, pp. 187-208. ISSN 0004-3702.

Seirawan, Y. (1997), The Kasparov - Deep Blue Games. *ICCA Journal*, Vol. 20, No. 2, pp. 102-125. ISSN 0920-234X.

Shannon, C.E. (1950). Programming a Computer to Play Chess. *Philosophical Magazine*, Vol. 41, pp. 256-275. ISSN 0-38796496-7.

Skiena, S.S. (1986). An Overview of Machine Learning in Computer Chess. *ICCA Journal*, Vol. 9, No. 1, pp. 20-28. ISSN 0920-234X.

Slagle, J.R. (1971). *Artificial Intelligence: The Heuristic Programming Approach*. New York, McGraw-Hill.

Slate, D.J. and Atkin, L.R. (1977). CHESS 4.5 – The Northwestern University Chess Program. *Chess Skill in Man and Machine* (ed. P.W. Frey). Springer Verlag, New York. ISBN 0-387-90815-3.

Slate, D.J. (1984). Interior-node Bounds in a Brute-force Chess Program. *ICCA Journal*, Vol. 7, No. 4, pp. 184-192. ISSN 0920-234X.

Stiller, L. (1992), KQNKRR. *ICCA Journal*, Vol. 15, No. 1, pp. 16-18. ISSN 0920-234X.

Tan, S.T. (1972) *Representation of knowledge for very simple pawn endings in chess*. Report MIP-R-98; School of Artificial Intelligence, Edinburgh University.

Thompson, K. (1986), Retrograde Analysis of Certain Endgames. *ICCA Journal*, Vol. 9, No. 3, pp. 131-139. ISSN 0920-234X.

Thompson, K. (1997). 6-Piece Endgames. *Advances in Computer Chess 8* (eds. H.J. van den Herik and J.W.H.M. Uiterwijk), pp. 3-9. Universiteit Maastricht, Maastricht, The Netherlands. ISBN 9062162347.

Thompson, K. and Condon, J.H. (1982). BELLE Chess Hardware. *Advances in Computer Chess 3* (ed. M.R.B. Clarke), pp. 45-54. Pergamon Press, Oxford.

Thompson, K. and Condon, J.H. (1983). BELLE. *Chess Skill in Man and Machine* (ed. P. Frey), 2nd edition, p. 206. ISBN 387-90815-3.

Turing, A.M., Strachey, C., Bates, M.A. and Bowden, B.V. (1953),.Digital Computers Applied to Games, in *Faster than Thought*, Bowden, B.V. (ed.), Pitman, pp. 286-310.

Verhoef, T.F. and Wesselius, J.H. (1987). Two-ply KRKN: Safety Overtaking Quinlan. *ICCA Journal*, Vol. 10, No. 4, pp. 181-190. ISSN 0920-234X.

Wiener, N. (1948). *Cybernetics, or Control and Communication in the Animal and the Machine*. The Technology Press. John Wiley and Sons, Inc., New York. Herman at CIF, Paris.

Wilkins, D.E. (1979). *Using Patterns and Plans to Solve Problems and Control Search*. Ph.D. Thesis. Report STAN-CS-79-747, Stanford University, Stanford, CA.

Winkler, and Fürnkranz J.. (1997), A Hypothesis on the Divergence of AI Research. *ICCA Journal*, Vol. 21, No. 1, pp. 3-13. ISSN 0920-234X.

Appendix

This appendix gives a FORTRAN subroutine that determines, for any chess position with just King and one Pawn versus King, whether it is a win or draw. The routine is known to be correct by exhaustive checking against a database created by retrograde minimax.

Figure A.1 is a listing of the main routine, KPKWV, which yields a result of 1 if the position specified by PF (pawn file), PR (pawn rank), WF (white king file), WR, BF, BR and with White to play is a win, 0 if it is a draw.

Figure A.2 lists the lookahead routine KPKBV that returns –1 if the configuration specified with black to play is a loss, 0 if a draw. Both routines assume the Pawn is white and that it is on the Queen's side of the board. Values for the other KPK positions can be obtained by symmetry. The Pawn must not be on the 8th rank. Figure A.2 also lists the utility routine DIST that returns the distance betweeen squares.

```
      FUNCTION KPKWV(PF,PR,WF,WR,BF,BR)
      INTEGER PF,PR,WF,WR,BF,BR,DIST
      INTEGER PPR,BQ,BPP,WPP,BRPU,BRPUU,BRPUUU
      INTEGER BLPU,BLPUU,BLPUUU,WRPU,WRPUU,WLPU
      INTEGER WLPUU,WBDD,SGF,SGR,WSG,SDR,WSD,BSD,TBF
      KPKWV=1
      PPR=PR
      IF(PR.EQ.2) PPR=3
      IF(PF.NE.1) GOTO 2
      IF(BF.NE.3) GOTO 1
      IF(PR.EQ.7.AND.WF.EQ.1.AND.WR.EQ.8.AND.BR.GT.6) GOTO 98
      IF(PR.EQ.6.AND.WF.LT.4.AND.WR.EQ.6.AND.BR.EQ.8) GOTO 99
1     IF(BF.EQ.1.AND.BR.GT.PR) GOTO 98
      IF(PR.EQ.7.AND.BF.GT.2) GOTO 99
      IF(BF.LE.3.AND.BR-PPR.GT.1) GOTO 98
      IF(WF.EQ.1.AND.BF.EQ.3.AND.WR-PR.EQ.1.AND.BR-PR.EQ.1)
     1 GOTO 98
2     BQ=DIST(BF,BR,PF,8)
      IF(BQ.GT.8-PPR) GOTO 99
      MBPF=BF-PF
      IF(MBPF.LT.0) MBPF=-MBPF
      BPP=DIST(BF,BR,PF,PPR)
      WPP=DIST(WF,WR,PF,PPR)
      IF(PF.EQ.1.AND.PR.LE.3.AND.WF.LE.2.AND.WR.EQ.8
     1 .AND.BF.EQ.4.AND.BR.GE.7) GOTO 99
      IF(PF.NE.2.OR.PR.NE.6.OR.BF.NE.1.OR.BR.NE.8) GOTO 3
      IF(WF.LE.3.AND.WR.EQ.6) GOTO 98
      IF(WF.EQ.4.AND.WR.EQ.8) GOTO 98
    3 IF(PR.NE.7) GOTO 4
      IF(WR.LT.8.AND.WPP.EQ.2.AND.BQ.EQ.0) GOTO 99
      IF(WR.EQ.6.AND.WF.EQ.PF.AND.BQ.EQ.0) GOTO 99
      IF(WR.GE.6.AND.WPP.LE.2.AND.BQ.NE.0) GOTO 99
    4 BLPUU=DIST(BF,BR,PF-1,PR+2)
      WBDD=DIST(WF,WR,BF,BR-2)
      BRPUU=DIST(BF,BR,PF+1,PR+2)
      IF(PR.NE.6) GOTO 6
      IF(DIST(BF,BR,PF+1,PR).GT.1.AND.
     1  BRPUU.GT.DIST(WF,WR,PF+1,PR)) GOTO 99
      IF(PF.EQ.1) GOTO 5
      IF(BLPUU.GT.DIST(WF,WR,PF-1,PR)) GOTO 99
      IF(BR.EQ.8.AND.MBPF.EQ.1.AND.WBDD.EQ.1) GOTO 99
      IF(BR.GT.6.AND.MBPF.EQ.2.AND.DIST(WF,WR,BF,5).LE.1)
     1  GOTO 99
      GOTO 6
    5 IF(WF.EQ.1.AND.WR.EQ.8.AND.BF.EQ.2.AND.BR.EQ.6) GCTO 9
    6 MWPF=WF-PF
      IF(MWPF.LT.0) MWPF=-MWPF
      IF(PR.GE.5.AND.MWPF.EQ.2.AND.WR.EQ.PR.AND.BF.EQ.WF
     1  .AND.BR-PR.EQ.2) GOTO 99
      BRPU=DIST(BF,BR,PF+1,PR+1)
      WRPU=DIST(WF,WR,PF+1,PR+1)
      BLPU=DIST(BF,BR,PF-1,PR+1)
      WLPU=DIST(WF,WR,PF-1,PR+1)
      IF(PF.EQ.1.OR.PR.NE.5) GOTO 7
      IF(MWPF.LE.1.AND.WR-PR.EQ.1) GOTO 99
      IF(WRPU.EQ.1.AND.BRPU.GT.1) GOTO 99
      IF(WR.GE.4.AND.BF.EQ.WF.AND.BR-PR.GE.2
     1  .AND.MBPF.EQ.3) GOTO 99
      IF(WLPU.EQ.1.AND.BLPU.GT.1) GOTO 99
    7 IF(PR.EQ.2.AND.BR.EQ.3.AND.MBPF.GT.1.AND.
     1  DIST(WF,WR,BF,BR+2).LE.1) GOTO 99
      IF(WR-PR.EQ.2.AND.BR.EQ.PR.AND.MBPF.EQ.1.AND.
     1  MWPF.GT.1.AND.(WF-PF)*(BF-PF).GT.0) GOTO 98
      IF(PF.EQ.1.AND.WF.EQ.1.AND.WR.EQ.BR.AND.BF.GT.3)
     1  GOTO 99
      SGF=PF-1
      IF(WF.GE.PF) SGF=PF+1
      SGR=WR-(MWPF-1)
      IF(MWPF.EQ.0.AND.WR.GT.BR) SGR=WR-1
      WSG=DIST(WF,WR,SGF,SGR)
      IF(WR-PR-MWPF.GT.0.AND.WR-BR.GE.-1.AND.BPP-(WSG+(SGR-
     1  PPR)).GE.-1.AND.DIST(BF,BR,SGF,SGR).GT.WSG) GOTO 99
      MD=MBPF-MWPF
      IF(PF.NE.1.OR.BF.LE.3) GOTO 8
      SDR=BR+(BF-3)
      IF(SDR.GT.8) SDR=8
      IF(WR.GT.BR+1) SDR=BR
      IF(SDR.LE.PPR) GOTO 8
      WSD=DIST(WF,WR,3,SDR)
      BSD=DIST(BF,BR,3,SDR)
      IF(BSD-WSD.LT.-1) GOTO 98
      IF(BSD.LE.WSD.AND.MD.LE.0) GOTO 98
8     BRPUUU=DIST(BF,BR,PF+1,PR+3)
      IF(BRPU.GT.WRPU.AND.BRPUUU.GT.WRPU.AND.PR-WR.NE.PF-WF)
     1  GOTO 99
      IF(BRPUUU.EQ.0.AND.WRPU.EQ.1) GOTO 99
      BLPUUU=DIST(BF,BR,PF-1,PR+3)
      IF(PF.EQ.1) GOTO 9
      IF(BLPU.GT.WLPU.AND.BLPUUU.GT.WLPU.AND.PR-WR.NE.WF-PF)
     1  GOTO 99
      IF(BLPUUU.EQ.0.AND.WLPU.EQ.1) GOTO 99
9     WRPUU=DIST(WF,WR,PF+1,PR+2)
      IF(BRPUU.GT.WRPUU) GOTO 99
      WLPUU=DIST(WF,WR,PF-1,PR+2)
      IF(PF.GT.1.AND.BLPUU.GT.WLPUU) GOTO 99
      IF(BR.NE.PR) GOTO 10
      IF(MWPF.LE.2.AND.WR-PR.EQ.-1.AND.MBPF.NE.2) GOTO 99
      IF(DIST(WF,WR,BF-1,BR+2).LE.1.AND.BF-PF.GT.1) GOTO 99
      IF(DIST(WF,WR,BF+1,BR+2).LE.1.AND.BF-PF.LT.-1) GOTO 99
10    IF(PF.EQ.1) GOTO 11
      IF(BR.EQ.PR.AND.MBPF.GT.1.AND.
     1  DIST(WF,WR,PF,PR-1).LE.1) GOTO 99
      IF(BR-PR.GE.3.AND.WBDD.EQ.1) GOTO 99
      IF(WR-PR.GE.2.AND.WR.LT.BR.AND.MD.GE.0) GOTO 99
      IF(MWPF.LE.2.AND.WR-PR.GE.3.AND.BF.NE.PF
     1  .AND.WR-BR.LE.1) GOTO 99
      IF(WR.GE.PR.AND.BR-PR.GE.5.AND.MBPF.GE.3
     1 .AND.MD.GE.-1.AND.PPR.EQ.3) GOTO 99
      IF(MD.GE.-1.AND.PR.EQ.2.AND.BR.EQ.8) GOTO 99
11    TBF=BF-1
      IF(PF.GT.BF) TBF=BF+1
      IF(MBPF.GT.1.AND.BR.EQ.PPR.AND.
     1  DIST(WF,WR,TBF,BR+2).LE.1) GOTO 99
      IF(BR.EQ.PR.AND.BF-PF.EQ.-2.AND.
     1  DIST(WF,WR,PF+2,PR-1).LE.1) GOTO 99
      IF(PF.GT.2.AND.BR.EQ.PR.AND.BF-PF.EQ.2.AND.
     1  DIST(WF,WR,PF-2,PR-1).LE.1) GOTO 99
98    KPKW=O
99    RETURN
      END
```

Figure A.1

```
      FUNCTION KPKBV(PF,PR,WF,WR,BF,BR)
      INTEGER PF,PR,WF,WR,BF,BR,INCF(8),INCR(8),DIST
      DATA INCF/O,1,1,1,O,-1,-1,-1/,INCR/1,1,0,-1,-1,-1,0,1/
      KPKBV=O
      NM=0
      DO 1 I=1,8
      NBF=BF+INCF(I)
      IF(NBF.LT.1.OR.NBF.GT.8) GOTO 1
      NBR=BR+INCR(I)
      IF(NBR.LT.1.OR.NBR.GT.8) GOTO 1
      IF(DIST(NBF,NBR,WF,WR).LT.2) GOTO 1
      IF(NBF.EQ.PF.AND.NBR.EQ.PR) GOTO 2
      IF(NBR.EQ.PR+1.AND.(NBF.EQ.PF-1.OR.NBF.EQ.PF+1)) GOTO 1
      NM=NM+1
      IF(KPKWV(PF,PR,WF,WR,NBF,NBR).EQ.0) GOTO 2
1     CONTINUE
      IF(NM.GT.0) KPKBV=-1
2     RETURN
      END

      INTEGER FUNCTION DIST(F1,Rl,F2,R2)
      INTEGER Fl,Rl,F2,R2,FD,RD
      FD=F2-Fl
      IF(FD.LT.0) FD=-FD
      RD=R2-Rl
      IF(RD.LT.0) RD=-RD
      DIST=FD
      IF(RD.GT.DIST) DIST=RD
      RETURN
      END
```

Figure A.2

Figure A.3 is a representation of the logic of the decision rules for KPKWV which is partly a decision table and partly boolean expressions. There are 48 rules (or tests), which are applied in sequence 1-48. If one is found to be applicable it yields a value W or D (win or draw) and immediate exit from the routine. If no applicable rule is found the default is draw. A rule is applicable if and only if every condition is met. In other words, it is the logical 'and' of the conditions and there are no 'or'ed conditions. This voluntary restriction facilitated recording the rules as a decision table during development, which was convenient as it required little writing, was easy to alter, and compact enough to enable the rules to be viewed *en bloc*. The top part of the table gives some conditions in decision-table form; the lower part contains additional conditions (as boolean expresions) that are part of some rules.

Due to width limitations, the top part of the table is split into three sections. Figure A.4 is the key to the notation used.

Rule No:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	D	W	D	W	D	D	W	D	W	D	D	W	W	W	W	W
PF	=1	=1	=1	=1	=1	=1			=1	=2	=2					>1
PR	=7	=6		=7					≤3	=6	=6	=7	=7	=7	=6	=6
WF	=1	<4				=1			≤2	≤3	=4					
WR	=8	=6							=8	=6	=8	<8	=6	≥6		
BF	=3	=3	=1	>2	≤3	=3			=4	=1	=1					
BR	>6	=8							≥7	=8	=8					
WR–PR						=1										
BR–PR			>0			=1										
WR–BR																
BF–PF																
| WF–PF |													=0			
| BF–PF |																
BF–WF																

Rule No:	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
	W	W	D	W	W	W	W	W	W	D	W	W	D	D	W	W
PF	>1	>1	=1		>1	>1	>1	>1			=1		=1	=1		
PR	=6	=6	=6	≥5	=5	=5	=5	=5	=2							
WF			=1								=1					
WR			=8				≥4									
BF			=2								>3		>3	>3		
BR	=8	>6	=6						=3							
WR–PR				=0	=1					=2						
BR–PR				=2			≥2			=0						
WR–BR											=0	≥-1				
BF–PF																
| WF–PF |				=2	≤1					>1						
| BF–PF |	=1	=2					=3		>1	=1						
BF–WF				=0			=0									

Rule No:	33 W	34 W	35 W	36 W	37 W	38 W	39 W	40 W	41 W	42 W	43 W	44 W	45 W	46 W	47 W	48 W
PF	>1	>1		>1				>1	>1	>1	>1	>1	>1			>2
PR													=2			
WF																
WR																
BF																
BR													=8			
WR–PR					=–1					≥2	≥3	≥0				
BR–PR					=0	=0	=0	=0	≥3			≥5			=0	=0
WR–BR										<0	≤1					
BF–PF						>1	<–1								=–2	=2
| WF–PF |					≤2						≤2					
| BF–PF |					≠2			>1			≠0	≥3		>1		
BF–WF																

Rule	Additional conditions applying to specific rules
5	(BR–PPR) > 1
7	(B→Q) > (PP→Q)
8	(B→PP) – (W→PP) < –1 and (BR–PR) ≠ | BF–PF |
12	(W→PP) = 2 and (B→Q) = 0
13	(B→Q) = 0
14	(W→PP) ≤ 2 and (B→Q) ≠ 0
15	(B→[RP++]) > (W→RP) and (B→RP) > 1
16	(B→[LP++]) > (W→LP)
17	(W→[B – –]) = 1
18	(W→[BF,5]) ≤ 1
22	(W→[RP+]) = 1 andand (B→[RP+]) > 1
24	(W→[LP+]) = 1 and (B→[LP+]) > 1
25	(W→[B++]) ≤ 1
26	Sign(WF–PF) = Sign(BF–PF)
28	(B→PP) – ((W→SG)+(SGR–PPR)) ≥ –1 and (WR–PR) > | WF–PF | and (B→SG) > (W→SG)
29	(B→SD) – (W→SD) < –1 and SDR > PPR
30	(B→SD) ≤ (W→SD) and SDR > PPR and | BF–PF | ≤ | WF–PF |
31	(B→[RP+]) > (W→[RP+]) and (B→[RP+++]) > (W→[RP+]) and (PR–WR) ≠ (PF–WF)
32	(B→[RP+++]) = 0 and (W→[RP+]) = 1
33	(B→[LP+]) > (W→[LP+]) and (B→[LP+++]) > (W→[LP+]) and (PR–WR) ≠ (WF–PF)
34	(B→[LP+++]) = 0 and (W→[LP+]) = 1
35	(B→[RP++]) > (W→[RP++])
36	(B→[LP++]) > (W→[LP++])
38	(W→[LB++]) ≤ 1
39	(W→[RB++]) ≤ 1
40	(W→[P–]) ≤ 1
41	(W→[B – –]) = 1
42	| BF–PF | ≥ | WF–PF |
44	PPR = 3 and | BF–PF | – | WF–PF | ≥ –1

45	$\lvert BF-PF \rvert - \lvert WF-PF \rvert \geq -1$
46	BR = PPR and $(W \rightarrow [TTB{+}{+}]) \leq 1$
47	$(W \rightarrow [RRP-]) \leq 1$
48	$(W \rightarrow [LLP-]) \leq 1$

Figure A.3

P	Square Pawn is on
W	Square white King is on
B	Square black King is on
F	File. e.g. PP = file Pawn is on
R (as suffix)	Rank. e.g. WR = rank of white King
PP	= P unless PR=2 when = square in front of Pawn
Q	Square on which Pawn will queen. i.e. QF=PF, QR=8
→	Distance between squares. e.g. $W \rightarrow P = \max(\lvert WF-PF \rvert, \lvert WR-PR \rvert)$
L	Left side. e.g. LP = square to left of Pawn
R (as prefix)	Right side
+ (within [ ])	Up. e.g. [P+] = square in front of Pawn
– (within [ ])	Down
[ ]	Square. e.g. [RP+] is the sq. diagonally in front of P
[f, r]	Square where f is the file and r the rank
T	1 file towards Pawn. e.g. TBF = *if* BF > PF *then* BF–1 *else* BF+1
SD	A square defined by: SDF = 3; SDR = *if* WR ≤ BR+1 *then* min(8, BR+BF–3) *else* BR
SG	A square defined by: SGF = *if* WF<PF *then* PF–1 *else* PF +1; SGB = *if* WF=PF and WR>BR *then* WR–1 *else* WR–$\lvert$WF–PF$\rvert$+1

Files are numbered 1-8 from the Queen's rook file to the King's rook file; the Queen's side is the left side; the ranks are numbered 1-8 from White's side to Black's side; and the Black side is up.

Figure A.4: Key to non-standard notation.

Index

Summary

This treatise deals with one of the fundamental topics of intelligent behaviour in competitive environments – minimax. Minimax has a long history as both a theory of perfect evaluation of perfect-information two-sided competitive situations, and a foundation for algorithms that play games. The thesis comprises a collection of published contributions to the understanding of minimax search over a number of years, now revised and with modern references included.

Chapter 1 provides a brief history of minimax, focussing on attempts to understand why minimax search is effective in practice, and how that understanding may be used to increase the efficiency of minimax search.

Chapters 2 and 3 describe models of minimax for game trees that attempt to capture, in simplified form, the essential characteristics of minimax in practice. Chapter 2 illustrates that simple but apparently sensible models predict that minimax search is useless or worse. Chapter 3 shows that more sophisticated models that only apply to game trees having a certain property do predict the observed success of minimax search. Typical games do have the indicated property.

Chapters 4 and 5 discuss technical details of implementing minimax search. A mechanism for programming simple searches with an elegant unification of values and bounds on values, mapped onto a single numerical scale is given. More generally, a separate mechanism is given for keeping essential information about values and bounds discovered by more elaborate algorithms that utilise multiple types of evaluation.

Chapters 6 and 7 give examples of how practical instances of useful evaluation functions for game playing produce evaluations of varying reliability which benefit from the minimax mechanism for sophisticated searches presented in Chapter 4.

Chapter 8 provides a detailed case study of retrograde minimax. When a game tree becomes small or sufficiently tractable, it becomes conceivable to build a complete table giving the value of every configuration, by backwards minimax (as the original theory of minimax envisaged.) This Chapter investigates the possibility of performing minimax search and retrograde minimax using patterns instead of individual positions. Although this is highly attractive in the sense of greatly reducing the number of distinct cases to be examined, this case study reveals that continual acts of creativity are required in the processing of patterns.

Chapter 9 discusses the question of envelopes for effective minimax searches. Techniques for increasing search efficacy introduce selection into the search process and implicitly define a search boundary called here the search envelope. The cost-effectiveness of any minimax search is intimately linked to its search envelope, and the concept of search envelope is helpful in discussing different algorithms.

Chapter 10 presents a major contribution. A domain-independent technique for efficient selective search is presented, together with insight into why it is effective. The technique is called null-move quiescence and has been widely used in game-playing programs. Special cases of the algorithm are known under other names. The effect of the null move is to establish (at low cost) bounds on the values that a larger minimax search would obtain. Experimental results are given which illustrate the gains attainable from the technique.

The conclusion of the thesis is that analysis of models of minimax, in terms of both theory and practical algorithms, has yielded greater understanding of minimax search, and considerable improvements to practical algorithms. Finally, it also concludes that further work could yield further benefits.

Samenvatting

Dit proefschrift behandelt een van de fundamentele onderwerpen in competitieve omgevingen: het minimax-principe. Minimax heeft een lange geschiedenis zowel als het gaat om de theorie van de volmaakte waardering van met elkaar wedijverende toestandsbeschrijvingen van twee partijen die beide over volledige informatie beschikken, als waar het gaat om het fundamentele raamwerk voor spelalgoritmen. Het proefschrift bevat een verzameling van gepubliceerde bijdragen die in de afgelopen jaren het begrip van het minimax-zoekproces hebben verdiept; de bijdragen zijn volledig herzien en aangevuld met recente referenties.

Hoofdstuk 1 geeft een kort historisch overzicht van minimax. Het richt zich vooral op de pogingen om te begrijpen waarom het minimax-zoekproces zo effectief is in de praktijk en hoe het begrijpen ervan kan worden gebruikt om de efficiency van het minimax-zoekproces te verhogen.

De hoofdstukken 2 en 3 beschrijven modellen van het minimax-zoekproces voor spelbomen; de modellen proberen in vereenvoudigde vorm de wezenlijke karakteristieken van het minimax-zoekproces in de praktijk vast te leggen. Hoofdstuk 2 laat zien dat eenvoudige maar duidelijk gevoelige modellen voorspellen dat het minimax-zoekproces zinloos is of nog erger. Hoofdstuk 3 toont aan dat meer verfijnde modellen, die alleen worden toegepast op spelbomen die een bepaalde eigenschap hebben, de waargenomen successen van minimax inderdaad voorspellen. We merken op dat de gebruikelijke spelen de hierboven bedoelde eigenschap bezitten.

De hoofdstukken 4 en 5 gaan in op de technische details van het implementeren van het minimax-zoekproces. Er wordt een mechanisme beschreven voor het programmeren van eenvoudige zoekprocessen met een elegante unificatie van waarden en grenzen van waarden, die afgebeeld worden op een enkelvoudige numerieke schaal. Er wordt ook een afzonderlijk mechanisme beschreven dat meer in het algemeen wezenlijke informatie bijhoudt over waarden en grenzen, die ontdekt zijn door ingewikkelde algoritmen met behulp van verschillende manieren van evalueren.

De hoofdstukken 6 en 7 laten met voorbeelden zien hoe een praktische invulling

van gewone evaluatiefuncties voor spelprogramma's waarderingen opleveren met een uiteenlopende betrouwbaarheid; de verfijnde mechanismen voor het minimax-zoekproces zoals beschreven in hoofdstuk 4 toont de voordelen aan van het gebruik van het begrip betrouwbaarheid.

Hoofdstuk 8 beschrijft een gedetailleerde *case study* van het retrograde minimax-zoekproces. Wanneer een spelboom klein of voldoende handelbaar is, wordt het aanvaardbaar om een volledige tabel te maken die de waarde van elke configuratie bevat; dit gebeurt door een achterwaarts minimax-zoekproces (precies zoals dat in de oorspronkelijke minimax-theorie bedoeld was). Het hoofdstuk onderzoekt de mogelijkheid om het minimax-zoekproces en het retrograde minimax-zoekproces uit te voeren met gebruikmaking van patronen in plaats van individuele stellingen. Hoewel dit idee bijzonder aantrekkelijk is in de zin dat het aantal verschillende gevallen aanzienlijk verminderd wordt, toont de *case study* aan dat een voortdurend creatief handelen noodzakelijk is in het proces om tot een verantwoorde verzameling van patronen te komen.

Hoofdstuk 9 behandelt de vraag of er een omhullend proces bestaat voor een effectief minimax-zoekproces. De technieken die ontwikkeld zijn om de effectiviteit van het zoekproces te verhogen introduceren selectief zoeken in het zoekproces en definiëren impliciet een grens voor elk zoekproces; deze grens noemen we hier de omhullende van een effectief zoekproces (*the search envelope*). De kosten-effectiviteit van elk minimax-zoekproces is nauw verbonden met de *search envelope*. Het concept van de *search envelope* is derhalve nuttig in een vergelijking met verschillende andere algoritmen.

Hoofdstuk 10 bevat een belangrijke bijdrage. Er wordt een domein-onafhankelijke techniek voor efficiënt selectief zoeken beschreven, tesamen met het inzichtelijk maken waarom deze techniek effectief is. De techniek wordt *null-move quiescence* genoemd en wordt hedentendage wijd en zijd toegepast in spelprogramma's. Bijzondere varianten van de algoritme zijn bekend onder andere namen. Het effect van de *null-move* algoritme is om op goedkope wijze de grenzen van de waarden die een uitgebreid minimax-zoekproces zou opleveren, vast te stellen. Er worden experimentele resultaten gegeven die de opbrengsten illustreren die verkrijgbaar zijn met deze techniek.

De conclusie van het proefschrift is dat een analyse van de minimax-modellen zowel in termen van de minimax-theorie als in termen van praktische algoritmen, geleid heeft tot een groter inzicht in het minimax-zoekproces en tot aanzienlijke verbeteringen in praktische algoritmen. Tenslotte concludeert de auteur dat het werk voortgezet kan worden omdat meer resultaten bereikbaar zijn door een verdere analyse van het minimax-zoekproces.

Curriculum Vitae

Name:	Donald Francis Beal
Date of birth:	1 September 1948
Place of birth:	Woking, England
Nationality:	English

Education

1959-1966:	Skinners Grammar School, Tunbridge Wells, Kent
1967-1970:	Kings College, London University (BSc Mathematics)
1970-1971:	Birkbeck College, London University (MSc Computer Science with Distinction)

Membership of Professional Associations

Associate Fellow of the Institute of Mathematics and its Applications.
Member of the British Computer Society (MBCS and C.Eng).

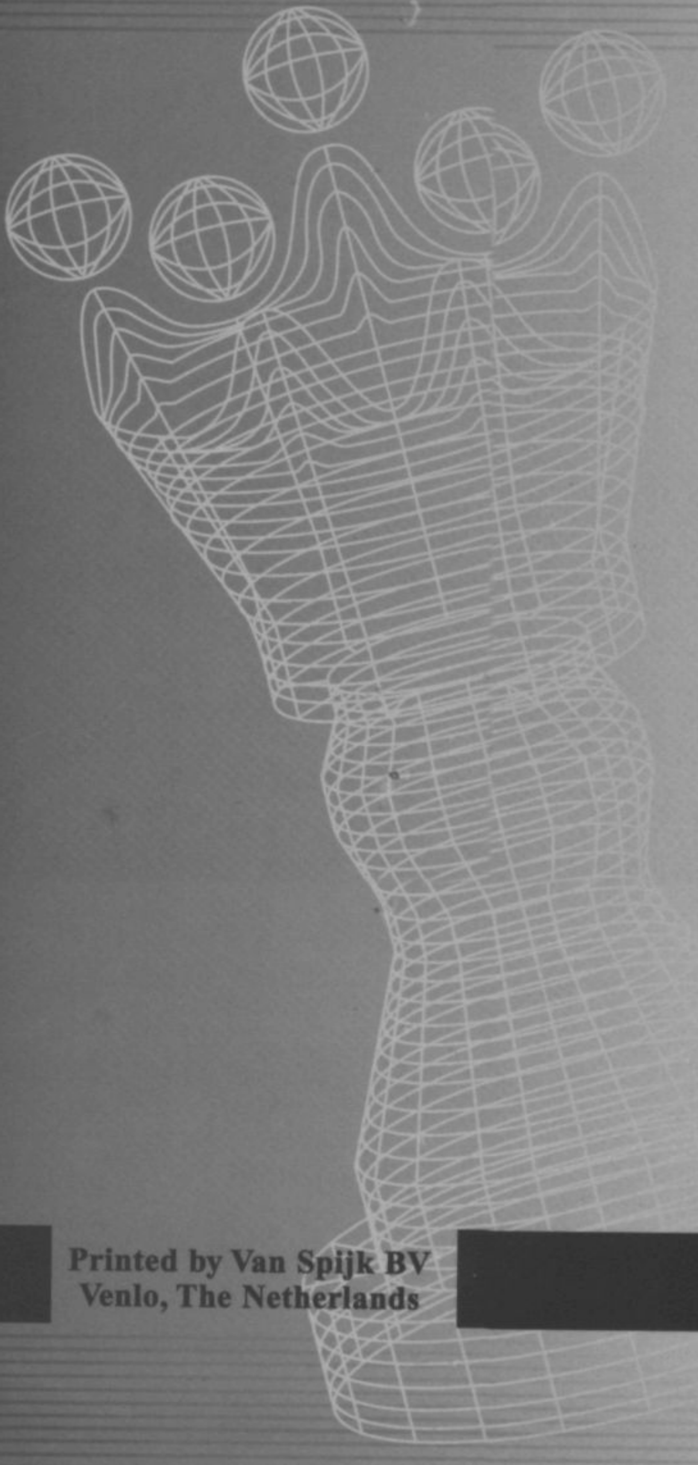
Employment and Research Experience

1971-1974	Scientific Officer in the UK civil service. Designed and implemented a program-development operating system.
1975-1977	Temporary lecturer, and then research assistant at QMC, London University, teaching computing and statistics to biologists, and researching knowledge representations.
1977-1983	Tenured lecturer at QMC, lecturing on Computer Science, and researching into Computer Architecture and Artificial Intelligence.
1983-present	Senior lecturer at QMC (now QMW), lecturing on topics including Expert Systems, Neural Networks, Computer Architecture, VLSI Circuit Design, Logic and Computation, Programming, Artificial Intelligence and continuing research into Computer Architecture, Hardware Design, and Artificial Intelligence.

SIKS Dissertatiereeks

In 1999 zijn de volgende SIKS-dissertaties verschenen.

99-1	Mark Sloof (VU)
	Physiology of Quality Change Modelling; Automated Modelling of Quality Change of Agricultural Products
Promotor:	Prof.dr. J. Treur
Co-promotor:	Dr.ir. M. Willems
Promotie:	11 mei 1999
99-2	Rob Potharst (EUR)
	Classification Using Decision Trees and Neural Nets
Promotor:	Prof.dr. A. de Bruin
Co-promotor:	Dr. C. Bioch
Promotie:	4 juni 1999
99-3	Don Beal (Queeen Mary and Westfield College)
	The Nature of Minimax Search
Promotor:	Prof.dr. H.J. van den Herik
Promotie:	11 juni 1999
99-4	Jacques Penders (KPN research)
	The Practical Art of Moving Physical Objects
Promotor:	Prof.dr. H.J. van den Herik
Co-promotor:	Dr. P.J. Braspenning
Promotie:	11 juni 1999



Printed by Van Spijk BV
Venlo, The Netherlands