ON FORWARD PRUNING IN GAME-TREE SEARCH

LIM YEW JIN

(B.Math., University of Waterloo)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2007

Dedicated to my family, especially Mum and Dad

Acknowledgements

I am very fortunate to have a loving wife, *Xin Yu*, who constantly reminds me that there is a life unrelated to research.

I have learnt a lot from *Lee Wee Sun*. I am grateful for his guidance in the course of my research, as well as his patience and willingness to share his opinions on my ideas in our weekly discussions. I am also indebted to *Jürg Nievergelt* for his wisdom and guidance, and for suggesting to me to try out the game of Tigers and Goats first. Portions of the text on Tigers and Goats had been co-authored with him. *Elwyn Berlekamp* pointed out Tigers and Goats and got us interested in trying to solve this game - an exhaustive search problem whose solution stretched out over three years.

As a collaborator, fellow student and friend, *Oon Wee Chong* has always been available for excellent help and suggestions in my research. I would like to acknowledge friends like *Weiyang*, *Yaoqiang* and *Yee Whye* who kept me sane and grounded during times of insanity.

And lastly, to those who I have not named, but have helped me in one way or another.

Thank you.

Contents

Summary

This thesis presents the results of our research aimed at the theoretical understanding and practical applications of forward pruning in game-tree search, also known as selective search. The standard technique used by modern game-playing programs is a depth-first search that relies on refinements of the Alpha-Beta paradigm. However, despite search enhancements, such as transposition tables, move ordering and search extensions, the game-tree complexity of many games are still beyond the computational limits of today's computers. To further improve game-playing performances, programs typically perform forward pruning, also known as selective search. Our work on forward pruning focuses on three main areas:

1. Solving Tigers and Goats - using forward pruning techniques in addition to other advanced search techniques to reduce the game-tree complexity of the game of Tigers and Goats to a reasonable size. We are then able to prove that Tigers and Goats is a draw using modern desktop computers.

2. Practical Application - developing a domain-independent forward pruning technique called RankCut for game-tree search. We show the effectiveness of RankCut in open source Chess programs, even with implemented together with other forward pruning techniques.

3. Theoretical Understanding - forming theoretical frameworks of forward pruning to identify two factors, the player to move and the depth of a node, that affect the performance of selective search. We also formulate risk-management strategies for forward pruning techniques to maximize performance based on predictions by the theoretical frameworks. Finally, we show the effectiveness of these risk-management strategies in simulated and Chess game-trees.

List of Tables

List of Figures

Chapter 1

Introduction

[When asked how many moves he looked ahead while playing]

Only one, but it's always the right one.

JOSÉ RAÚL CAPABLANCA Y GRAUPERA

World Chess champion between 1921 and 1927

Search is the basis of many Artificial Intelligence (AI) techniques. Most AI applications need to search for the best solution, given resource constraints, from many alternatives. Logically, such problems are trivial since we simply have to try every possibility until a solution is found. However, for practical game-playing programs, this strategy is not feasible. Expert humans can easily outplay computers in games with large branching factors such as Shogi, Bridge and Go due to the exponential growth in computational effort with increasing search depths.

Humans naturally perform selective search in game-tree searches. And we do it so well that the best human Chess players are still competitive with modern Chess programs that search in excess of 200 million Chess positions per second [Björnsson and

Newborn, 1997]. This approach keeps the exponential explosion in computational effort with increasing search depth manageable, as selective search only considers reasonable moves, thereby reducing the branching factor. The fact that humans can perform selective search so effectively has led experts to believe that full-width searchers in Chess would be dominated by selective searchers [Abramson, 1989]. However, selective searchers are difficult to implement correctly - in an early 4-game Chess experiment between a selective search program and a full-width search program, the selective searcher lost handily [Abramson, 1989].

This experiment illustrated the relative difficulty in implementing a selective search compared to a full-width search. While the premise of considering only "reasonable" moves is simple to vocalize, it is much harder to construct algorithms that can identify "good" and "bad" moves accurately. Nevertheless, effective selective search techniques such as search extensions (Section 2.2.5), Razoring, Futility Pruning, Null-Move Pruning, and ProbCut (Section 6.1), that have been developed to date have been shown to be effective in game-tree search. Despite these techniques, however, the exponential explosion of computational effort needed to search game-trees is beyond the computational limits of modern computers. Hence the need for effective forward pruning techniques has never diminished.

The goal of our research on forward pruning is to improve upon the state-of-the-art of both the practical application and theoretical understanding of forward pruning techniques in game-tree search. Our research comprises of work in several areas:

1. Combining co-evolutionary computing and neural networks to learn forward pruning heuristics for use in a forward search to help find the game-theoretic value of the game of Tigers and Goats. By using these forward pruning heuristics in addition to other advanced search techniques, we proved that Tigers and Goats is a

draw.

2. Developing a practical domain-independent forward pruning technique for game-tree search called RankCut that is effective even when combined with other forward pruning techniques.

3. Forming a theoretical understanding of forward pruning to identify the factors that affect the performance of selective search.

1.1 Tigers and Goats

The game of Tigers and Goats is the national game of Nepal. Tigers and Goats is a two-player perfect-information zero-sum game to which the Minimax paradigm is easily applicable. As it is played on a 5×5 board, it looks deceptively easy to solve. However, the game has an estimated game-tree complexity of 10^{41}. To give an idea of the size of this game-tree, we assume that a search program can process 10^9 positions per second. At this rate of searching, it will take approximately 10^{24} years to complete the search. It is therefore clear that advanced search techniques, domain-specific optimizations and selective search are needed to reduce the game-tree complexity to a reasonable size. Our work on Tigers and Goats resulted in a program that proved that Tigers and Goats is a draw using less than three days of computational time.

1.2 RankCut

Next, we introduce RankCut – a domain independent forward pruning technique that makes use of move ordering, and prunes once no better move is likely to be available.

Since game-playing programs already perform move ordering to improve the performance of Alpha-Beta search, this information is available at no extra cost. As RankCut uses additional information untapped by current forward pruning techniques, RankCut is a forward pruning method that can be used to complement existing methods, and is able to achieve improvements even when conventional pruning techniques are simultaneously employed. We implemented RankCut in modern open-source Chess programs to show its effectiveness.

1.3 Properties of Forward Pruning in Game-Tree Search

We also explore forward pruning using theoretical analyses and Monte Carlo simulations and show two factors of forward pruning error propagation in game-tree search. Firstly, we find that pruning errors propagate differently depending on the player to move, and show that pruning errors on the opponent's moves are potentially more serious than pruning errors on the player's own moves. While this suggests that pruning on the player's own move should be performed more aggressively compared to pruning on the opponent's move, empirical experiments with Chess programs suggest that this effect might not be that important in practical settings. Secondly, we examined the ability of the Minimax search to filter away pruning errors and give bounds on the rate of error propagation to the root. We find that if the rate of pruning error is kept constant, the growth of errors with the depth of the tree dominates the filtering effect, which suggests that pruning should be done more aggressively near the root and less aggressively near the leaves.

1.4 List of Contributions

The contributions of this research can be summarized as follows:

- *Learning Heuristic Players for Tigers and Goats*

  In this research, neural networks are evolved using evolutionary computing to reorder moves in searches that prove a specific hypothesis. This is different from the usual goal of learning how to play optimally against some set of opponents, and is shown to be effective in creating forward pruning heuristics. These forward pruning heuristics are used to show that Goat has at least a draw.

- *Finding the game-theoretic value of the game of Tigers and Goats*

  Through the use of carefully-crafted selective searches, the game of Tigers and Goats is weakly solved and found to be a draw under best play by both players.

- *RankCut*

  RankCut is a novel domain-independent forward pruning technique in game-tree search. It is designed to be simple to implement and has been shown to be highly effective in Chess, even when existing forward pruning techniques are used together with RankCut.

- *The Player to Move affects the propagation of forward pruning errors*

  We show that the player to move of a node affects how forward pruning errors propagate in game-tree search. To the best of our knowledge, this effect has not been observed before and we give a theoretical analysis and present empirical experiments to verify that this effect is present even in simulated and Chess game-trees.

- *Depth of a node affects the propagation of forward pruning errors*

We show that the depth of a node in the search affects the propagation of forward pruning errors in game-tree search. We derive a theoretical analysis that shows that the rate of error propagation increases with increasing search depth, and show evidence that this effect is present even in simulated and Chess game-trees.

1.5 Thesis Outline

This thesis is organized as follows. Chapter 1 gives a summary of the thesis and outlines the contributions of our research.

Chapter 2 introduces the basic game-tree search techniques and the more advanced search enhancements used in current game-playing programs.

Chapters 3, 4 and 5 explain how the game of Tigers and Goats was solved. Chapter 3 describes the game of Tigers and Goats, and provides an analysis on the state-space complexity and game-tree complexity of the game. An introduction to how other games have been solved is also given. Chapter 4 presents the evolutionary computation method used to create heuristic players employed during the forward search to show that Goat has at least a draw. Chapter 5 outlines the techniques used to show that Tiger has at least a draw, thus weakly solving the game of Tigers and Goats. This solution involved intensive computation on numerous machines over a time period of approximately three years.

Chapter 6 describes RankCut, which is a forward pruning technique in game-tree search. It is designed to be simple to implement and has been shown to be highly effective in Chess, even when existing forward pruning techniques are used together with RankCut.

Chapters 7 and 8 show how the player to move and the depth of a node affects

the propagation of forward pruning errors during game-tree search. To the best of our knowledge, the player to move effect has not been reported in the literature. The depth of a node effect is novel as an analysis of forward pruning although it builds on prior work of *Minimax pathology*, or the property that Minimaxing amplifies errors as search depth increases.

Chapter 9 concludes this thesis with a summary and a look at areas for future research.

Chapter 2

Game-Tree Search

This thesis studies the theory and practice of forward pruning in game-tree search. It presents research on applications of forward pruning in game-tree search to solve and play games, and a theoretical analysis of forward pruning in game-tree search. In this chapter we introduce game-tree search and search enhancements, and outline how they are employed in game-playing programs.

2.1 Game-Tree Search

AI techniques have been applied to board games for the past 40 years. For example, Chess has been a popular testbed for AI techniques, and one of the most memorable result of such research is the defeat of reigning world champion Garry Kasparov to a computer system named DEEP BLUE under regular time controls in 1997.

The underlying algorithm typically used for AI in board games is based on the Minimax paradigm. The Minimax paradigm can be implemented by game-tree search algorithms. In this thesis, we will focus on *two-player zero-sum games with perfect information*. The term *two-player* simply refers to a game that involves two players. The

term *perfect information* means that the states of the game are completely visible to all players. In contrast, the term *imperfect information* means that states of the game are only partially observable, and therefore some relevant information is hidden from the players. *Zero-sum* means that the gain of a player is the loss of his or her opponent. Let $score_A(p)$ and $score_B(p)$ represent the scores of A and B in position p respectively. In a *zero-sum* game, it is necessary that $score_A(p) + score_B(p) = 0 \; \forall p$. This is equivalent to saying that there is no move that benefits both players simultaneously.

2.1.1 Successes of Game-Tree Search in Game-Playing AI

Computers are able to play board games such as Chess [Baxter et al., 1998, Björnsson and Newborn, 1997], Checkers [Chellapilla and Fogel, 2001a,Schaeffer, 1997], Go [Müller, 2002, Dayan et al., 2001] and Othello [Buro, 1997a, Chong et al., 2003], which are all two-player, perfect information, and zero-sum games. Advances in the playing strength of computers can be largely attributed to the increased computing power available and sophisticated game-tree search techniques, such as Alpha-Beta searching [Knuth and Moore, 1975] and proof number searching [Allis et al., 1994]. In this section, we will outline the research results of each game, and briefly see how selective search is employed to play them effectively.

Chess

Since the early development of computer games research, Chess has been considered the pinnacle of AI research. Intensive research has been done since then and the dominant paradigm used to tackle computer Chess is game-tree search with Alpha-Beta searching.

In 1988, IBM built DEEP THOUGHT, the first Chess machine to beat a chess grandmaster in tournament play. DEEP THOUGHT used game-tree search with Alpha-Beta

searching and had a single-chip Chess move generator that could search in the neighborhood of 500,000 positions per second to 700,000 positions per second.

In May 1997, DEEP BLUE [Björnsson and Newborn, 1997], the descendent of DEEP THOUGHT, beat world champion Garry Kasparov with a score of 3.5-2.5. DEEP BLUE was based on a redesigned evaluation function had over 8,000 features and a new chip that added hardware repetition detection, a number of specialized move generation modes and efficiency improvements. DEEP BLUE is a massively parallel system with over 200 Chess chips, and each chip searches about 2-2.5 million positions per second. By using over 200 of these chips, the overall speed of the program is 200 million positions per second.

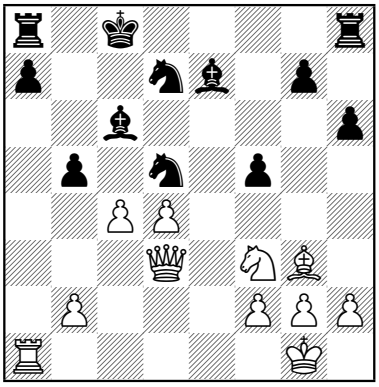


Figure 2.1: Final Position of DEEP BLUE (White) versus Kasparov (Black) game in 1997 where Kasparov loses in 19 moves

Computer Chess has advanced rapidly since then, and modern Chess programs play at grandmaster level even when using personal desktop computers. For example, in October 2002, a man-machine match held in Bahrain between the human world champion Vladimir Kramnik and DEEP FRITZ, a commercial Chess program on a standard computer configuration, finished in a 4-4 draw, with 2 wins each and 4 draws. And in January 2003, a six-game match between Garry Kasparov and another computer program named

DEEP JUNIOR resulted in a 3-3 draw, with a win each and 4 draws. Most recently in a match from 25 November to 5 December 2006, DEEP FRITZ beat World Champion Vladimir Kramnik 4-2, with two wins for the computer and four draws.

While DEEP BLUE was a sophisticated brute-force searcher, modern computer Chess programs for desktop computers are also able to play at grandmaster level partially due to the successful forward pruning techniques such as futility/Razoring (Section 6.1.1) and Null-Move Pruning (Section 6.1.2). Nearly all world-class chess programs apply various forward pruning techniques throughout the search [Heinz, 1999].

Checkers

Checkers, also known as American Checkers, is played on a 8×8 board. Two players, on opposite sides of the board, alternate move pieces diagonally, and pieces of the opponent are captured by jumping over them. The player who has no pieces left or cannot move loses the game.

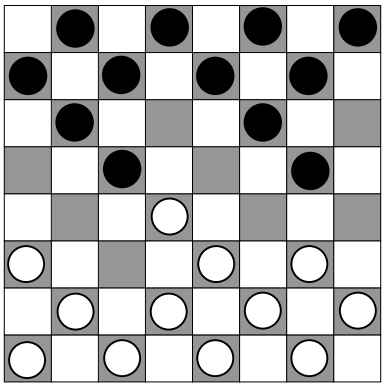


Figure 2.2: The White Doctor Opening which has been shown to be a draw [Schaeffer et al., 2005]

The first intelligent computer Checkers program can be attributed to A. L. Samuel in 1959 when he developed a Checkers program that used reinforcement learning. The

Checkers program had won a game against a strong human in 1959. Interestingly, the win has since been noted to be dubious, as analyses of game records had showed that the human had made several huge blunders uncharacteristic of a strong player. The stigma of Checkers being a 'solved' game had resulted in lack of research being done on the game.

In 1988, Jonathan Schaeffer, and a team at the University of Alberta started developing CHINOOK [Schaeffer, 1997], which defeated the current human world champion in match play. The highlight of CHINOOK was its matches against the previous human world champion, Marion Tinsley, who had been World champion since 1954 and was perceived by many to be invincible in match play. In the 1992 series, Marion Tinsley won 4, lost 2 and drew 33 games against CHINOOK. In the 1994 series, the match was interrupted when Marion Tinsley fell seriously ill and CHINOOK was rescheduled to play the second best human player, Don Lafferty. CHINOOK competed against Don Lafferty in 1994 and won 1, lost 1 and drew 18 games, and in 1995, it won 1 and drew 32 games.

CHINOOK uses traditional AI techniques such as endgame database, Alpha-Beta searching and opening books. CHINOOK could not use the Null-Move Pruning to perform forward pruning as many positions in Checkers are *zugzwang* (defined as positions where the player-to-move benefits more if he or she does not move), for which the Null-Move Pruning is known to be ineffective in (Section 6.1.2 for details). Schaeffer had to therefore spend considerable time implementing hand-crafted heuristics to extend and prune the search tree [Schaeffer et al., 1992].

In 2007, the CHINOOK team had computed the game-theoretic values of all Checkers positions up to 10 piece positions [Schaeffer et al., 2005]. By using these endgame

databases and forward search, Checkers is computationally proven to be a draw [Schaeffer et al., 2007].

Othello

Othello, also known as Reversi, is a strategic two-player board game on a 8×8 board with Black and White pieces. The starting position is shown in Figure 2.3, and by convention, Black makes the first move. Players must place a new piece in a position such that there exists at least one straight line (horizontal, vertical or diagonal) between the new piece and a piece of the player already on the board, with one or more opponent pieces between them. After placing a piece on the board, the player flips all opponent pieces lying on a straight line between the new piece and any other piece of the player already on the board. If a player cannot make a valid move, play passes to the other player. If neither player can move, the game ends. The player with more pieces on the board at the end wins.

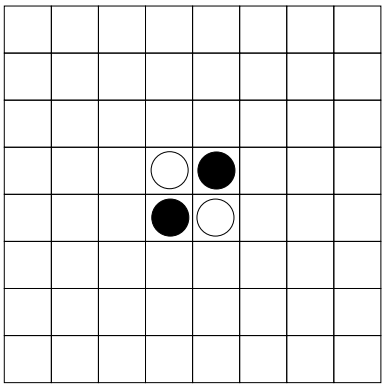


Figure 2.3: Initial Position for Othello

In 1997, LOGISTELLO [Buro, 1997a] defeated Takeshi Murakami, the world Othello champion, by winning all 6 games of the match. LOGISTELLO is able to learn its opening books [Buro, 1997c] and uses a table-based evaluation function, which can

capture more non-linear dependencies than small neural networks based on sigmoid functions [Buro, 1998]. While the move selection process is a commonly-used Alpha-Beta search, LOGISTELLO also incorporates a sophisticated forward pruning technique called ProbCut. ProbCut is based on the idea that the result of a shallow search is a rough estimate of a deeper search, and therefore it is possible to eliminate certain moves during normal search based on a shallow search. ProbCut has been shown to be effective in Othello, Chess, and Shogi [Jiang and Buro, 2003].

Go

Go is a two player Oriental board game that originated between 2500 and 4000 years ago. It is one of the oldest games in the world that is still widely played in Asian countries and is gaining popularity in Western countries. It is also known as Weiqi in China and Baduk in Korea.

Like Chess, Go is a deterministic, perfect information, zero-sum game of strategy between two players. Go is played on a board, which consists of a grid made by the intersection of horizontal and vertical lines. The number of intersections determines the size of the board. Go is normally played on a 19×19 sized board as shown in Figure 2.4. However, smaller board sizes, such as 9×9 and 13×13 sized boards, are also used for playing quicker games. Two players alternate in placing black and white stones on the intersection points of the board (including the edges and corners of the board), with the black player moving first.

The aim of Go is to surround more territory and capture more prisoners than your opponent. Two players alternate placing stones on the intersection points on the board, but unlike Chess, the stones do not move on the board unless they are captured.

The traditional approach of Minimax game-tree search has proven to be difficult to

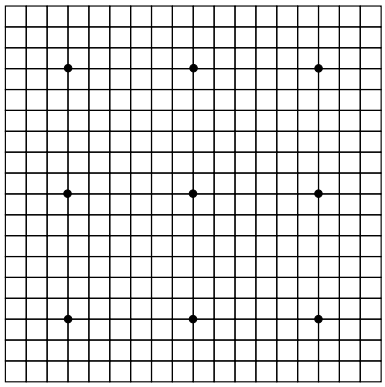


Figure 2.4: Go board, or "goban"

implement in Go due to its high branching factor. Programs which use search trees extensively can only play on smaller boards such as 9×9 as a result. Many programs such as GNU Go[1] therefore resort to using knowledge-based systems such as encoding Go knowledge in patterns and using pattern matching algorithms to choose and evaluate potential moves.

One alternative to using game-tree search is the use of Monte Carlo search techniques [Bouzy, 2003, Bouzy, 2005, Coulom, 2006, Kocsis and Szepesvári, 2006]. These methods generate a list of potential moves, and for each move, many random games are simulated to the endgame where evaluation can be done. The move which gives the best average score for the current player is chosen as the move to play. However, since the moves used for evaluations are generated at random, it is possible for a weak move to appear strong if there are only a few specific enemy counter-move. This problem is usually handled by incorporating a shallow ply search before invoking the Monte Carlo simulations. So while the game-tree is not searched in a Minimax manner, forward pruning remains important even in Monte Carlo tree search as not considering bad moves improves the accuracy (and efficiency) of the search. One example of a strong

[1] Available at `http://www.gnu.org/software/gnugo/gnugo.html`

Go-playing program using UCT, a Monte Carlo search method, is MoGo [Gelly et al., 2006, Gelly and Silver, 2007]. In this thesis, however, we do not consider the application of forward pruning in Monte Carlo search.

The playing level of even the best Go programs [Fotland, 2004] remains modest [Müller, 2002], compared to the successes achieved in other game domains such as Chess and Checkers. Since the playing style of computers is different from humans, it is difficult to make an accurate assessment of the strength of current Go programs. This is especially true as humans can learn the weaknesses of computers after a few games and are able to defeat the programs in subsequent games. As a rough estimate, the best Go programs are ranked about 15 kyu using conventional search techniques [Müller, 2002], and *Dan* level (equivalent to expert player) on 9×9 boards using *Monte Carlo* methods [Gelly and Silver, 2007].

2.1.2 Game-Tree

A turn-based game can be represented as a game-tree, where each node in the tree represents a board position. A game-tree consists of a *root node* representing the current board position, *terminal nodes* that represent the end of a game and *interior nodes* that have a value that is the function of their *child nodes*. Each edge represents one possible move, and moves change the board position from one to another.

The number of branches from each node is defined as the *branching factor*. The *depth* of a game-tree is the maximum length of a path from the *root node* to a *terminal node*. If we assume a game-tree of uniform branching factor b and depth d, the number of nodes in the game-tree is $O(b^d)$. Figure 2.5 shows the game-tree of the starting position of a Tic Tac Toe game up to depth 2.

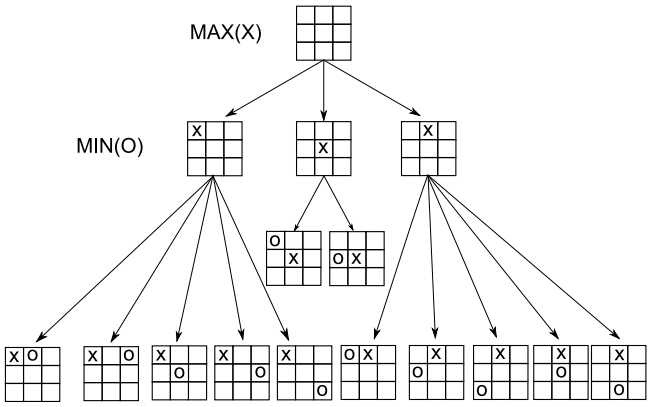


Figure 2.5: Game-Tree of initial Tic Tac Toe board

2.1.3 Minimax and Negamax Search

In mathematical game theory, the *zero-sum* condition leads rational players to act in a *Minimax* fashion. This means that both players will try to maximize their own gains, as this will simultaneously minimize that of their opponents. From the viewpoint of the score of a single player, this is achieved by the player maximizing the score, and his opponent minimizing the score. Minimax search can therefore be implemented by alternating between maximizing and minimizing the score. In a two-player setting, the player maximizing the score is typically called the MAX player, and the player minimizing the score is called the MIN player. The Minimax value of a node u defined mathematically is

$$score_*(u) = \begin{cases} utility(u) & u \text{ is a leaf node,} \\ \max\{score_*(child(u))\} & u \text{ is a Max node,} \\ \min\{score_*(child(u))\} & u \text{ is a Min node.} \end{cases}$$

where $child(u)$ returns the set of child nodes of u.

By using the zero-sum condition of the game, there is an equivalent formulation of Minimax search that simplifies its implementation. Negamax search evaluates each

Pseudocode 1 Minimax($state$, $depth$, $type$)

```
1: if depth == 0 or isTerminal(state) then
2:     return Evaluate(state)
3: if type == MAX then
4:     score ← −∞
5:     for move ← NextMove() do
6:         value ← Minimax(successor(state, move), depth − 1, MIN)
7:         score ← max(value, score)
8: else
9:     score ← ∞
10:    for move ← NextMove() do
11:        value ← Minimax(successor(state, move), depth − 1, MAX)
12:        score ← min(value, score)
13: return score
```

position as a maximizing player by negating the scores of positions resulting from moves in the current position. To see this, we note that by definition, the score of a player is the negation of the score of his or her opponent in any position in a zero-sum game. After making a move in the current position, the opponent is the player to move in the resulting position. This means that rational players will try to maximize scores evaluated by negating the score returned by a move. Negamax search simplifies the implementation of Minimax search as it does not have to discriminate between a MAX or MIN node, since all nodes are MAX nodes within the search.

Pseudocode 2 Negamax($state$, $depth$)

```
1: if depth == 0 or isTerminal(state) then
2:     return Evaluate(state)
3: score ← −∞
4: for move ← NextMove() do
5:     value ← −Negamax(successor(state, move), depth − 1)
6:     score ← max(value, score)
7: return score
```

2.1.4 Alpha-Beta Search

Minimax and Negamax search are exhaustive searches that visit all nodes of a game-tree to find its Minimax score. This can be shown to be non-optimal in many cases where there are nodes visited by the search that do not affect the final Minimax score.

After finding the score of the first move, say x, in a MAX node, a MAX player should only need to be concerned with moves that result in scores greater than x, as he is trying to maximize his score. Consider the situation where MAX makes a second move, and the first child of that MIN node, which we denote m_2, returns a score y, such that $y \leq x$. Since the MIN node is trying to minimize the score, the eventual value of the MIN node is at most y. The MAX parent will therefore never pick m_2 since MAX already has a move that leads to a score of $x > y$. In other words, the MAX node has imposed a lower bound on its MIN children in the above example. Conversely, a MIN node would impose an upper bound on its MAX children. The lower and upper bounds are equivalent to the values of *alpha* (α) and *beta* (β), respectively, in Alpha-Beta search. In other words, the alpha bound is used by MAX nodes to represent the minimum value that MAX is guaranteed to have, while the beta bound is used by MIN to represent the maximum value that MIN is guaranteed to have. The propagation of alpha and beta values [Knuth and Moore, 1975] can be demonstrated using Figure 2.6.

2.1.5 Game-Tree Search Definitions

Principal Variation The *principal variation* is a sequence of moves by players that lead to the Minimax value. If there are multiple sequences of moves that lead to the Minimax value, we can refer to any of them as the principal variation. The principal variation can be easily retrieved from a Minimax or Alpha-Beta search by storing the

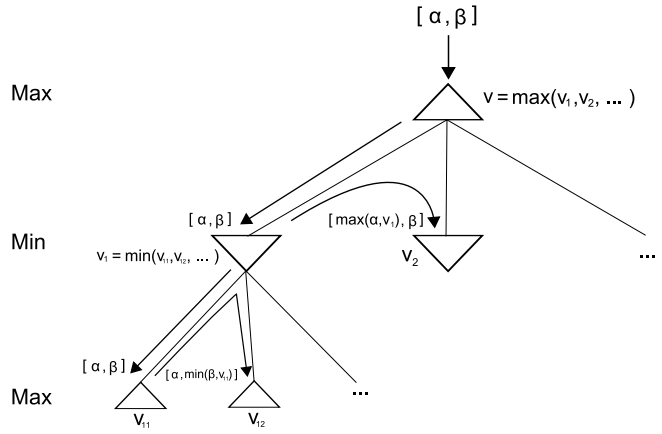


Figure 2.6: How alpha and beta values propagate in Alpha-Beta Search

Pseudocode 3 AlphaBeta($state$, α, β, $depth$)

```
1: if depth == 0 or isTerminal(state) then
2:    return Evaluate(state)
3: score ← −∞
4: for move ← NextMove() do
5:    value ← −AlphaBeta(successor(state, move), −β, −α, depth − 1)
6:    score ← max(value, score)
7:    if score > alpha then
8:       alpha ← score
9:       if score ≥ beta then
10:         return score
11: return score
```

move that returns the best score in a move sequence (breaking ties at random) at each ply.

Minimal Tree If Alpha-Beta search is shown the principal variation first, then we know that Alpha-Beta search is able to eliminate many branches of remaining moves as their scores are guaranteed *not* to affect the Minimax value at the root. It is possible to consider a subset of a game-tree called the *minimal*, also known as *optimal* or *critical*, game-tree that a search algorithm requires to determine the Minimax value of the root. The Minimax value of the root depends only on the node values present in the minimal

tree, and the values of other nodes in the game-tree do not affect the Minimax value at the root.

We are able to obtain a Minimal tree of any given game-tree by the following procedure, shown graphically in Figure 2.7 [Marsland and Popowich, 1985, Reinefeld and Marsland, 1987]:

1. The root node is defined to be a PV node.

2. At a PV node, at least one child has the Minimax value of the root. Define one such child to be a PV node, and the remaining child nodes to be CUT nodes.

3. At a CUT node, at least one child has a Minimax value less than the Minimax value of the principal variation. Define one such child to be an ALL node. All remaining child nodes do not affect the Minimax value of the root.

4. At an ALL node, all child nodes are defined as CUT nodes.

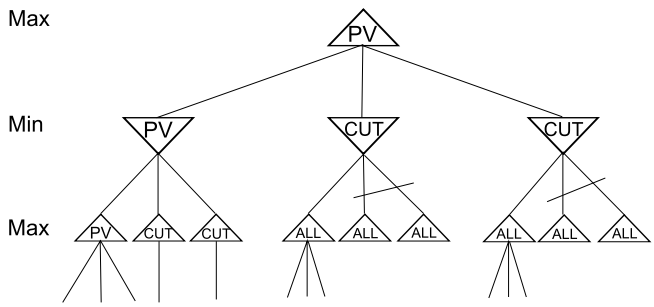


Figure 2.7: Minimal Alpha-Beta Search Tree

The nodes searched in Alpha-Beta search can therefore be categorized into several types; in [Knuth and Moore, 1975], the minimal game-tree is made up of type 1, type 2 and type 3 nodes, but it is more common (and clearer) to refer to these nodes as PV, CUT and ALL nodes [Marsland and Popowich, 1985, Reinefeld and Marsland, 1987].

The best-case time complexity of Alpha-Beta search in the minimal game-tree is $b^{\lceil \frac{d}{2} \rceil} + b^{\lfloor \frac{d}{2} \rfloor} - 1$ [Slagle and Dixon, 1969], where b is the branching factor and d is the depth of the game-tree. This is the minimum number of nodes that must be examined by any search algorithm to determine the Minimax value [Knuth and Moore, 1975]. However, the worst-case time complexity of Alpha-Beta search is the same as that of Minimax search, or b^d.

2.2 Search Enhancements

While Alpha-Beta search is able to prune off branches of the game-tree that provably cannot affect the Minimax value of the root, the time complexity, even in the best case, is still exponential. There has therefore been much research done on improving search performance using other techniques, which we will outline in this section.

2.2.1 Transpositions

In board games, it is possible for different sequences of moves to lead to the same board position. Since the board positions are exactly the same, they should have the same Minimax value. To avoid repeating the search effort needed to re-evaluate the board position, game-playing programs should make use of transposition tables that store the Minimax values of board positions.

The transposition table also stores key features such as the search depth, best move, the score of the search and the search window used. Since the transposition table is typically used within an Alpha-Beta search, it needs to keep track of the bound information of the score, which can be an exact value, a lower bound, or an upper bound. When there is a transposition table hit, it is possible for the cached score to cause a Alpha-Beta

cutoff by failing high or low, or to be able to narrow the search window.

Due to the high performance requirements of game-playing, an incremental hash code of board positions is used to address the hash table. One common technique for creating hash codes in game-playing programs is zobrist hashing [Zobrist, 1969]. The advantages of the zobrist key are that it is simple to implement, incremental and fairly collision resistant. The technique initializes by associating each possible piece (e.g., King, Queen, Bishop, Empty, etc.) in each position on the board with a random value. To create a hash code for a position, the values of each position on board referenced by the piece in that position are XORed together.

Enhanced Transposition Cutoff

Enhanced transposition cutoff [Plaat et al., 1996] is a simple but effective method of improving transposition table use during search - before actually searching any move, check all successor positions and see if they are in the transposition table and can cause a cutoff. If such a position is found, then Alpha-Beta search can immediately return a value and no further search needs to be done.

2.2.2 Alpha-Beta Search Window

The *search window* of an Alpha-Beta search algorithm is defined as the interval between the alpha and beta values. During the search, only moves that result in scores within this window are considered, and all other moves are pruned. If the actual Minimax value of the root is not within the initial search window, Alpha-Beta search will not return the actual Minimax value but will instead *fail high* or *fail low* appropriately. It is possible to guarantee that Alpha-Beta search always returns a correct value by setting the initial search window to $(-\infty, \infty)$.

Using a small search window may seem like a bad idea, since the result might not be exact and it is necessary to re-search with larger search windows to get the correct value. However, searches with small search windows are sped up massively as they are able to prune off more nodes. There are several search enhancements that make use of small search windows to improve search performance.

Aspiration Search

Aspiration search [Slate and Atkin, 1977, Baudet, 1978] works by searching with an initial search window $(v - \Delta, v + \Delta)$, where v is an estimated evaluation of the board position and Δ is a pre-determined range. If the search *fails high*, it is possible to re-search with search window $(v + \Delta, \infty)$ to find the exact score. Similarly, if the search *fails low*, a re-search with search window $(-\infty, v - \Delta)$ will return the exact score.

The estimated evaluation v can be obtained via several means, such as using the evaluation of the previous board position, or when using Iterative-Deepening Search (Section 2.2.3), to use the evaluation of the board position at depth $d - 1$ when at depth d.

The constant Δ should compromise between the time saved from having a smaller search window, and the time it takes to re-search if the true score is not within $(v - \Delta, v + \Delta)$.

Negascout/Principal Variation Search

The *minimal window* is defined when it is the case that $beta = alpha + 1$. Searches with a *minimal window* do not return exact scores, but instead return a bound on the score. There are only two possible cases: (1) the search *fails high* and $score \geq beta = alpha + 1 > alpha$, and (2) the search *fails low* and $score \leq alpha$.

This might seem like futile work as a search with a *minimal window* will never return an exact score. However, we note that after evaluating at least one child node, an Alpha-Beta search algorithm would ideally only need to consider, for any evaluation $score$ of subsequent child nodes, that $score \leq alpha$; this occurs if the best move was the first move searched.

The Negascout/Principal Variation Search (PVS) [Marsland, 1983, Reinefeld, 1983, Reinefeld, 1989] works on this principle and assumes good move ordering is performed on the game-tree. For the first move and PV nodes, the search window is the usual (α, β); for all other moves, the search window is the minimal window $(\alpha, \alpha + 1)$. If the game-tree is a minimal Alpha-Beta game-tree, all searches with the minimal window will *fail low*, and search effort is saved as the minimal search window should have reduced search effect. If a search with the minimal window *fails high*, then a re-search with the usual (α, β) search window is required to get an exact score.

MTD

Memory-enhanced test drivers (MTD) algorithms [Plaat, 1996] use Memory-enhanced test (MT) algorithms to search the Minimax value of a game-tree. MT algorithms implement an efficient transposition table to act as the algorithm's memory. MT algorithms are essentially minimal window Alpha-Beta searches that use transposition tables to avoid duplicate work. The use of transposition tables allows the algorithm to narrow the search to look at the most promising moves first, while using the Minimax paradigm to search.

A variant called MTD(f) is a strong Minimax search algorithm that performed better, on average, than NegaScout/PVS in tests with Chess, Checkers and Othello [Plaat, 1996]. MTD(f) is efficient due to the use of minimal-window searches. Typically,

MTD(f) is called within an iterative deepening framework, and the value of the previous iteration can be used as the initial estimate to MTD(f).

2.2.3 Iterative-Deepening Search

The basic Minimax search previously explained is a *depth-limited depth-first search*, that is, it searches the state-space in a *depth-first* manner until at most a fixed depth. This presents a problem with game-playing programs, as they require real-time decision-making during game-play; if a search with a high depth limit does not return a result within a time limit, the program is unable to make a move. Iterative-Deepening Search (IDS) is a search strategy that repeatedly calls a depth-limited search with increasing depth limit d.

By applying IDS to the root node, a game-playing program is able to respond as soon as a search to the base depth returns a result, and yet can be allowed to search as deeply as possible until a time limit is reached; this is desirable as research [Junghanns et al., 1997, Thompson, 1982] empirically demonstrates a positive correlation between increasing search depth and game-playing performance.

There is another advantage to using IDS in game-tree search, as information from searches at lower depths can be used to improve search performance at higher search depths; examples include the History Heuristic and Killer Heuristic that improve move ordering, and using search evaluations of moves from lower search depths as estimates in aspiration search.

The time complexity of an IDS that calls a depth-first search from depth $0, 1, \ldots, d$ is surprisingly the same as a depth-first search to depth d – $O(b^{d+1})$, for a game-tree of uniform branching factor b and depth d. This is due to the dominating exponential growth of leaf nodes. Perhaps even more surprising, it is typically the case that the number of

nodes searched by all iterations of an IDS is smaller than the number of nodes searched by a single depth-first search to the full depth in modern Chess programs [Heinz, 2000] due to the performance gains that Alpha-Beta search window algorithms such as aspiration search and PVS give.

2.2.4 Move Ordering

Good move ordering is essential to achieving more cutoffs when using Alpha-Beta search [Knuth and Moore, 1975]. Variants of Alpha-Beta search like PVS and MTD(f) also depend on good move ordering for their efficiency. A slightly better move ordering can improve search performance by $50\% - 100\%$ and above [Heinz, 2000] in a game-playing program. This high dependency on move ordering for good search performance has resulted in the development of *dynamic* and *static move ordering heuristics* that allow modern Chess programs to search game-trees that have only $20\% - 30\%$ more nodes [Heinz, 2000] than the minimal Alpha-Beta game-tree. Move ordering heuristics are therefore usually used in practical implementations.

Dynamic Move-Ordering

Dynamic move ordering heuristics collect information about moves during search and use that information to better re-order subsequent moves. Examples of dynamic move ordering heuristics are the History Heuristic [Schaeffer, 1989] and the Killer Heuristic [Akl and Newborn, 1977]. Both the History Heuristic and the Killer Heuristic are *domain-independent* techniques that work by storing information when a move has worked well in previous board positions.

In Chess, the History Heuristic maintains a table for all possible *from* squares and *to* squares for each piece. Whenever a move is considered to have worked well at depth

d, such as causing a beta cutoff, the value in the table referenced by that move is incremented by the value of a function that grows quickly with respect to d (e.g., d^2 or 2^d). The function is designed to reward moves that worked well in deep searches more than moves that worked well in shallow searches. Moves are then ordered by sorting them in descending order of history values. The Killer Heuristic also rewards moves that worked well in previous instances by maintaining a list of killer moves for each ply of the search. Killer moves are determined by counters that track the number of times a move has worked well in that ply, and are sorted and replaced using the values of the counters.

Static Move-Ordering

Static move ordering heuristics do not depend on information of previous searches, but there are several domain-independent techniques that work well in games such as *MVV/LVA*, *SEE* and SOMA [Levy and Newborn, 1991] (and its variant SUPER-SOMA [Rollason, 2000]).

MVV/LVA stands for "Most Valuable Victim/Least Valuable Attacker" and it works in games that have captures of pieces that have different material value. MVV/LVA orders the best capture first as it assumes that the best capture will be the one that captures the piece with the highest material value. If there are ties, the heuristic breaks ties by capturing with the player's own piece of the lowest material value.

"Static Exchange Evaluation" (SEE) is also a common technique that resolves all possible captures and re-captures from a position in all permutations, and sorting the moves based on the material won. It is more sophisticated than MVV/LVA, but it is more accurate and can be used to prune off likely bad captures.

SUPER-SOMA [Rollason, 2000] is an extension of the SOMA algorithm [Levy and

Newborn, 1991] that analyzes capture sequences. The SOMA algorithm is more sophisticated than SEE and takes other static tactical features such as pins, forks and mate threats into account. While SOMA considers each attacked square at a time, SUPER-SOMA extends the capture analysis to the whole board. SUPER-SOMA is considerably slower than the other techniques and seems to have been successful in Shogi [Rollason, 2000].

2.2.5 Search Extensions

Instead of searching the game-tree to a fixed depth limit, it is possible to explore some parts of the game-tree more deeply, while other parts of the tree are terminated early. Humans do this intuitively by only searching moves that are deemed interesting and ignoring moves that are deemed useless. We will outline the search extension techniques used in programs.

Fractional-Ply Extensions

Fractional-play extensions [Levy et al., 1989, Rijswijck, 2000, Björnsson and Marsland, 2000] is a search-extension scheme that allocates different weights (in terms of ply) to different classes of moves. These weights can be fractional. For example, in Chess, move classes can be checking moves and recaptures. During the search, each move is categorized under one of the move classes, and the depth of the current move path is the sum of the values of plies along the path. As the value of a ply can be less than (or more than) 1.0, this allows certain move sequences to be searched deeper (or shallower).

Singular Extensions

Singular extensions [Anantharaman et al., 1990] was implemented in DEEP THOUGHT (the predecessor to DEEP BLUE) and it improved DEEP THOUGHT's playing strength by about 30 USCF rating points [Anantharaman, 1990]. Singular extensions works by extending the search depth of *singular moves*, defined to be moves that return an evaluation higher than all of its siblings by a pre-determined margin, also known as the *singular margin*. If a move causes a fail high, the algorithm will check all its siblings by searching them to a reduced depth. The move is also deemed singular if the evaluations of all its siblings are lower by at least the singular margin.

Quiescence Search

Game-tree search algorithms are subject to the *horizon effect* - this occurs when a losing board position is beyond the search depth of the search, and the evaluation function is not sophiscated enough to recognize the losing move sequence at the leaf nodes. The game-tree search proceeds by making a losing move, but returns a good evaluation. A few moves later, the game-tree search is able to search to the losing board positions and the evaluation returns the correct assessment that the current board position is losing. Unfortunately, at this point, the game-playing program is unable to avoid a loss. This is akin to the game-playing program walking down a plank towards the sharks while blindfolded and unaware of the immediate danger it is in until it is too late.

Quiescence search mitigates the horizon effect by continuing the search even when the given depth limit is reached until a "quiet" or more stable position is encountered. In Chess, this would commonly refer to positions where there are no possible captures and no player is in check. Quiescence search is therefore able to give a better estimate of the board position, especially when using a poor evaluation function that is unable to

accurately evaluate boards that are “unstable”.

2.3 Chapter Conclusions

This chapter introduced the various search techniques of game-tree search within the Minimax framework. We also described several search enhancement techniques such as move-reordering and search extensions. In the next few chapters, we will expand on the methods presented here to solve games (Chapter 5) and to perform forward pruning (Chapter 6).

Chapter 3

Solving Tigers and Goats

This chapter has portions of text extracted from our article "Computing Tigers and Goats" published in the *International Computer Games Association Journal* 27(3), pages 131-141, 2004.

In this chapter, we first introduce how games are classified and solved. We then introduce the game of Tigers and Goats before computing the state-space complexity and game-tree complexity of the game. We also solved the endgame of Tigers and Goats by retrograde analysis and present statistics on the endgame database. Finally, we discuss the complexity of solving Tigers and Goats.

3.1 Solving Games

Compared to research in real-time game-playing programs, there has been relatively less work on solving games. We will outline some of the main results of solving games in this section, but interested readers should refer to the excellent overview of solved games by Van den Herik et al. [van den Herik et al., 2002].

3.1.1 Levels of Solving Games

There are several levels [Allis, 1994] of solving games:

1. Ultra-weakly solved. Games are ultra-weakly solved if the game-theoretic value from the initial position is known. Note that this does not mean that the actual moves to achieve the game-theoretic value are known. For example, Hex can be shown by the "strategy-stealing" argument to be a first-player win, but no actual winning strategy is currently known.

2. Weakly Solved. Weakly solved games mean that a strategy is known to achieve the game-theoretic value of the game from the initial position. Most of the research done has been at this level, such as solving Nine Men's Morris [Gasser, 1996], Connect-Four [Allen, 1989], Go-moku [Allis, 1994] and Awari [Romein and Bal, 2003].

3. Strongly solved. Games are strongly solved if for all possible board positions, a strategy is known to achieve the game-theoretic value of that board position. This is the most computationally expensive level of solving a game, but it is also the most useful for research and recreational purposes, as it allows a computer to show the exact moves needed to achieve a specific result. For example, Awari is strongly solved [Romein and Bal, 2003] as Romein and Bal computed several databases that can used together to select the best move from any legal board position.

3.1.2 Classification of Games

Classification by Complexity

In this section, we describe the complexity of games by using two different measures [Allis, 1994], *state-space complexity* and *game-tree complexity*. It is useful to first define the *solution depth* and *solution search tree* of a node.

The solution depth of a node J is the minimal depth (in plies) of a full-width search that can determine the game-theoretic value of J. The solution search tree of a node J is the full-width search tree with the same depth as the solution depth of J.

The state-space complexity of a game is the number of different legal positions in the solution search tree of the initial position(s) of the game. Note that the same board positions encountered when searching different branches of the game-tree are counted once. For example, assume that a Chess position J with White to move has 10 legal moves and for each legal move, Black has 5 legal moves, of which 1 mates White. Furthermore, all moves lead to distinct positions. The solution search tree of J is the full-width tree consisting of J, the 10 children of J, and the 50 grandchildren of J. Subsequently, the state-space complexity of J is $1 + 10 + 50 = 61$.

The game-tree complexity of a game is the number of leaf nodes in the solution search tree of the initial position(s) of the game. Note that two same board positions encountered when searching different branches of the game-tree are counted twice. For example, assume that a Chess position J with White to move has 10 legal moves and for each legal move, Black has 5 legal moves, of which 1 mates White. The solution search tree of J is the full-width tree consisting of J, the 10 children of J, and the 50 grandchildren of J. Subsequently, the game-tree complexity of J is 50. The game-tree complexity can be estimated using Monte Carlo simulations [Knuth, 1975].

Classification by game-type

Van den Herik et al. [van den Herik et al., 2002] introduced the concept of *convergence* to describe games that have a state-space that decreases in size as the game progresses. If the number of states increases as the game progresses, the game is said to be *divergent*. Examples of convergence games are Nine Men's Morris, Mancala games, Checkers, and Tigers and Goats; examples of divergent games are Connect-Four, Go-Moku, Othello and Go.

3.1.3 Game Solving Techniques

Search Techniques

The most basic method to solve games is to use an *exhaustive search* method like Alpha-Beta search and its variants. Unfortunately, for all but the simplest games like Tic Tac Toe, this method is infeasible to the exponential increase in computational effort needed to search through the state-space. To reduce the search complexity, it is beneficial to use domain-independent techniques that exploit common features of games. These are termed as *knowledge-based* methods [van den Herik et al., 2002]. Examples of knowledge-based methods are threat-space search [Allis, 1994, Thomsen, 2000, Cazenave, 2001], proof-number search [Allis et al., 1994, Allis, 1994, Seo et al., 2001], lambda search [Thomsen, 2000] and pattern search [Rijswijck, 2000].

Retrograde Analysis

In general, convergent games can be solved using retrograde analysis by computing the endgame database for positions of which the state-space complexity is sufficiently small to enumerate. This relies on the property of decreasing state-space complexity of

convergent games.

Retrograde analysis was pioneered in Chess, and has been successfully used in Checkers, Nine Men's Morris and Awari. Whenever a forward search reaches a database position, an exact game-theoretic value of the position can be returned immediately.

The first step in retrograde analysis is to initialize all easily recognizable terminal positions as wins or losses, and all remaining positions as a draw. An iterative process can then correctly determine the correct value of all remaining positions by the following logic - if a position is lost for the player to move, all predecessors of the position are wins for the opponent. Similarly, if a position is won for the player to move, all predecessors are potential losses for the opponent. They are only true losses if all of their successors are also won for the player to move. See Appendix C for more details on retrograde analysis.

Transposition Tables

The use of transposition tables is part of the trade-off between memory usage and computational effort needed in game solving. By storing results in a transposition table, it is possible to avoid recomputing board positions that have already been searched before. However, a forward search proof has to address the so-called *Graph History Interaction* (GHI) problem, which introduces errors in games that contain repeated positions. GHI problems occur when a search algorithm uses a cached search result that depends on the move history. For example, in Chess and Checkers, repeated positions are scored as draws. If the cached result is reached using a different move sequence, it might be possible that the result is no longer correct.

Because no cycles occur in the placement phase of Tigers and Goats, the GHI problem does not affect the forward search proofs presented in our solution for Tigers and

Goats. However, if the GHI problem can indeed occur in a game, forward search proofs can use the techniques described in [Kishimoto and Müller, 2004] to avoid the GHI problem, or simply not store any position that has a value influenced by the move sequence of the search in the transposition table.

3.1.4 Solved Games

Connection Games

In this section, we will discuss connection games. The aim of players in connection games is to form a straight line of k pieces on the board. A simple example of a connection game is Tic Tac Toe, which can be shown to be a draw by exhaustively searching the game-tree. Connection games typically have very high state-space complexity and are divergent games, and therefore retrograde analysis is infeasible. Instead, knowledge-based search techniques [Allis, 1994, Thomsen, 2000] have proven to be highly successful in these types of games.

Connect-Four Connect-Four is a four-in-a-row connection game played on a vertically-placed board, with the standard board with six rows and seven columns. Players take turns dropping discs onto the board, and the winner is the first player to get four of his or her own discs in a line.

The game was shown to be a first-player win using brute-force techniques by Allen [Allen, 1989], and later using knowledge-based search techniques by Allis [Allis, 1994]. Allen's program used standard techniques such as Alpha-Beta search, transposition tables and killer-move heuristics, whereas Allis' program combined a set of strategic rules that identified potential threats by the opponent with Conspiracy-Numbers Search [McAllester, 1988, Schaeffer, 1990]. The first player forces a win by starting in the middle column;

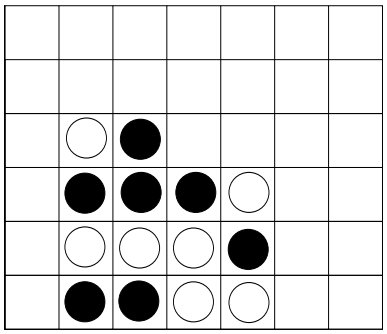


Figure 3.1: Example Connect-Four Game - White to move; Black wins

starting in any other column allows the second player to draw or even force a win.

Go-Moku/Renju Go-Moku is traditionally played with Go pieces on a Go board, where players alternate placing a stone of their color on an intersection. The winner is the first player to place a row of five stones horizontally, vertically, or diagonally. It can be thought of as a generalization of Tic Tac Toe. There are several variations of the game:

1. **Free-style Go-Moku** is played on a full 19×19 Go board or a 15×15 Go board .

2. **Standard Go-Moku** requires a row of exactly five stones to win; rows of six or more, also known as *overlines*, do not count as a win.

3. **Renju** is played on a 15×15 board, and Black is not allowed to (1) form overlines, (2) simultaneously form two rows of three stones that are not blocked by White's stone at either end, and (3) simultaneously form two rows of four stones. There are also special rules for the opening[1].

[1]Interested users can refer to *Renju International Federation: The International Rules of Renju* (`http://www.renju.nu/rifrules.htm`) for more details.

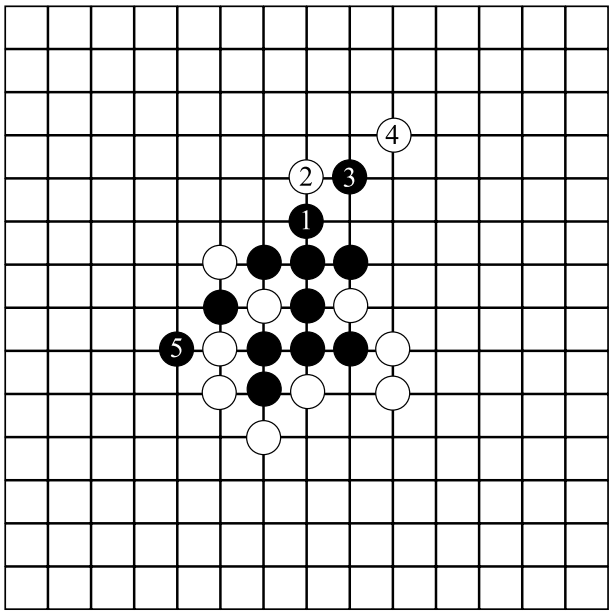


Figure 3.2: Example Free-style Go-moku Game where Black wins by a sequence of forced threats

The game has been shown to be a first-player win by Allis using proof number search [Allis, 1994], which is a best-first AND/OR search. Free-style Go-Moku can be won by the first play in 18 moves against the optimal defense.

Renju, which is played professionally in Japan, has been shown to be a first-player win without opening rules [Wágner and Virág, 2001] using dependency-based search [Allis, 1994], transposition tables, and expert knowledge for moves which are non-threats. By using iterative-deepening search, the program was able to find threat sequences up to 17 plies, which is sufficient to find the game-theoretic value of Renju.

k-in-a-row Games It is interesting to note that Tic Tac Toe, Connect-Four, Go-Moku and Renju can be generalized into k-in-a-row games, where the goal is to obtain a straight line of pieces on board. To be more precise, in $Connect(m, n, k, p, q)$ games, two players alternate placing p stones on a $m \times n$ board except that the first player

places q stones for the first move [Wu and Huang, 2005]. For example, Tic Tac Toe is Connect(3,3,3,1,1), and free-style Go-Moku is Connect(15,15,5,1,1). Threat-based searches perform very well in Connect(m,n,k,p,q) games, and many game-theoretic values of such games have been published [Uiterwijk and van den Herik, 2000, van den Herik et al., 2002, Wu and Huang, 2005].

Mancala

Mancala is a family of board games that are generally played on a board with holes that have pieces known as *seeds* in them. Players move by picking up all the seeds in a hole, then *sowing* the seeds by placing one seed at each subsequent hole from the initial hole until all seeds have been sowed. Capturing of seeds occur based on the state of the board and the rules vary widely between games.

Kalah Kalah is a game in the Mancala family and most variants of this game have been solved and shown to be a first-player win in most cases [Irving et al., 2000]. Endgame databases for Kalah were first built using retrograde analysis. By using iterative-deepening MTD(f), the program is able to solve several starting configurations of Kalah up to six holes and five counters per hole. Search enhancement techniques such as move ordering, transposition tables, Futility Pruning, and enhanced transposition cut-off were also used.

Awari Awari is also a variant of the Mancala family of games that allows 'grand slams' to end games. This allowed Bal and Romein [Romein and Bal, 2003] to use retrograde analysis to create endgame databases. The scores for 889,063,398,406 positions were determined by parallel retrograde analysis. The endgame databases store the scores

rather than best moves, and therefore some forward search is required [Lincke, 2002] to find the correct move to make. By combining this huge endgame databases with minimal forward search, they showed that the game-theoretic value of Awari is a draw. Furthermore, as the endgame databases contain all positions that can occur in a game, it is possible to obtain the game-theoretical score of every position given reasonable resource constraints. This means that the game of Awari is strongly solved.

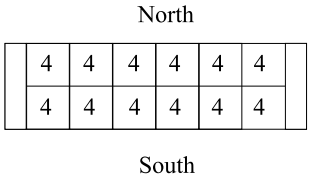


Figure 3.3: Initial Board Position of Awari

Nine Men's Morris

Nine Men's Morris is a two-player game that is played on a board with 24 intersections consisting of interlocking squares. Each player has nine pieces and the objective of the game is to remove all enemy pieces.

The game starts with an empty board and players alternate placing pieces on any empty intersection. After all pieces have been placed by both players, players take turns moving the pieces along one of the board lines to an adjacent intersection. If a move by a player results in a line of three, also known as a *mill*, on the board, the player can remove one enemy piece that is not part of a mill. An excellent position for a player is to be able to move one piece between two mills, thereby removing an opponent piece at every turn, as shown in Figure 3.4.

The game has been shown by Gasser [Gasser, 1996] to be a draw for both players under best play. This was done by performing a forward search from the starting

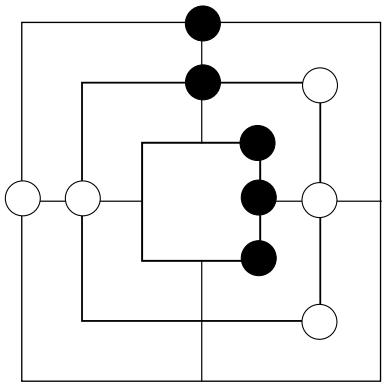


Figure 3.4: Example Nine Men's Morris Game - White to move; Black wins

board position and terminating at the endgame databases. Gasser first built the endgame databases by using retrograde analysis. After removing unreachable positions and symmetrical positions, the total number of distinct positions in the endgame database is 7,673,759,269. A 18-ply forward search was then started from the starting position using a subset of databases and Gasser was able to show that both players have at least a draw, therefore establishing that the game is a draw under optimal play.

Checkers Endgame

A team headed by Schaeffer has been working on Checkers endgame databases, and have computed the game-theoretic values of all Checkers positions up to 10 piece positions [Schaeffer et al., 2005]. These endgame databases play a vital role in finding the game-theoretic value of Checkers as the forward searches are able to terminate and return an evaluation when an endgame position in the database is seen. For example, the opening *White Doctor* has been shown to be a draw [Schaeffer et al., 2005] using a hybrid combination of best-first and depth-first searches with the endgame databases. In solving the traditional game, the researchers have also solved 21 of the 156 three-move openings. By using these endgame databases and forward search, Checkers is weakly

solved [Schaeffer et al., 2007] (although it has elements of being strongly-solved due to storing the proof-tree of the solution), and is shown to be a draw from the initial starting position.

3.1.5 Partially Solved Games

Chess Endgame

A Chess endgame database contains all possible endgame positions with small groups of material, and their game-theoretic value of win, lose or draw. This allows a forward search in a computer Chess program to return a game-theoretical result without searching further. In addition, Chess endgame databases have been used to verify or change historical endgame analysis by humans chess experts.

The first Chess databases were the four and five piece Chess endgame databases [Thompson, 1986] computed by Thompson using retrograde analysis in 1986. As of 2006, all endgame positions with 6 or fewer pieces (including the two kings) have been completely solved [Bourzutschky et al., 2005].

3.2 Introduction to Tigers and Goats

Bagha Chal, or "Moving Tiger", is an ancient Nepali board game, which has recently attracted attention among game fans under the name Tigers and Goats. This game between two opponents, whom we call "Tiger" and "Goat", is similar in concept to a number of other asymmetric games played around the world - asymmetric in the sense that the opponents fight with weapons of different characteristics, a feature whose entertainment value has been known since the days of Roman gladiator combat.

On the small, crowded board of 5×5 grid points shown in Figure 3.5, four tigers face up to 20 goats. A goat that strays away from the safety of the herd and ventures next to a tiger gets eaten, and the goats lose if too many of them get swallowed up. A tiger that gets trapped by a herd of goats is immobilized, and the tigers lose if none of them can move. Various games share the characteristic that a multitude of weak pieces tries to corner a few stronger pieces, such as "Fox and Geese" in various versions, as described in "Winning ways" [Berlekamp et al., 2001] and other sources.

The rules of Tigers and Goats are simple. The game starts with the four tigers placed on the four corner spots (grid points), followed by alternating moves with Goat to play first. In a *placement phase*, which lasts 39 plies, Goat drops his 20 goats, one on each move, on any empty spot. Tiger moves one of his tigers according to either of the following two rules:

- a tiger can slide from his current spot to any empty spot that is adjacent and connected by a line, or
- a tiger may jump in a straight line over any single adjacent goat, thereby killing the goat (removing it from the board), provided the landing spot beyond the goat is empty.

If Tiger has no legal move, he loses the game; if a certain number of goats have been killed (typically five), Goat loses.

These rules are illustrated in the Figure 3.2, which also show that Goat loses a goat within 10 plies unless his first move is on the center spot of a border. Goat has five distinct first moves (ignoring symmetric variants). All but the first move shown in Figure 3.5 lead to the capture of a goat within at most 10 plies, as these two forcing sequences show. At the right of Figure 3.2, Tiger's last move eight sets up a double attack against

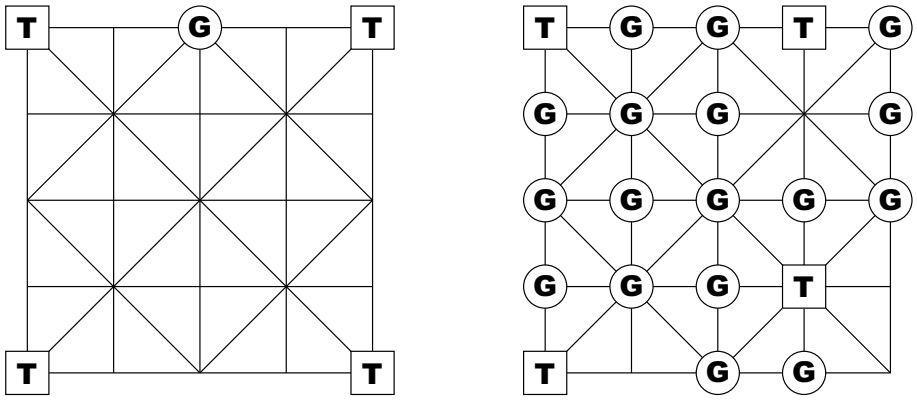


Figure 3.5: Left: the position after the only (modulo symmetry) first Goat move that avoids an early capture of a goat. Right: puzzle with Goat to win in 5 plies if Tiger captures a goat

the two goats labeled 1 and 3.

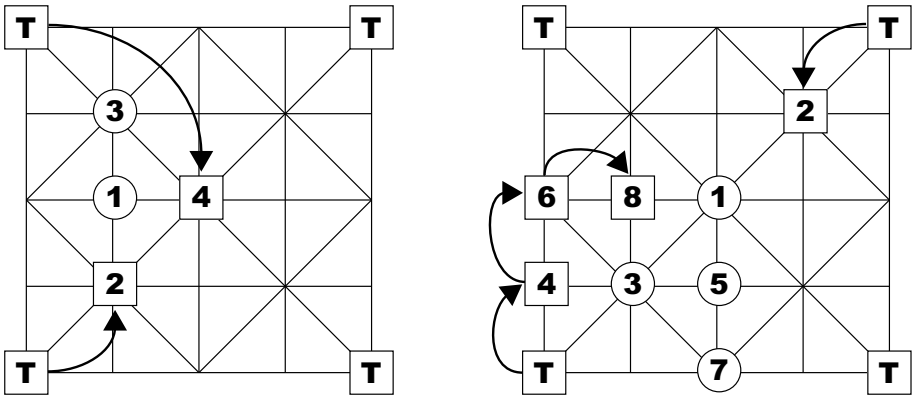


Figure 3.6: Two of the five initial Goat moves that lead to the capture of a goat

The 39-ply placement phase is followed by the *sliding phase* that can last forever. Whereas the legal Tiger moves remain the same, the Goat rule changes: on his turn to play, Goat must slide any of his surviving goats to an adjacent empty spot connected by a line. If there are 17 or fewer goats on the board, four tigers cannot block all of them and such a move always exists. In some exceptional cases (which arise only if Goat cooperates with Tiger) with 18 or more goats, the four tigers can surround and block off a corner and prevent any goat moves. Since Goat has no legal moves, he loses the game.

Although various web pages that describe Tigers and Goats offer some advice on how to play the game, we have found no expert know-how about strategy and tactics. Plausible rules of thumb about good and bad play include the following. First, it is obvious that the goats have to hug the border during the placement phase - any goat that strays into the center will either get eaten or cause the demise of some other goat. Goat's strategy sounds simple: first populate the borders, and when at full strength, try to advance in unbroken formation, in the hope of suffocating the tigers. Unfortunately, this recipe is simpler to state than to execute. In contrast, we have found no active Tiger strategy. It appears that the tigers cannot do much better than to wait, "doing nothing", i.e. moving back and forth, until near the end of the placement phase. Their goal is to stay far apart from each other, for two reasons: one, in order to probe the full length of the goats' front line for gaps; two, so as to make it hard for the goats to immobilize all the four tigers at the same time. Tiger's big chance comes during the sliding phase, when the compulsion to move causes some goat to step forward and offers Tiger a forcing sequence that leads to capture. Thus, it seems that Tiger's play is all tactics, illustrating Chess grandmaster Tartakover's famous pronouncement: "Tactics is what you do when there is something to do. Strategy is what you do when there is nothing to do".

3.3 Analysis of Tigers and Goats

This section discusses the size and structure of the two relevant state-spaces, corresponding to the placement phase and the sliding phase. We compute their sizes using Polya's theory of counting (See Appendix B). We also describe the sliding phase data bases and the placement phase forward searches respectively, with statistical summaries.

3.3.1 Size and Structure of the State Space

The first objective when attacking any search problem is to learn as much as possible about the size and structure of the state-space in which the search takes place. For Tigers and Goats it is convenient to partition this space into 6 subspaces:

S_0: all the positions that can occur during the placement phase,
including 4 tigers and 1 to 20 goats.

S_k: for $k = 1 \ldots 5$, all the positions that can occur during the sliding phase,
with 4 tigers, $21 - k$ goats, and k empty spots on the board.

Notice that any position in any S_1 to S_5 visually looks exactly like some position in S_0, yet the two are different positions: in S_0, the legal Goat moves are to drop a goat onto an empty spot, whereas in S_1 to S_5, the legal moves are to slide one of the goats already on the board. For each subspace S_1 to S_5, a position is determined by the placement of pieces on the board, which we call the board image, and by the player whose turn it is to move. Thus, the number of positions in S_1 to S_5 is twice the number of distinct board images.

For S_0, however, counting positions is more difficult, since the same board image can arise from several different positions, depending on how many goats have been captured. As an example consider an arbitrary board image in S_5, i.e. with 16 goats and 5 empty spots. This same board image could have arisen, as an element of S_0, from 10 different positions, in which 0, 1, 2, 3, or 4 goats have been captured, and in each case, it is either Tiger's or Goat's turn to move. Although for board images in S_1 through S_4 the multiplier is less than 10, these small subspaces do not diminish the average multiplier by much. Thus, we estimate that the number of positions in S_0 is close to 10 times the number of board images in S_0, which amounts to about 33 billion.

Since the game board has all the symmetries of a square that can be rotated and flipped, many board positions have symmetric "siblings" that behave identically for all game purposes. Thus, all the spaces S_0 to S_5 can be reduced in size by roughly a factor of 8, so as to contain only positions that are pairwise inequivalent. Using Polya's counting theory [Polya, 1937] we computed the exact size of the symmetry-reduced state-spaces S_1 to S_5, and of the board images of S_0, as shown in Table 3.1.

	# of board images	# of positions
S_0	3,316,529,500	∼33,000,000,000
S_1	33,481	66,962
S_2	333,175	666,350
S_3	2,105,695	4,211,390
S_4	9,469,965	18,939,930
S_5	32,188,170	64,376,340

Table 3.1: Number of distinct board images and positions for corresponding subspaces

S_0 is very much larger than all of S_1 to S_5 together, and has a more complex structure. Due to captures during the placement phase, play in S_0 can proceed back and forth between more or fewer goats on the board, whereas play in the sliding phase proceeds monotonically from S_k to S_{k+1}. These two facts suggest that the subspaces are analyzed differently: S_1 to S_5 are analyzed exhaustively using retrograde analysis, whereas S_0 is probed selectively using forward search [Gasser, 1996].

3.3.2 Database and Statistics for the Sliding Phase

We determined the game-theoretic value of each of the 88,260,972 positions in the spaces S_1 to S_5, i.e. during the sliding phase, using retrograde analysis. A Tiger win is defined as the capture of five goats, a Tiger loss is defined as the inability to move, and a draw (by repetition) is defined as a position where no opponent can force a win, and each can avoid a loss. Table 3.2 shows the distribution of won, drawn and lost positions.

More detailed statistics of the distribution of won, drawn and lost positions can be found in Appendix A.

Num Goats Captured	Goat to Move Wins	Goat to Move Draws	Goat to Move Losses	Total	Tiger to Move Wins	Tiger to Move Draws	Tiger to Move Losses	Total
4	913,153 (2.8%)	8,045,787 (25.0%)	23,229,230 (72.2%)	32,188,170 (100%)	30,469,634 (94.7%)	1,569,409 (4.9%)	149,127 (0.5%)	32,188,170 (100%)
3	1,315,111 (13.9%)	6,226,358 (65.7%)	1,928,496 (20.4%)	9,469,965 (100%)	6,260,219 (66.1%)	2,918,104 (30.8%)	291,642 (3.1%)	9,469,965 (100%)
2	882,523 (41.9%)	1,199,231 (57.0%)	23,941 (1.1%)	2,105,695 (100%)	465,721 (22.1%)	1,353,969 (64.3%)	286,005 (13.6%)	2,105,695 (100%)
1	252,381 (75.8%)	80,706 (24.2%)	88 (0.03%)	333,175 (100%)	6,452 (1.9%)	197,537 (59.3%)	129,186 (38.8%)	333,175 (100%)
0	30,609 (91.4%)	2,812 (8.4%)	60 (0.2%)	33,481 (100%)	146 (0.4%)	9,468 (28.3%)	23,867 (71.3%)	33,481 (100%)

Table 3.2: Tigers and Goats Endgame Database Statistics

In order to verify their correctness, the endgame databases S_1 to S_5 of the sliding phase were computed twice, independently, using different software and hardware. The first computation, based on the algorithm of [Gasser, 1996], took approximately a week on a Pentium 4 personal computer. The second, based on the retrograde algorithm [Wu and Beal, 2002], took 4 days on a PowerPC G4 Apple computer. See Appendix C for more details on implementing retrograde analysis for Tigers and Goats.

3.3.3 Game-Tree Complexity

The search space S_0, with approximately 33 billion positions, is too large for a static data structure that stores each position exactly once. Hence it is generated on the fly, with portions of it stored in hash tables. As a consequence, the same position may be generated and analyzed repeatedly. A worst case measure of the work thus generated is called game-tree complexity. The size of the full search tree can be estimated by a Monte Carlo technique as described by [Knuth, 1975]. For each of a number of random paths from the root to a leaf, we evaluate the quantity $F = 1 + f_1 + f_1 \times f_2 + f_1 \times f_2 \times f_3 + \ldots$,

where f_j is the fan out, i.e. the number of children, of the node at level j encountered along this path. The average of these values F, taken over the random paths sampled, is the expected number of nodes in the full search tree. Table 3.3 lists the estimated game-tree complexity (after the removal of symmetric positions) of five different "games", where the game ends by capturing one to five goats during the placement phase. These estimates are based on 100,000 path samples.

Goats captured	Estimated Tree Complexity
1	1.28×10^{24}
2	4.23×10^{36}
3	8.92×10^{38}
4	3.09×10^{40}
5	4.88×10^{41}

Table 3.3: Estimated Tree Complexity for various winning criterions

3.4 Complexity of Solving Tigers and Goats

Table 3.4 is reproduced from [van den Herik et al., 2002] and compares the state-space complexity and game-tree complexity of several games together with the placement and sliding phases of Tigers and Goats (discussed in Chapter 3). One conclusion reached by [van den Herik et al., 2002] is that the determining factors in solving a game is either low state-space complexity or low game-tree complexity. In general, brute-force methods are used to solve games like Nine Men's Morris, Kalah and Awari with relatively low state-space complexity. Knowledge-based methods are used to solve games like Go-Moku and Renju with relatively low game-tree complexity. Finally, games like Connect-Four and Qubic with a low state-space complexity and a low game-tree complexity are solved by both methods.

As we have solved the sliding phase of Tigers and Goats by retrograde analysis, we

Game	State-space complexity	Game-tree complexity
Pentominoes	10^{12}	10^{18}
Kalah(6,4)	10^{13}	10^{18}
Connect-Four	10^{14}	10^{21}
Domineering (8×8)	10^{15}	10^{27}
Checkers	10^{21}	10^{31}
Awari	10^{12}	10^{32}
Dakon-6	10^{15}	10^{33}
Qubic	10^{30}	10^{34}
Placement phase of Tigers and Goats	10^{10}	10^{41}
Nine Men's Morris	10^{10}	10^{50}
Draughts	10^{30}	10^{54}
Othello	10^{28}	10^{58}
Go-Moku (15×15)	10^{105}	10^{70}
Renju (15×15)	10^{105}	10^{70}
Hex (11×11)	10^{57}	10^{98}
Chess	10^{46}	10^{123}
Chinese Chess	10^{48}	10^{150}
Shogi	10^{71}	10^{226}
Go (19×19)	10^{172}	10^{360}

Table 3.4: Estimated State-Space Complexities and Game-Tree Complexities of various games [van den Herik et al., 2002] and Tigers and Goats (sorted by Game-tree complexity)

are primarily concerned with the complexity of solving the placement phase of Tigers and Goats. While the placement phase of Tigers and Goats has low state-space complexity, its game-tree complexity is larger than that of Awari and Checkers. In addition, unlike the solving of Nine Men's Morris [Gasser, 1996], we do not have the benefit of an existing strong program that can play the game well, or the availability of experts to guide us in constructing heuristics for the game. The moderately high game-tree complexity of Tigers and Goats therefore presents a challenge to finding the game-theoretic value of the game.

3.5 Chapter Conclusions

In this chapter, we introduced the game of Tigers and Goats and analyzed its state-space complexity and game-tree complexity. In addition, we solved the sliding phase (endgame phase) of Tigers and Goats by computing the game-theoretical values of all 88,260,972 positions. The complexity of solving Tigers and Goats was compared with other games and we found that solving the placement phase of the game is computationally hard but within current state of the art.

Chapter 4

Goat has at least a Draw

Consider a computer system built to the specifications of DEEP BLUE so that it can search 200 million positions per second in the game of Tigers and Goats. It would still take approximately 1.5×10^{25} *years* to completely traverse the game-tree of Tigers and Goats, which is estimated at 10^{41}. The computationally hard problem of finding the game-theoretic value of Tigers and Goats clearly requires a more directed search.

However, the naïve idea of performing selective Minimax search for both players accomplishes little - if both players' moves are forward pruned during the search, then the result is neither a lower nor upper bound on the true game-theoretic value. If, on the other hand, we forward prune only Goat moves during search, then the result returned is a lower bound on the true game-theoretic value for Goat. This is simple to see once we note that the moves forward pruned during search might lead to better results for Goat. Similarly, if we forward prune only Tiger moves, then the result returned is a lower bound on the game-theoretic value for Tiger.

We attempt to show that the game is at least a draw for Goat as the higher branching factor of Goat means that the reduction in computational effort if Goat is selectively

searched is potentially much larger than if Tiger is selectively searched. This requires good heuristics to differentiate between "good" and "bad" Goat moves during the search.

The idea of incorporating domain-specific knowledge to speed up brute-force methods to solve games is not new - games like Connect-Four [Allis, 1994], Nine Men's Morris [Gasser, 1996], Go-Moku [Allis, 1994] and Kalah [Irving et al., 2000] have been solved using a combination of brute-force and knowledge-based methods. However, given our lack of access to human expertise, we used evolutionary computing to learn heuristic players.

In this chapter, we show how we developed these heuristic players and how we used them to perform forward pruning in order reduce the game-tree complexity sufficiently to be able to show that Goat has at least a draw.

4.1 Cutting Search Trees in Half

A 39-ply search with a branching factor that often exceeds a dozen legal moves is a big challenge. Therefore, the key to successful forward searches through the state-space S_0 of the placement phase is to replace a 39-ply search with a number of carefully designed searches that are effectively only 20 plies deep. This is achieved by 1) formulating the hypotheses of the type "player X can achieve result Y", such as "Goat has at least a draw", 2) programming a competent and efficient heuristic player X that generates only one or a few candidate moves in each position, and 3) confronting the selective player X with his exhaustive opponent who tries all his legal moves. If this search that alternates selective and exhaustive move generation succeeds, the hypothesis Y is proven. In the ideal scenario, the search tree is "cut in half" to a 20-ply search if the first move suggested by the selective player X succeeds against all his exhaustive opponent's

legal moves.

If the search fails, one may try to develop a stronger heuristic player X, or weaken the hypothesis, e.g. from "X wins" to "X can get at least a draw". Using such searches designed to verify a specific hypothesis we were able to prove several results including the following: 1) Tiger can force the capture of a single goat within 39 plies, i.e. within the placement phase, and 2) Tiger can force the capture of two goats within 40 plies, i.e. by the end of the placement phase, but not earlier.

4.1.1 Heuristic Attackers and Defenders

In order to make these searches feasible we had to develop strong heuristic players. Given our lack of access to human expertise, we developed player programs that learn from experience by being pitted against each other. For example, the proof that Tiger can kill a certain number of goats requires a strong Tiger that tries to overcome an exhaustive Goat. Conversely, the proof that Goat has a drawing strategy after the most plausible opening requires a strong heuristic Goat that defies an exhaustive Tiger.

Since Goat usually has more legal moves than Tiger, it seemed prudent to prove that Goat has at least a draw based on the winning criterion of capturing five goats by focusing on developing good heuristic Goat players. Recently, Chellapilla and Fogel implemented a co-evolutionary system [Chellapilla and Fogel, 2001b] using neural networks as evaluation functions that taught itself to play Checkers at expert level. Co-evolution is a flexible method for learning strategies to complex games where there is little available information about the domain. In the absence of expert knowledge for Tigers and Goats, we used co-evolutionary computing to learn a heuristic Goat player.

Co-evolutionary techniques have been applied to several games in the past, including

Chess [Kendall and Whitewell, 2001], Go [Luuberts and Miikkulainen, 2001], and Othello [Moriarty and Miikulainen, 1995]. However, possibly the most successful demonstration of the machine learning capability of evolutionary computation in the creating of a game-playing program was performed by Kumar Chellapilla and David Fogel in the creation of their Checkers program Anaconda [Chellapilla and Fogel, 1999, Chellapilla and Fogel, 2001b, Fogel, 2002]. This program, created using co-evolution of artificial neural networks without expert knowledge, is able to play Checkers at expert level, and is a success story for evolutionary computing.

4.2 Neural Network Architecture

The setup of our neural networks follows the one used in [Chellapilla and Fogel, 1999], but instead of restricting the evaluation function inputs to the raw board, we selected 26 features (shown in Figure 4.1) of the board position as inputs. Examples of the selected features include whose turn to move, the number of goats captured so far, the number of tigers which are currently trapped (i.e., unable to move), the number of legal tiger moves, and the number of goat moves that avoid immediate capture. An initial population of neural networks is generated with random weights. Each neural network competes against a fixed number of randomly chosen opponents and its fitness score is determined by the results of these games. After every neural network has played its games, the "fittest" or highest-scoring networks are retained, while the rest are removed. Each of the retained neural networks creates an offspring by mutating its weights. The process is then repeated until the desired number of iterations is reached.

The architecture of each neural network is illustrated in Figure 4.1: (1) the inputs have 24 nodes representing the current board position and there are 2 inputs directly

Index	Description of Feature
1	Number of captured goats
2	Number of trapped tigers
3	Number of possible Tiger non-capture moves
4	Number of possible Tiger capture moves
5	Sum of all Manhattan distances between pairs of tigers
6	Sum of all Manhattan distances between a tiger and the closest corner it is to
7	Number of empty points which are surrounded by goats
8	Number of empty points on the edge which are surrounded by goats
9	Number of empty points not on the edge which are surrounded by goats
10	Number of goats which are immediately capture-able by a tiger
11	Number of goats in a corner of the board
12	Number of goats on the edge of the board
13	Number of goats not on the edge of the board
14	Number of goats which are next to another goat
15	Number of goats which are next to a tiger
16	Number of goats which can be captured by a forced sequence of two ply
17	Number of Tiger non-capture moves that result in a capture threat
18	Number of Tiger non-capture moves that result in the tiger on the edge
19	Number of Tiger capture moves that result in a capture threat
20	Number of Tiger capture moves that result in the tiger on the edge
21	Number of tigers in a corner of the board
22	Number of tigers on the edge of the board
23	Number of tigers not on the edge of the board
24	Number of goats on the board
25	Player to move - it can be Tiger (-1) or Goat (+1)
26	Piece on the center of the board - it can be a tiger (-1), empty (0) or a goat (+1)

Table 4.1: Input Features to the Neural Network

linked to the output node, (2) there are two hidden layers and they consist of 12 and five nodes, respectively, and (3) the single output node gives the evaluation of the board. The transfer function of each node in the neural network is the hyperbolic tangent (tanh bounded by ± 1). Hence an output value closer to 1.0 denotes a better position for the player to move, and a value closer to -1.0 denotes a worse position.

Formally, we define the output o of a neuron with weights $\{w_1, w_2, \ldots, w_N\}$ and inputs $\{x_1, x_2, \ldots, x_N\}$ by

$$o = tanh(\sum_{i=1}^{N} w_i x_i + \sigma) \tag{4.1}$$

where σ is a scalar threshold value.

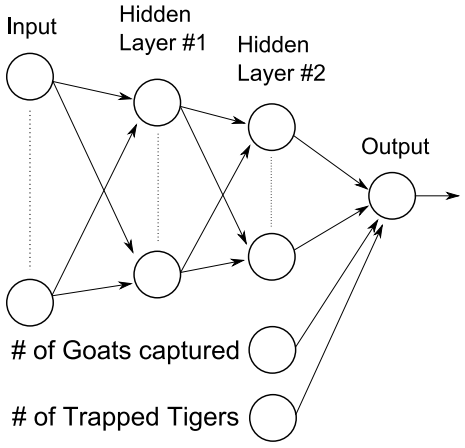


Figure 4.1: Neural Network Architecture of Tigers and Goats evaluation function

When initializing the neural networks, the connection weights are generated randomly from a uniform distribution over [-0.2, 0.2]. Each connection also has an associated threshold $\sigma_i(j)$, for $j = 1, \ldots, N_w$, which are all set initially to 0.05. Reproduction is achieved solely via mutation (no crossover operation is used). Specifically, for each parent P_i, an offspring P_i' is created by

$$\sigma_i'(j) = \sigma_i(j)exp(\tau x) \tag{4.2}$$

$$w_i'(j) = w_i(j) + \sigma_i'(j)y_j \tag{4.3}$$

where $\tau = 1/\sqrt{2\sqrt{N_w}}$, N_w is the number of connection weights, $x \sim N(0,1)$, $y_j \sim N(0,1)$, and $N(0,1)$ is the standard Gaussian random distribution with mean 0 and standard deviation 1.

4.3 Variants of Learning Heuristic Players

For our experiments, the resultant neural networks are used to perform move ordering and forward pruning during a search to prove the hypothesis "Goat can at least draw the game". Pitting a heuristic Goat player against all attacks by Tiger breaks into six cases as shown in Figure 4.2. However, as mentioned in chapter 3, only the move at the middle of the edge avoids an early capture. We therefore started the searches in our experiments by first placing a goat in the middle of the edge.

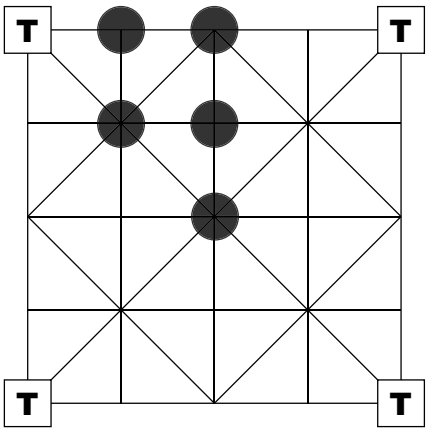


Figure 4.2: Goat has six symmetrically distinct initial moves (highlighted)

The top neural network obtained after 1,000 generations is used as a move ordering mechanism in a search to prove that Goat can at least draw the game. At each Goat-to-move node, moves are re-ordered according to a two-ply shallow search using the neural network as the evaluation function. Only the top three Goat moves are then tried, and the rest of the legal moves are pruned.

We experimented with four co-evolutionary setups as shown in Table 4.2. These setups are combinations of two features - (1) the number of populations (OnePop or TwoPop) and (2) whether or not the search depths are biased depending on which player

the neural network is playing as (Normal or Biased).

For OnePop setups, a single population of 30 neural networks are created and each neural network can be used as the evaluation function for either Tiger or Goat. For TwoPop setups, two populations of 20 neural networks each are created, and one population is designated as the evaluation functions for Goat and the other population is designated as the evaluation functions for Tiger. The OnePop setups are used in [Chellapilla and Fogel, 1999] to evolve evaluation functions for Checkers. However, in Checkers, the objectives of both players are the same - capture all opponent pieces. It is therefore appropriate to evolve the players using the same population. On the other hand, the players in Tigers and Goats have asymmetric objectives, and could require vastly different neural networks to evaluate accurately. The TwoPop setups should then allow the neural networks to learn specialized evaluation functions appropriate for its designated player. Furthermore, since the evaluation functions are used to score positions for only one type of players in TwoPop setups, it can be argued that the learning task in these populations is easy than in the OnePop setups which have to learn to score positions for both Tiger and Goat. It is therefore expected that the neural network evolved under TwoPop setups will perform better than those evolved under OnePop setups.

For Normal setups, moves are made using a search limited to depth 4 for both Tiger and Goat. For Biased setups, searches are limited to depth 4 when playing as Tiger, and depth 2 when playing as Goat. It is apparent that the Biased setups are placing Goat at an disadvantage during the evolutionary process. However, as we expect to use the neural network as a forward pruning heuristic to prove that Goat has at least a draw, this means that the neural network has to be able to suggest moves that can achieve a draw against *all* Tiger moves. The Biased setups are therefore a method of creating evaluation functions that can play conservatively against a strong opponent.

Name	Description
OnePopNormal	A single population of 30 neural networks is created. When playing as Tiger, the search is limited to depth 4. When playing as Goat, the search is limited to depth 4.
OnePopBiased	A single population of 30 neural networks is created. When playing as Tiger, the search is limited to depth 4. When playing as Goat, the search is limited to depth 2.
TwoPopNormal	Two populations of 20 neural networks each are created. One population is designated as the evaluation functions for Goat and the other for Tiger. When playing as Tiger, the search is limited to depth 4. When playing as Goat, the search is limited to depth 4.
TwoPopBiased	Two populations of 20 neural networks each are created. One population is designated as the evaluation functions for Goat and the other for Tiger. When playing as Tiger, the search is limited to depth 4. When playing as Goat, the search is limited to depth 2.

Table 4.2: Description of Co-Evolutionary Setups

Each neural network plays against four randomly chosen neural networks of the opposite player. Similar to [Chellapilla and Fogel, 1999], we employ an asymmetric scoring system where we reward a win more than we penalize a loss by awarding +2 to a win, 0 to a draw and -1 to a loss. The fitness of each neural network is the sum of the scores achieved after playing the four opponents. For single population setups, the top 15 neural networks are retained for the next generation. For two population setups, the top 10 neural networks of each population are retained for the next generation. Each retained neural network generates one offspring by mutating its weights. This is the end of one generation and the process is repeated until 1,000 generations have been produced.

4.4 Performance of Heuristic Players

Of the four co-evolutionary setups shown in Table 4.2, only TwoPopBiased succeeded in proving that in the initial position shown in the left of Figure 3.5 is at least a draw for Goat. In this experiment, we used a simple synchronous Alpha-Beta search [Hopp

and Sanders, 1995] to perform parallel tree search. Using 10 Pentium 4 Xeons, each process managing its own hash table of 8,000,000 entries, the search took 6 months to prove this position to be at least a draw for Goats. All the other three setups failed during the search, i.e. Tiger manages to find a winning sequence against the defensive moves suggested by the neural network.

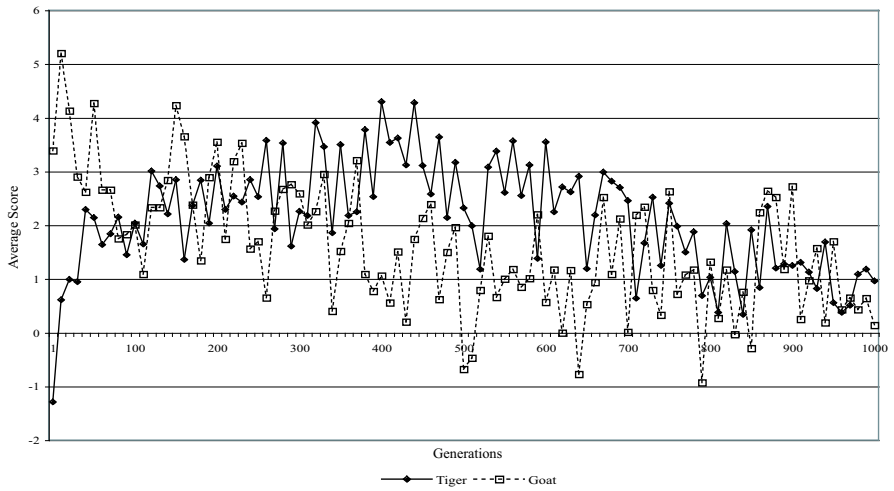


Figure 4.3: Average scores of all neural networks of each generation in TwoPopNormal

It is useful to see how the biased search depths affect the co-evolution of the neural networks. In Figure 4.3, we see that the Tiger and Goat are comparable in relative playing ability across the generations during co-evolutionary in TwoPopNormal. Normally, this is desirable as the dynamics of competitive co-evolution [Rosin and Belew, 1997] would result in an "arms-race" where both players try to outmaneuver each other by learning better strategies. However, an aggressive Goat player that tries to win by making risky moves would not likely be a good heuristic to use in proving that the game is at least a draw for Goat. Notice that the average scores of all neural network are usually above 0. This is due to the unbalanced scoring system that assigns +2 to wins, 0 to draws and -1 to losses.

The plot of averages scores for TwoPopBiased in Figure 4.4 shows that the higher search depth of the heuristic Tiger player resulted in Tiger initially winning more often during the evolutionary process. However, Goat soon learnt to draw the game despite having the handicap of a lower search depth. The handicap during the evolutionary process prepares the neural network for Goat to handle opponents that search deeper than it does, and therefore the neural network is still be able to achieve a draw. This is vital, as the top neural network for Goat after 1,000 generations is used to perform move ordering and forward pruning during the search.

The plots for OnePopNormal and OnePopBiased are not shown as they provide little insight to the individual playing ability of either player. From the results, we see that while the single population mechanics of [Chellapilla and Fogel, 1999] is well suited to learning evaluation functions for both players, it is difficult to use the same mechanism to create an effective heuristic.

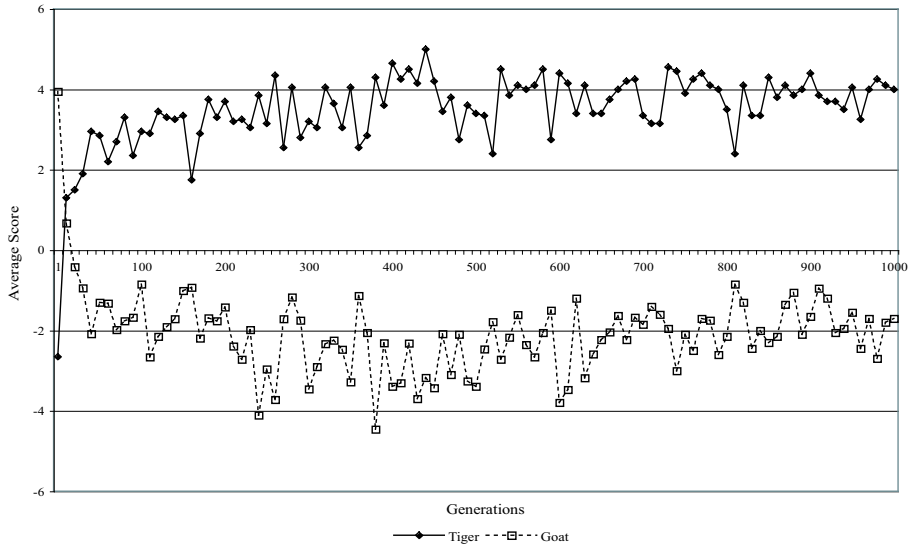


Figure 4.4: Average scores of all neural networks of each generation in TwoPopBiased

4.5 Chapter Conclusion

By co-evolving two populations, each designating a player, instead of a single population like in [Chellapilla and Fogel, 1999, Chellapilla and Fogel, 2001b, Fogel, 2002], we show that co-evolution can learn search heuristics by biasing specific populations. We experimented with this method in the game of Tigers and Goats, and created a heuristic player for Goat.

By restricting the search depth of Goat to 2 while Tiger searched to depth 4, the co-evolution created a heuristic which was used to prove that Goat can at least draw the game (with the winning criterion of five goats captured). It is important to note that due to the lack of human expert knowledge of the game, attempts to create a heuristic "by hand" to re-order and prune moves during the search failed. This means that this result would not have been possible without the heuristic learnt by the co-evolutionary technique.

The experiments also showed that the co-evolutionary setup for a single population with and without biased search depths could not learn a search heuristic that was able to prove the result. This suggests that the two population co-evolutionary setup is more suitable to zero-sum two-player games with perfect information. More work and experiments with other games are needed to further investigate this result.

Nevertheless, we have shown that Goat has at least a draw in Tigers and Goats. Furthermore, we have shown that co-evolutionary computing is capable of creating heuristics for use in a search to find the game-theoretic result of a two-player zero sum game with perfect information.

Chapter 5

Tigers and Goats is a Draw

This chapter is a modified version of our paper "Tigers and Goats is a Draw", to be published in the forthcoming book *Games of No Chance 3*, edited by Michael Albert and Richard Nowakowski.

In this chapter we describe how a combination of retrograde analysis and Alpha-Beta search was used to show that Tiger has at least a draw. Combining this result with the evidence that Goat can at least draw (Chapter 4), we show that each player can achieve at least a draw under optimal play, and thus, weakly solve the game of Tigers and Goats.

5.1 Insights into the Nature of the Game

We were unable to discover easily formulated advice to players beyond plausible rules-of-thumb such as "goats cautiously hug the border, tigers patiently wait to spring a surprise attack". On the other hand, our database explains the seemingly arbitrary number "five" in the usual winning criterion "Tiger wins when five goats have been killed". This magic number "five" must have been observed as the best way to balance the chances. We know that Tiger can kill some goats, so Tiger's challenge must be more ambitious

than “kill any one goat”. On the other hand, we see from Table 3.2 that there is a significant jump in the number of lost positions for Goat from three goats captured to four goats captured. It is therefore fairly safe to conjecture that once half a dozen goats are gone, they are all gone - Goat lacks the critical mass to put up resistance. But as long as there are at least 16 goats on the board, i.e. at most four goats are captured, the herd is still large enough to have a chance to trap the tigers.

Table 3.2 also shows that unless Tiger succeeds in capturing at least two goats during the placement phase, he has practically no chance of winning. If he enters the sliding phase facing 19 goats, less than 2% of all positions are won for Tiger, regardless of whether it is his turn to move or not. The fact that Tiger can indeed force the capture of two goats within 40 plies, i.e. by the end of the placement phase, as stated in Section 4.1, is another example of how well-balanced the opponents’ chances are.

5.2 Tiger has at least a Draw

The result that Goat has at least a draw had brought us tantalizingly close to determining the game-theoretic value of Tigers and Goats. Computing the endgame database had been relatively straightforward, but the 39-ply forward search had not yielded to the judicious application of established techniques. Experience had shown that by “cutting the tree in half” as described in Section 4.1, i.e. approximating a 39-ply search by various 19-ply and 20-ply searches, we were able to answer a variety of questions. It appeared plausible that by formulating sufficiently many well-chosen hypotheses this approach would eventually yield a complete analysis of the game. We conjectured that Tiger also has a drawing strategy, and set out to try to prove this using the same techniques that had yielded Goat’s drawing strategy.

The asymmetric role of the two opponents, however, made itself felt at this point: the searches pitting a heuristic Tiger player against an exhaustive Goat progressed noticeably more slowly than those involving a heuristic Goat versus an exhaustive Tiger. In retrospect we interpret this different behavior as due to the phenomenon "Tiger's play is all tactics". Positional considerations - keep the goats huddled together - make it easy to generate one or a few "probably safe" Goat's moves, even without any look-ahead at the immediate consequences. For Tiger, on the other hand, neither we nor apparently the neural network that trained the player succeeded in recognizing "good moves" without a local search. An attempt to make Tiger a stronger hunter (by considering the top 3 moves suggested by the neural network followed by a few plies of full-width search) is inconsistent with the approach of "cutting the tree in half" and made the search unacceptably slow.

Thus, a new approach had to be devised. The experience that 20-ply forward searches proved feasible suggests a more direct approach: compute a database of positions of known value halfway down the search tree. Specifically, we define *halfway position* as one arising after 19 plies, i.e. after the placement of 10 goats, with Tiger to move next. The value of any such position can be computed with a search that ends in the endgame database after at most 20 plies. If sufficiently many such "halfway positions" are known and stored, searches from the root of the tree (the starting position of the game) will run into them and terminate the search after at most 19 plies.

The problem with this approach is that the number of halfway positions is large, even after symmetric variants have been eliminated. Because of captures not all 10 goats placed may still be on the board, hence a halfway position has anywhere between 6 and 10 goats, and correspondingly, 15 to 11 empty spots. Using the terminology of Section 3.3, the set of halfway positions is (perhaps a subset of) the union of S_{11}, S_{12}, S_{13},

S_{14} and S_{15}, where S_k is the set of all symmetrically inequivalent positions containing 4 tigers, $21-k$ goats, and k empty spaces. S_{11}, with about equally as many goats as empty spots, is particularly large. On the assumption that in any subspace S_k the number of symmetrically inequivalent positions is close to 1/8 of the total, S_{11} contains about 550 million inequivalent positions. The union of S_{11} through S_{15} contains about 1.6×10^9 positions. This number is about 25 times larger than the largest endgame database we had computed before, namely S_5.

The approach to overcome the problem of constructing a large halfway database exploits two ideas. First, the database of halfway positions of known value need not necessarily include all halfway positions. In order to prove that Tiger has a drawing strategy, the database need only include a sufficient number of positions known to be drawn or a win for Tiger so that any forward search is trapped by the filter of these positions. Second, the database of halfway positions is built on the fly: whenever a halfway position is encountered whose value is unknown, this position is entered into the database and a full-width search continues until its value has been computed.

Although there was no a priori certainty that this approach would terminate within a reasonable time, trial and error and repeated program optimization over a period of five months led to success. Table 5.1 contains the statistics of the halfway database actually constructed. For each of S_{15} through S_{11}, it shows the number of positions whose value was actually computed, broken down into the two categories relevant from Tiger's point of view, win-or-draw vs. loss.

Although the construction of the halfway database is intertwined with the forward searches, i.e. a position is added and evaluated only as needed, logically it is clearest to separate the two. We discuss details of the forward searches in the next section.

Num Captured	# Win-or-Draw	# Loss	Total	Estimated State Space Size
4 Captured	17,902,335	0	17,902,335	85,804,950
3 Captured	33,152,214	0	33,152,214	183,867,750
2 Captured	64,336,692	17,944	64,354,636	321,768,563
1 Captured	84,832,697	329,183	85,161,880	464,776,813
0 Captured	15,857,243	91,676	15,948,919	557,732,175
Total	216,081,181	438,803	216,519,984	1,613,950,251

Table 5.1: Halfway database statistics: the number of positions computed and their value from Tiger's point of view: win-or-draw vs. loss

5.3 Implementation, Optimization, and Verification

Our investigation of Tigers and Goats has been active, on and off, for the past three years. The resources used have varied form a Pentium 4 personal computer to a cluster of Linux PC workstations. Hundreds of computer runs were used to explore the state-space, test and confirm hypotheses, and verify results. The longest continuous run lasted for five months as a background process on an Apple PowerMac G5 used mainly for web surfing.

The algorithmic search techniques used are standard, but three main challenges must be overcome in order to succeed with an extensive search problem such as Tigers and Goats. First, efficiency must be pushed to the limit by adapting general techniques to the specific problem at hand, such as the decision described above on how to combine different search techniques. Second, programs must be optimized for each of the computer systems used. Third, the results obtained must be verified to insure they are indeed correct. We address these three issues as follows.

5.3.1 Domain Specific Optimizations

The two databases constructed, of endgame positions and halfway positions, limit all forward searches to at most 20 plies. Still, performing a large number of 20-ply searches

in a tree with an average branching factor of 10 remains a challenge that calls for optimization wherever possible.

The most profitable source of optimizations is the high degree of symmetry of the game board. Whereas the construction of the two databases of endgame and halfway positions is designed to avoid symmetric variants, this same desirable goal proved not to be feasible during forward searches - it would have meant constructing a database consisting of all positions. Instead, the goal is to avoid generating some, though not necessarily all, symmetrically equivalent positions in the first place when this can be done fast, namely during move generation.

During the placement phase, any empty intersection is a legal goat move. Thus for each empty point on the board, we pre-compute the result of performing each symmetry operation and store the operations that generate the lowest board index. During the forward search, we check which symmetries are active on the board. The empty points that generate the lowest index from an active symmetry are redundant, as there is already an empty point (and therefore a valid move by Goat) that already generates the same board position under symmetry. For the Goat player, this check is sufficient to generate each canonical position once.

For Tiger moves, the general idea is straightforward - Any position that arises during the search is analyzed to determine all active symmetries. Thereafter, among all the moves that generate symmetric outcomes, only the one that generates the resulting position of lowest index is retained. This analysis guarantees that all immediate successors to any given position are inequivalent. Because of transpositions, of course, symmetric variants will appear among successor positions further down in the tree. In other words, this check ensures that only “canonical” Tigers can move. However, an additional check is needed to ensure that the Tiger makes only canonical moves.

Table 5.2 shows the effect of this symmetry-avoiding move generation for the starting position. Although there is a considerable reduction in the number of positions generated, the relative savings diminish with an expanding horizon.

Ply	Naïve move gen.	Symmetry-avoiding move gen.	# of distinct positions
1	21	5	5
2	252	36	33
3	5,052	695	354
4	68,204	9,245	2,709
5	1,304,788	173,356	18,906
6	18,592,000	2,441,126	93,812

Table 5.2: Number of positions created by different move generators

5.3.2 System Specific Optimization

The proof that Goat has at least a draw in Tigers and Goats used a cluster of eight Linux PC workstations with a simple synchronous distributed game-tree search algorithm. However, there are fundamental problems with synchronous algorithms, discussed in [Brockington, 1997], that limit their efficiency. Furthermore, the cluster was becoming more popular and was constantly overloaded. We therefore decided against implementing a more sophisticated asynchronous game-tree search and instead relied on a sequential program running on a single dedicated processor.

We focused our attention on improving the sequential program to run on an Apple PowerMac G5 1.8 GHz machine running Mac OS-X. Firstly, the neural network code was optimized using the Single Instruction Multiple Data (SIMD) unit in the PowerPC architecture called AltiVec. AltiVec consists of highly parallel operations which allow simultaneous execution of up to 16 operations in a single clock cycle. This provided a modest improvement of about 15% to the efficiency of neural network evaluations of the board, but sped up the overall efficiency of the search much more as the neural network is used repeatedly within the search to evaluate and re-order the moves.

Next, we moved much of the computations off-line. For example, the moves for Tiger at each point on the board in every combination of surrounding pieces are pre-computed into a table so that the program simply retrieves the table and appends it to the move list during search. Operations like the indexing of the board and symmetry transformation are also pre-computed such that the program only needs to retrieve data from memory to get the result. Finally, we recompiled the software with G5-specific optimizations.

5.3.3 Verification

Two independent re-searches confirm different components of the result. They used separately coded programs written in C, and took 2 months to complete.

The first verification search used the database of halfway positions to confirm the result at the root, namely, "Tiger has a drawing strategy". Notice that this verification used only the positions marked as win-or-draw in the database.

The second verification search confirmed the halfway positions marked as win-or-draw by searching to the endgame database generated by the retrograde analysis described in [Lim and Nievergelt, 2004]. All other positions can be ignored, as they have no effect on the first search.

Re-verification with even more Forward Pruning

Another program was written in C to 're-prove' the results. This program had the benefit of a posteriori knowledge that the game is a draw, and this fact allowed us to concentrate on using aggressive forward pruning techniques to verify the result. The program used the same domain-specific optimizations such as symmetry reduction and the halfway databases.

The halfway database was optimized for size by storing the boolean evaluation of each position using a single bit. Depending on the type of search, this boolean evaluation could mean "Goat can at least draw" or "Tiger can at least draw". Due to this space optimization the halfway positions and endgame databases could be stored in memory, thereby avoiding disk accesses and speeding up the search by orders of magnitude.

As Tiger is able to force the capture of two goats only by the end of the placement phase (i.e. at ply 40), any position during the forward pruning with two goats already captured can be deemed to be sub-optimal. In conjunction with move ordering and forward pruning with the co-evolved neural network (see Chapter 4), the search for "Goat can at least draw" used an aggressive forward pruning strategy of pruning positions which had two (or more) goats already captured. The halfway database was set at ply 23, when 12 goats have already been placed and it is Tiger's turn to move. The search confirmed that "Goat can at least draw" in approximately 7 hours while visiting 7,735,443,119 nodes.

The program was also able to confirm that "Tiger can at least draw". Due to the large game-tree complexity of this search, two "halfway" databases were placed at ply 21 and ply 31. These databases contribute towards efficiency in two ways: first, they terminate some searches early, and second, they generate narrower search trees. The latter phenomenon is due to the fact that these databases are free of symmetrically equivalent positions. In exchange for a large memory footprint of approximately 2 GB, search performance was dramatically improved. Additional forward pruning heuristics eliminated nodes where Tiger had few legal moves. The searched confirmed that "Tiger can at least draw" in approximately 48 hours while visiting 40,521,418,103 nodes.

Additional searches of the program to prove "Goat can at least draw" and "Tiger can at draw" without forward pruning ran for more than a week without results before

they were terminated. This shows that while the halfway databases helped to exploit the relatively small state-space complexity of Tigers and Goats, forward pruning played a major role in the efficiency of the search.

5.4 Chapter Conclusion

The theory of computation has developed powerful techniques for estimating the asymptotic complexity of problem classes. By contrast, there is little or no theory to help in estimating the concrete complexity of computationally hard problem instances, such as determining the game-theoretic value of Tigers and Goats. Although the general techniques for attacking such problems have been well-known for decades, there are only rules of thumb to guide us in adapting them to the specific problem at hand in an attempt to optimize their efficiency [Nievergelt, 2000].

The principal rule of thumb we have followed in our approach to solving Tigers and Goats is to pre-compute the solutions of as many subproblems as can be handled efficiently with the storage available, both in main memory (hash-tables) and disks (position data bases). If the net of these known subproblems is dense enough, forward pruning techniques serves to truncate the depth of many forward searches, an effect that plays a decisive role since the computation time tends to grow exponentially with search depth. This suggests that developing better forward pruning techniques can enable us to solve ever more computationally hard problems in the future.

Chapter 6

RankCut

This chapter is an extended version of our article "RankCut – A Domain Independent Forward Pruning Method for Games" presented in the Twenty-First National Conference on Artifical Intelligence (AAAI-06).

Alpha-Beta pruning [Knuth and Moore, 1975] and its variants like NegaScout/Principal Variation Search [Reinefeld, 1989] and MTD(f) [Plaat, 1996] have become the standard methods used to search game-trees as they greatly reduce the search effort needed. Apart from theoretical exceptions where decision quality decreases with search depth [Beal, 1980, Nau, 1979], it is generally accepted that searching deeper will result in higher move decision quality [Junghanns et al., 1997]. However, search effort increases exponentially with increasing search depth. To further reduce the number of nodes searched, game-playing programs perform forward pruning [Marsland, 1986], where a node is discarded without searching beyond that node if it is believed that the node is unlikely to affect the final Minimax value of the node.

In this chapter, we describe RankCut [Lim and Lee, 2006b], which estimates the

probability of discovering a better move later in the search by using the relative frequency of such cases for various states during search. These probabilities are precomputed off-line using several self-play games. RankCut can then reduce search effort by performing a shallow search when the probability of a better move appearing is below a certain threshold. RankCut implicitly requires good move ordering to work well. However, game-playing programs already perform move ordering as it improves the efficiency of Alpha-Beta search, and thus good move ordering is available at no extra cost. We implement RankCut in the open source Chess programs, CRAFTY and TOGA II, and show its effectiveness with test suites and matches.

6.1 Existing Forward Pruning Techniques

Even with the search enhancements described in the chapter 2, search effort increases exponentially with increasing search depth. However, a difficult conundrum arises as, based on empirical data, move decision quality improves with increasing search depth [Condon and Thompson, 1983, Thompson, 1982, Heinz, 1998a, Heinz, 2001b, Junghanns et al., 1997]. To further reduce the number of nodes searched, game-playing programs perform forward pruning [Marsland, 1986, Björnsson and Marsland, 2000b], where a node is discarded without searching beyond that node if it is believed that the node is unlikely to affect the final Minimax value of the node. This means that some good move sequences might not be explored with forward pruning, which can affect the Minimax evaluation and hence the move chosen by the search. Thus, using forward pruning methods entails a compromise between the risk of making an error and pruning more, thereby potentially making more errors, to allow deeper searches.

Search extensions can be viewed as a type of forward pruning as the search is typically not full-width, and thus certain nodes are not visited despite no theoretical assurance that they will not affect the Minimax value. However, it is useful to understand the differences in the viewpoints of search extensions and forward pruning techniques. As suggested by their names, search extension techniques strive to explore potential good moves deeper to better evaluate them; forward pruning techniques strive to prune off potential bad moves to avoid searching unnecessary parts of the game-tree.

Despite the stigma of forward pruning techniques during the early development of Chess programs [Abramson, 1989], modern game-playing programs for Chess [Donninger, 1993,Björnsson and Newborn, 1997], Checkers [Schaeffer, 1997], Othello [Buro, 1997a], Shogi [Iida et al., 2002], Abalone [Aichholzer et al., 2002] employ forward pruning along with search extensions such as quiescence search, i.e., the search is not full-width in some non-quiescence positions in the game-tree. This is largely due to the development of several highly effective domain-independent forward pruning techniques.

6.1.1 Razoring and Futility Pruning

Both *Razoring* and *Futility Pruning* have been successfully used in Chess programs. Razoring was first introduced in [Birmingham and Kent, 1977], while Futility Pruning was first described in Schaeffer's PhD thesis [Schaeffer, 1986]. Both techniques observe that static evaluations of a board position one ply from the leaf nodes, also known as frontier nodes, tend to be accurate enough to identify bad positions that need not be considered further.

Razoring techniques [Birmingham and Kent, 1977, Heinz, 1998b] prune nodes just above the horizon if their static evaluations fail low by a predetermined margin. Futility

Pruning further observes that evaluation functions can usually be separated into major and minor components. For example, in Chess, the major component would include material count and the minor component would include positional evaluations. As the minor component of the evaluation is composed solely of smaller adjustments to the score, it can typically be bounded. At frontier nodes, if the evaluation of major components falls greatly outside Alpha-Beta bounds and the bounds on the evaluation of the minor component cannot possibly adjust the score to within the Alpha-Beta bound, the frontier node can be safely pruned.

Heinz improved on Futility Pruning by applying Futility Pruning to pre- and pre-pre-frontier nodes and called it "Extended Futility Pruning" [Heinz, 1998b]. Experiments with well-known test suites and the Chess program DARKTHOUGHT [Heinz, 1998a] showed improvements of $10\% - 30\%$ in fixed search depths of eight-12 plies.

6.1.2 Null-Move Pruning

Null-Move Pruning [Beal, 1989, Goetsch and Campbell, 1990, Donninger, 1993], which was introduced in the 1990s, enables programs to search deeper with little tactical risk. Null move pruning assumes that not making any move and passing (also known as making a *null move*), even if passing is illegal in the game, is always bad for the player to move. So when searching to depth d, the heuristic makes a null move by simply changing the turn to move to the opponent and performs a reduced-depth search. If the reduced-depth search returns a score greater than Beta, then the node is likely to be a strong position for the player to move and will have a score greater than Beta when searched to depth d without the null move. The program should therefore fail-high and return Beta.

However, there is a class of positions known as *zugzwang* positions where making

Pseudocode 4 NullMovePruning($state$, α, β, $depth$)

```
1: if depth == 0 or isTerminal(state) then
2:     return Evaluate(state)
3: MakeNullMove(state)
4: R ← depthReduction(state)
5: score ← -NullMovePruning(successor(state, move), depth − 1 − R, −β, −β + 1)
6: UnmakeNullMove(state)
7: if score ≥ β then
8:     return β
9: // standard Alpha-Beta search follows
10: ...
```

the null move is actually good for the player to move. This violates the assumption that Null-Move Pruning makes and causes search errors. In Chess, zugzwang positions normally occur during endgame positions. As a result, Null-Move Pruning is disabled when the game is likely to be in the endgame phase (e.g., when the number of pieces on the board is small).

6.1.3 ProbCut

The ProbCut heuristic [Buro, 1995a] prunes nodes that are likely to fail outside the Alpha-Beta bounds during search by using simple statistical tools. ProbCut first requires off-line computations to record results of both shallow and deep searches of the same positions. The results of both searches are then correlated using linear regression. Formally, the result v' of a shallow search is used to estimate the result v of a deeper search by a linear relationship:

$$v = a \cdot v' + b + e$$

where e is a normally distributed error variable with mean 0 and standard deviation σ. The parameters a, b and σ can be estimated by linear regression over training search results. ProbCut is therefore able to form a confidence interval from the result of a shallow search to obtain an estimated range in which a deep search will return. By algebraic manipulation, given v', $v \geq \beta$ with probability at least p is equivalent to $v' \geq (\Phi^{-1}(p) \cdot \sigma + \beta - b)/a$, where Φ is the standard Gaussian distribution. Similarly, $v \leq \alpha$ with probability at least p is equivalent to $v' \leq (-\Phi^{-1}(p) \cdot \sigma + \alpha - b)/a$.

Pseudocode 5 ProbCutAlphaBeta($state$, α, β, $depth$, $height$)

```
1:  S ← 4 // shallow search depth
2:  H ← 8 // check height
3:  T ← 1.0 // pruning threshold
4:  if depth == 0 or isTerminal(state) then
5:     return Evaluate(state)
6:  if height == H then
7:     bound ← (T × σ + β − b)/a
8:     if ProbCutAlphaBeta(successor(state, move), bound − 1, bound, S) ≥ bound
       then
9:        return β
10:    bound ← (−T × σ + α − b)/a
11:    if ProbCutAlphaBeta(successor(state, move), bound, bound + 1, S) ≤ bound
       then
12:       return α
13: // standard Alpha-Beta search follows
14: ...
```

If this confidence interval falls outside the Alpha-Beta bounds, the program should fail high or low appropriately. ProbCut was successfully applied in Logistello, an Othello program that beat the then World Human Othello Champion 6-0 under standard time controls [Buro, 1997b].

Multi-ProbCut [Buro, 1999] is an enhancement of ProbCut which uses different regression parameters and pruning thresholds for different stages of the game and multiple depth pairings. Preliminary experiments show that Multi-ProbCut can be successfully

applied to Chess [Jiang and Buro, 2003], but requires a fair amount of work to get good results. Jiang and Buro [Jiang and Buro, 2003] suggest that there are at least two reasons for the poor performance:

1. The Null-Move Pruning forward prunes the same type of positions as ProbCut. Since the Null-Move Pruning is commonly applied in Chess program, this results in ProbCut not being able to significantly improve game-performance when implemented alongside the Null-Move Pruning. On the other hand, ProbCut has been successfully applied to Othello as it is a zugzwang game, and therefore Null-Move Pruning is ineffective in this game.

2. The probability of forward pruning error propagating in a Chess game-tree search is high due to its high branching factor. This results in linear regression being unable to accurately model a deep search result using a shallow search result.

6.1.4 N-Best Selective Search

The N-Best Selective Search is a simple form of forward pruning that bears close resemblance to RankCut. The moves are ranked according to an evaluation function prior to considering moves for pruning, and during the search only the N best moves are considered. This heuristic requires the move ordering to be able to rank the best move within the top N moves constantly. So while this heuristic has the advantage of ensuring the branching factor of the search is at most N, it normally introduces far too many pruning errors to be a viable forward pruning technique. RankCut can be seen as a mechanism that dynamically adapts N, but also has the additional advantage of being able to adjust its risk appetite on a case-by-case basis.

Pseudocode 6 NBestSearch($state$, α, β, $depth$)

```
 1: if depth == 0 or isTerminal(state) then
 2:    return Evaluate(state)
 3: N ← NBest // pre-defined constant
 4: count ← 0
 5: for move ← NextMove(state) do
 6:    if count ≥ N then
 7:       break
 8:    N ← N + 1
 9:    // standard Alpha-Beta search follows
10:    ...
```

6.1.5 Multi-cut pruning

In Negascout/PVS, nodes are assumed to be searched in the same order as a minimal Alpha-Beta tree, and nodes can be classified as PV, CUT or ALL nodes. This allows Negascout/PVS to use minimal search windows to quickly search nodes to check that they are within bounds.

Similarly, Multi-Cut Pruning [Björnsson and Marsland, 2001] assumes that nodes can be estimated with good probability to be PV, CUT or ALL nodes. In addition, multi-cut pruning notes that the search returns a new principal variation if every expected CUT-node on the path from a leaf node to the root node must become an ALL-node. As a result, in expected CUT nodes, the first M children of a node are searched with reduced depth, and if at least $C < M$ of these children return a value greater or equal to Beta, then multi-cut pruning will assume a Beta cutoff is likely to occur if the usual full-depth search is done, and will therefore prune and return the local search immediately.

Pseudocode 7 MultiCut($state$, α, β, $depth$)

```
1: if depth == 0 or isTerminal(state) then
2:    return Evaluate(state)
3: score ← −∞
4: counter ← 0
5: if expectedCutNode(state) then
6:    for movesSearched = 1, ..., M do
7:       move ← moveIndex(movesSearched)
8:       value ← −MultiCut(successor(state, move), −β, −α, depth − 1 − R)
9:       if value ≥ Beta then
10:         counter ← counter + 1
11:         if counter == C then
12:            return β
13: for move ← NextMove() do
14:    value ← −MultiCut(successor(state, move), −β, −α, depth − 1)
15:    score ←max(value, score)
16:    if score > α then
17:       alpha ← score
18:       if score ≥ Beta then
19:          return score
20: return score
```

6.1.6 History Pruning/Late Move Reduction

History Pruning, also known as Late Move Reduction, is a forward pruning technique that has been described and discussed extensively in internet computer Chess forums[1]. History Pruning/Late Move Reduction assumes that good move ordering is done during search, and therefore a Beta cutoff either occurs after the first few moves searched, or not at all. The first few moves are then searched to the full depth, and any remaining move is searched with reduced depth unless the score returned is deemed interesting, such as the score being greater than Alpha.

There are different implementations of identifying moves to search with reduced depth. For example, the strong open source Chess program FRUIT, which implements

[1]Two popular forums are the Winboard Forum - http://wbforum.vpittlik.org and TalkChess - http://www.talkChess.com/forum

Pseudocode 8 LateMoveReduction($state$, α, β, $depth$)

```
1: if depth == 0 or isTerminal(state) then
2:    return Evaluate(state)
3: score ← −∞
4: movesSearched ← 0
5: for move ← NextMove() do
6:    if (movesSearched ≥ MinimumFullDepthMoves) and (depth ≥ Mini-
      mumDepthForLMR) and ValidForPruning(move) then
7:       newDepth ← depth − 1−depthReduction(state)
8:    else
9:       newDepth ← depth − 1
10:   value ← −LateMoveReduction(successor(state, move), −β, −α, newDepth)
11:   score ← max(value, score)
12:   movesSearched ← movesSearched + 1
13:   if score > α then
14:      alpha ← score
15:      if score ≥ Beta then
16:         return score
17: return score
```

History Pruning/Late Move Reduction, counts the number of times a move has failed high and failed low in previous searches, and moves that have a high fail high to fail low ratio are not reduced. In addition, FRUIT does not reduce the search depth if the board position is in check or the node is a PV node. As mentioned before, there is no standard implementation of History Pruning/Late Move Reduction and there are several Chess programs that use other forms of threat detection, like positional threat and evaluation data, to decide whether a move should be searched with reduced depth.

There has been a lot of discussion on History Pruning/Late Move Reduction, but surprisingly there has not been any scientific papers that show the effectiveness of History Pruning/Late Move Reduction in games. RankCut shares some similarities with History Pruning/Late Move Reduction, and later in this chapter we discuss the similarities and differences between History Pruning/Late Move Reduction and RankCut.

6.2 Preliminaries

We recall some properties of Alpha-Beta search algorithms and several definitions which will be relevant to this chapter. Alpha-Beta search square roots the effective branching factor compared to the Minimax algorithm when the nodes are evaluated in an optimal or near optimal order [Knuth and Moore, 1975]. A reduced-depth search is a search with depth $d - r$, where d is the original search depth and r is the amount of depth reduction. The higher the value of r, the more errors the reduced-depth search makes due to the horizon effect [Berliner, 1974]. However, a reduced-depth search will result in a smaller search tree and therefore a compromise has to made between reduced search effort and the risk of making more search errors.

Although it is technically more correct to refer to the value of a node (and its corresponding position) in the Minimax framework, we will use the term "value of a move" to refer to the value of the node as the result of that move, as it is conceptually easier to describe how to use move order information during search in this manner. In later sections, we will therefore use $v(m_i)$ to refer the value of the board after making move i, instead of the more correct notation of v_i.

6.3 Is There Anything Better?

In section 6.1, we described various forward pruning techniques currently used successfully in game-playing programs such as Null-Move Pruning, ProbCut, Futility Pruning, Razoring, History Pruning/Late Move Reduction and N-best selective search. Most of these techniques can be seen as using the same principle of locally testing the probability of a node scoring outside the Alpha or Beta bound, and pruning the nodes that have a high probability of failing high or low. In other words, for a node n with k moves,

$m_1 \ldots m_k$, current pruning techniques test $p(v(m_i) \geq \beta)$ and $p(v(m_i) \leq \alpha)$, where $v(m_i)$ is the Minimax evaluation of the board position after making move i, and prune if either of the probabilities is high. For example, the null move heuristic decides to prune m_k if the search returns a score greater than Beta after making a null move, as it assumes that doing nothing is generally bad for the player to move. This is equivalent to saying that $p(v(m_k) \geq \beta)$ is high and the program can prune the node while taking little risk. ProbCut extends this by also considering that if $p(v(m_k) \leq \alpha)$ is high, the program should fail low and return Alpha. However, the fact that m_k is the k^{th} move considered is not used by existing techniques.

Information on move ordering can be beneficial when making forward pruning decisions. For example, consider the same scenario in which we are deciding whether or not to prune m_k, and the first move m_1 has remained as the current best move, while $v(m_1) > v(m_2) > v(m_3) \ldots > v(m_{k-2}) > v(m_{k-1})$. Furthermore, if we know that, under similar conditions in representative games, no moves ranked k or higher have ever been found to be better, then it is unlikely that m_k will return a score better than the score of the current best move. This means m_k can be be pruned with a small risk of error based on move order information.

One pruning method that utilizes ordering is N-Best Selective Search. In N-Best Selective Search, only the N best moves, as ranked by an evaluation function, are actually searched. This heuristic requires the move ordering to be able to rank the best move consistently within the top N moves. However, the simple N-best selective search either introduces too many errors or prunes too little, depending on how N is set. For example, consider a node where the best move changes every time a new move is searched. In this case, move ordering is evidently not performing well and pruning after a small number of moves is likely to introduce errors. One easy, but incorrect, solution would be to

increase N to a high value where such occurrences are rare. However, if we now consider a node where the best move has not changed since the first move, and the scores of the subsequent moves are decreasing in a monotonic fashion, then it is likely that a high value of N is too conservative for this particular case, and forward pruning opportunities have been lost.

A better way of factoring in move ordering when pruning is by considering the probability $\Pi_x(\vec{f_i}) = p(v(m_i) > x \mid \vec{f_i})$, where $\vec{f_i} = (f(m_1), \ldots, f(m_{i-1}))$ are the salient features of the previous moves considered, and x is an adaptive bound. In our experiments, x is set to v_{best}, the score of the current best move. However, x can also be set to a value like Alpha, or a variable bound so as to minimize the risk of error. For brevity, we use $\Pi(\vec{f_i})$ to represent $\Pi_{v_{best}}(\vec{f_i})$. Good features allow $\Pi(\vec{f_i})$ to identify when moves are unlikely to affect the final score, and examples include the current move number and the scores of prior moves. So when performing forward pruning, the probability $\Pi(\vec{f_i})$ gives the likelihood of m_i returning a score better than the current best move, and if it is below a certain threshold, we can prune this move as it is unlikely to affect the final score. This approach is more adaptive than the static N-best selection search.

Since good move ordering is essential to achieving more cutoffs when using Alpha-Beta search [Knuth and Moore, 1975], heuristics like the History Heuristic [Schaeffer, 1989] and the Killer Heuristic [Akl and Newborn, 1977], together with domain dependent knowledge are used to improve move ordering. Good move ordering is therefore usually available in practical implementations, and is another good reason to consider move order in any forward pruning decision. This observation is not new – Björnsson and Marsland [Björnsson and Marsland, 2000a] mention this insight, but restrict the application to only the possibility of failing high, or $p(v(m_i) > \beta \mid v(m_1) < \beta, \ldots, v(m_{i-1}) < \beta)$. Moriarty and Miikkulainen [Moriarty and Miikkulainen, 1994]

and Kocsis [Kocsis, 2003, Chapter 4] also considered pruning nodes while taking move ordering into consideration by using machine learning techniques like neural networks to estimate the probability. However, the results of these experiments have not been conclusive and more research in their effectiveness is needed.

6.4 RankCut

In this section, we introduce a new domain independent forward pruning method called RankCut. We later show its effectiveness by implementing it in the open source Chess programs, CRAFTY[2] and TOGA II[3].

6.4.1 Concept

In the previous section, we suggested considering the probability $\Pi(\vec{f_i})$ when forward pruning. However, if the moves are ordered and $\Pi(\vec{f_i})$ is low, then the remaining moves m_j, where $j > i$, should also have low probabilities $\Pi(\vec{f_j})$. Testing each probability $\Pi(\vec{f_i})$ is thus often redundant, and RankCut considers instead the value of $\Pi'(\vec{f_i}) = p(\max\{v(m_i), v(m_{i+1}), \ldots, v(m_k)\} > v_{best} \mid \vec{f_i})$, where k is the total number of legal moves of the current node. RankCut can be thought of as asking the question "Is it likely that any remaining move is better than the current best move?". These probabilities are estimated off-line by using the relative frequency of a better move appearing and can be represented by x/y where x is the number of times a move m_j, where $j \geq i$, returns a score better than the current best when in the state $\vec{f_i}$, and y is the total number of

[2] Available at ftp://ftp.cis.uab.edu/pub/hyatt
[3] Available at http://www.uciengines.de/UCI-Engines/TogaII/togaii.html

instances of the state $\vec{f_i}$, regardless of whether or not the best move changes. This off-line procedure of collecting the statistics requires modifying the game-playing program to store the counts, and then playing a number of games under the same (or longer) time controls expected.

One potential problem is that RankCut assumes the statistics of $\Pi(\vec{f_i})$ collected without forward pruning remain the same when forward pruning. While our experiments indicate that the assumption is reasonable for practical purposes, more research is needed to understand how RankCut still works despite this simplifying assumption. We refer the reader to Chapter 9 for more discussions on this issue.

RankCut tests $\Pi'(f_{m_i}) < t$ for each move, where $t \in (0, 1)$ is user-defined. If true, RankCut does not prune the move, but instead does a reduced-depth search and returns the score of that shallow search. The full-width nature of the reduced-depth search helps to retain tactical reliability while reducing search effort. RankCut is domain independent as it does not require any game logic and is easily added to an Alpha-Beta search as shown in Pseudocode 9.

The main modifications to the Alpha-Beta search are the computations of $\vec{f_i}$ (Line 7) and tests of $\Pi'(\vec{f_i})$ (Line 8). Prior to making a move, RankCut tests if $pruneRest$ is set or $\Pi'(\vec{f_i}) < t$. If either is true, RankCut makes a reduced-depth search. Otherwise, the usual Alpha-Beta algorithm is executed.

Note that RankCut typically uses the results of a depth-reduced search to perform Alpha-Beta cutoffs; although this is not theoretically sound, this has worked relatively well in our experiments in Chess programs. Alternatively, there is an optional boolean variable $RankCutReSearch$ (Line 12) that can be set to require a re-search without depth reduction *if* the reduced-depth search returns a score greater than Alpha. As this forces a re-search without depth reduction, this option is a compromise between more

Pseudocode 9 RankCut($state$, α, β, $depth$)

```
 1: if depth == 0 or isTerminal(state) then
 2:    return Evaluate(state)
 3: pruneRest ← false
 4: score ← −∞
 5: for move ← NextMove() do
 6:    r ← 0
 7:    Compute(f_i)
 8:    if pruneRest or (Π'(f_i) < t) then
 9:       r ← depthReduction(state)
10:       pruneRest ← true
11:    score ← −RankCut(successor(state, move), −β, −α, depth − 1 − r)
12:    if RankCutReSearch and (score > α) and pruneRest then
13:       score ← −RankCut(successor(state, move), −β, −α, depth − 1)
14:    score ← max(value,score)
15:    if score > α then
16:       pruneRest ← false
17:       α ← score
18:    if score ≥ β then
19:       return score
20: return score
```

accurate search results and faster search times. The experiments in this chapter did not enable this option.

One potential problem is that RankCut assumes the statistics of $\Pi(\vec{f_i})$ collected without forward pruning remain the same when forward pruning. Nevertheless, our experiments indicate that the assumption is reasonable for practical purposes.

6.4.2 Implementation in CRAFTY

CRAFTY is a very strong open-source Chess engine and its rating is about 2600 on a 1.2 GHz Athlon with 256MB of RAM when tested independently by the Swedish Chess Computer Association, or SSDF[4]. CRAFTY uses modern computer Chess techniques

[4]Details at http://web.telia.com/∼u85924109/ssdf/

such as bitboards, 64-bit data structures, NegaScout search, Killer Move heuristics, Static Exchange Evaluation, Quiescence search, and selective extensions. We incorporated RankCut into CRAFTY Version 19.19, which features Null-Move Pruning and Futility Pruning, and ran all experiments on a PowerMac 1.8GHz. We will differentiate between the two versions of CRAFTY when needed in our discussions by calling them ORIGINAL CRAFTY and RANKCUT CRAFTY.

CRAFTY has 5 sequential phases of move generation – (1) principal variation from the previous search depth during iterative deepening, (2) capture moves sorted based on the expected gain of material, (3) Killer moves, (4) at most 3 History moves, and (5) the rest of the moves. We modified CRAFTY so that at phase 4 or the History move phase, it would continue sorting remaining moves according to the History Heuristic until no more suitable candidate was found.

During testing, we discovered that the probability of a better move appearing for moves generated in phases before the History Move phase is always too high to trigger a pruning decision. RANKCUT CRAFTY saves computational effort by only starting to forward prune when move generation is in the History move phase.

The probabilities $\Pi'(\vec{f_i})$ were calculated by collecting the statistics from 50 self-play games, each with a randomly-chosen opening, where CRAFTY played against itself in a time control of 80 minutes per 40 moves. The pruning threshold t and the amount of depth reduction in the shallow search were conservatively set at 0.75% and 1 ply respectively. As we calculated the relative frequencies with a small set of 50 games, we use the probabilities $\Pi'(\vec{f_i})$ only if 1,000 or more instances of $\vec{f_i}$ were seen to ensure that the statistics are reliable. The following features $\vec{f_i}$ were used:

1. Current depth of search

2. Whether or not player is in check
3. Current move number
4. Number of times the best move has changed
5. Difference between the score of the current best move from the given Alpha bound (discretised to 7 intervals)
6. Difference between the score of the last move from the current best move (discretised to 7 intervals)
7. Phase of the move generation (History moves or Remaining moves)

CRAFTY uses search extensions to explore promising positions more deeply. RANKCUT CRAFTY therefore does not reduce the search depth even when $\Pi'(\vec{f_i}) < t$ if CRAFTY extends the search depth as CRAFTY has signaled that the current node needs more examination. RANKCUT CRAFTY is also set to forward prune only nodes that have search depth, defined as the length of a path from the current node to the leaf nodes, greater or equal to 7. This is because move ordering tends to be less reliable when iterative deepening is initially searching the first few depths. Furthermore, when searching higher search depths, move ordering also becomes less reliable when the search is further from the root.

Test Suites

We tested RANKCUT CRAFTY with all 2,180 positions from the tactical Chess test suites ECM, WAC and WCS (see Appendix D) by searching to fixed depths of 8, 10, and 12 plies respectively. Table 6.1 shows the results of these experiments and compares them

	Original			RankCut				
Test Suite	#Nodes	Avg Time (s)	#Solved	ΔNodes	Δ%	ΔTime	Δ%	ΔSolved
ECM-08	1,170,566,012	1.68	549	-104,561,591	-8.9%	-0.03	-1.98%	-2
ECM-10	11,641,894,779	15.43	620	-3,636,256,257	-31.2%	-3.98	-25.80%	-7
ECM-12	88,924,232,803	120.92	678	-36,761,238,388	-41.3%	-47.34	-39.15%	-13
WAC-08	160,052,139	0.63	289	-11,554,559	-7.2%	-0.02	-2.87%	-2
WAC-10	1,961,241,110	6.85	294	-667,242,604	-34.0%	-1.26	-18.32%	0
WAC-12	13,361,000,868	66.92	297	-4,763,207,469	-35.7%	-35.95	-53.72%	-1
WCS-08	914,031,896	1.11	840	-96,470,205	-10.6%	-0.06	-4.96%	1
WCS-10	9,534,443,036	10.75	863	-2,234,703,349	-23.4%	-2.32	-21.60%	1
WCS-12	77,536,489,398	87.36	873	-212,586,355	-36.1%	-27.36	-31.31%	-2
Sum-08	2,244,650,047	1.27	1678	-212,586,355	-9.5%	-0.04	-3.23%	-3
Sum-10	23,137,578,925	12.10	1777	-6,538,202,210	-28.3%	-2.84	-23.50%	-6
Sum-12	179,821,723,069	98.08	1848	-69,528,669,092	-38.7%	-36.60	-37.31%	-16

Table 6.1: Comparison of performance in test suites with fixed depths

with the results of the ORIGINAL CRAFTY. The last three rows of Table 6.1 also show the combined results of the three test suites.

The absolute standard error [Heinz, 1999] of n test positions with k correct solutions is $SE = n \times \sqrt{p \times (1-p)/n}$ where $p = k/n$. The standard error allows us to ascertain whether the errors introduced by the pruning method is within 'statistical' error bounds. The combined results of all three suites for ORIGINAL CRAFTY in Table 6.1 shows that the standard error for "Sum-08", "Sum-10" and "Sum-12" are $SE_8 = 19.6$, $SE_{10} = 18.1$ and $SE_{12} = 16.7$, respectively. RANKCUT CRAFTY, however, solves only 3, 6 and 16 fewer test positions while searching less nodes (with the difference from ORIGINAL CRAFTY denoted by Δ columns). Hence all the results of RANKCUT CRAFTY are within one standard error of the results of ORIGINAL CRAFTY.

We also tested using the LCT II test (see Appendix D), a set of 35 positions divided into positional, tactical and end-game positions. The LCT II estimates an ELO rating for the program based on the solution times. Both ORIGINAL CRAFTY and RANKCUT CRAFTY solved the same 28 out of 35 problems, but due to faster solutions, RANKCUT CRAFTY obtained a rating of 2635 ELO, whereas ORIGINAL CRAFTY was estimated at 2575 ELO. During the test, ORIGINAL CRAFTY searched to an average of 15.7 plies,

whereas RANKCUT CRAFTY was able to search to an average of 16.5 plies, almost 1 ply deeper on average.

Match Games against Original

We played RANKCUT CRAFTY against ORIGINAL CRAFTY with a set of 62-openings, consisting of 32 openings used in [Jiang and Buro, 2003], 10 Nunn positions, and 20 Nunn-II positions (see Appendix D) under the time control of 40 moves/40 minutes. This 124-game match resulted in a decisive result of +40 -10 =74 or 62.0% in favor of RANKCUT CRAFTY, or 40 wins, 10 losses and 74 draws. We will outline the statistical tools used in [Heinz, 2001b, Heinz, 2001a] to show that this result is statistically significant.

The standard error of a scoring rate $w = x/n$ is $s(w) = \sqrt{w \times (1-w)/n}$, where $x \leq n$ is the number of points scored in a match of n games. Let $z_{\%}$ denote the upper critical value of the standard $N(0,1)$ normal distribution for a desired $\%$-level statistical confidence, where $z_{90\%} = 1.645$ and $z_{95\%} = 1.96$. Then $w \pm z_{\%} \times s(w)$ is the $\%$-level confidence interval on the real winning probability of a player with scoring rate w.

We can now derive the $\%$-level confidence lower bound on the difference in real winning probability between two players of scoring rates $w_1 = x_1/n_1$, and $w_2 = x_2/n_2$, where $w_1 \geq w_2$. Let $l_{\%} = (w_1 - z_{\%} \times s(w_1)) - (w_2 + z_{\%} \times s(w_2))$. If $l_{\%} > 0$, we are $\%$-level confident that the player with the higher scoring rate is indeed stronger than the other.

From above, $l_{95\%} \approx 0.099$, and therefore we can claim that RANKCUT CRAFTY is indeed stronger than ORIGINAL CRAFTY with 95% confidence.

Match Games against FRUIT 2.1

FRUIT is one of the strongest Chess programs in the world. FRUIT 2.2.1 finished in second place at the 2005 World Computer Chess Championships [Björnsson and van den Herik, 2005] and obtained a rating of more than 2800 when tested by SSDF. FRUIT was an open source Chess engine until Version 2.1[5]. We tested the ORIGINAL CRAFTY and RANKCUT CRAFTY against FRUIT 2.1 with the 32-openings used in [Jiang and Buro, 2003] under blitz time controls of 2 min + 10 sec/move. ORIGINAL CRAFTY lost to FRUIT by +11 -39 =14 or 28.1% and RANKCUT CRAFTY lost to FRUIT by +15 -40 =9 or 30.5%. In addition, we played 20 Nunn-II opening positions under 40 moves/40 minutes. ORIGINAL CRAFTY lost to FRUIT by +2 -26 =12 or 20.0% and RANKCUT CRAFTY lost to FRUIT by +5 -28 =7 or 21.25%.

These results suggest that the performance gains of RANKCUT CRAFTY extend to games against other Chess engines.

6.4.3 Implementation in TOGA II

To further validate RankCut's performance, we implemented RankCut in TOGA II 1.2.1A, which is based on FRUIT 2.1. TOGA II is also a very strong open source Chess engine and while it is not officially rated by SSDF, it is stated to be stronger than FRUIT 2.1, with improved forward pruning and evaluation functions. TOGA II is written in C++ and uses similar modern computer Chess techniques as CRAFTY, but as a derivative of FRUIT, TOGA II also implements History Pruning/Late Move Reduction (Section 6.1.6). As before, we will differentiate between the two versions of TOGA II in subsequent discussions when needed by naming them ORIGINAL TOGA II and RANKCUT

[5]Source code is available at http://arctrix.com/nas/fruit/

TOGA II. Both RANKCUT TOGA II and ORIGINAL TOGA II had History Pruning/Late Move Reduction enabled.

We collected statistics for RankCut in a similar manner. The probabilities $\Pi'(\vec{f_i})$ were calculated by collecting the statistics from 50 self-play games, each with a randomly-chosen opening, where TOGA II played against itself in a time control of 80 minutes per 40 moves. All options of ORIGINAL TOGA, such as History Pruning/Late Move Reduction, were unchanged during this training phase. The pruning threshold t and the amount of depth reduction in the shallow search were again conservatively set at 0.75% and 1 ply respectively. We used the probabilities $\Pi'(\vec{f_i})$ only if 1,000 or more instances of $\vec{f_i}$ were seen to ensure that the statistics are reliable. Due to the different implementations between TOGA II and CRAFTY, the same features used in RANKCUT CRAFTY could not be entirely replicated. Instead, the following features $\vec{f_i}$ were used:

1. Current move number
2. Number of times the best move has changed
3. Difference between the score of the current best move and the given Alpha bound (discretised to bins of 100 points)
4. Difference between the score of the last move and the current best move (discretised to bins of 100 points)
5. Phase of the move generation
6. Reduction in depth based on tactical heuristics[6]

[6]Most search depths will be reduced by one, except for cases where the player had only one legal move, or if the last move was advancing a passed pawn.

Note that unlike our implementation in CRAFTY, we do not use the *depth* feature. If RankCut is able to prune accurately without the *depth* feature, we consider this to be beneficial as it removes the need for RankCut to have seen a move at a particular depth during training to be able to assess whether or not to prune.

TOGA II has different phases of move generation, depending on whether the position is in check, the search is quiescence, and for all other types of searches which we term "normal" searches. As RANKCUT TOGA II does not prune when the position is in check, or when the search is in quiescence, we are only concerned with the phases of move generation in "normal" searches. In "normal" searches, TOGA II has 5 phases of move generation - (1) Killer moves in the transposition table, (2) "good" captures, (3) "bad" captures, (4) Killer moves not in the transposition table and lastly, (5) "quiet" moves. Within these move generation phases, moves are sorted using heuristic values such as history values and other heuristic evaluation functions.

TOGA II performs search extensions to explore promising or tactical positions more deeply. The amount of search extension is captured by the reduction in depth based on tactical heuristics. RANKCUT TOGA II decides whether or not to prune regardless of the search extensions, as long as $\Pi'(\vec{f_i}) < t$. RANKCUT TOGA II is also set to forward prune only nodes that have search depth greater or equal to 7.

Results

We played RANKCUT TOGA II against ORIGINAL TOGA II using 52 openings, consisting of 32 openings used in [Jiang and Buro, 2003], and 20 Nunn-II positions under time control of 40 moves/40 minutes. RANKCUT TOGA II won ORIGINAL TOGA II by +42 -19 =43, or approximately 61.06%. This result is statistically significant with 95% confidence.

We also played RANKCUT TOGA II and ORIGINAL TOGA II against HIARCS 10 on a set of 20 Nunn-II positions under standard time control 40 moves/40 minutes. HIARCS 10 is a strong commercial Chess program rated 2853 by SSDF[7], and is known for its human-like playing style. ORIGINAL TOGA II won HIARCS 10 by +17 -11 =12 or 57.5% and RANKCUT TOGA II won HIARCS 10 by +21 -6 =13 or 68.75%. This suggests that the improvement that RankCut gives extends to game performance against other Chess engines.

6.4.4 Related Work

As noted in section 6.3, most of the current forward pruning techniques, such as Null-Move Pruning, ProbCut, Futility Pruning, Razoring, and N-best selective search, assess whether or not to prune based on different criterions.

However, History Pruning/Late Move Reduction (Section 6.1.6) also assumes that good move ordering is done during search, and searches moves with reduced depth if it is deemed to be unlikely to affect the final score. While both algorithms share this basic idea, there are two major differences between History Pruning/Late Move Reduction and RankCut.

First of all, RankCut considers the *rank* of a move, and since moves are assumed to be sorted in descending order of quality, RankCut is therefore able to prune all remaining moves after a move is deemed to be unlikely to affect the final score. In contrast, History Pruning/Late Move Reduction considers each move in isolation after a fixed number of initial moves are searched to full depth.

In addition, RankCut decides whether or not to prune a move based on off-line computations whereas History Pruning/Late Move Reduction uses on-line computations. By

[7]Rating on 10 Sep 2006

using off-line computations, RankCut is able to see how often a move is unlikely to affect the final score across multiple games. In contrast, History Pruning/Late Move Reduction makes a decision to prune based on information from the start of the game up to the current board position. Furthermore, the more features that are used to estimate the probability of a better move appearing, the more training data is required. Using on-line computations therefore limit the number of features used to predict whether or not a better move will appear. For example, FRUIT uses only the type of piece, the from square and the to square as features for History Pruning.

Despite the differences, both algorithms have been shown to be effective in practical Chess programs. Furthermore, RankCut seems to improve game-playing performance when implemented alongside History Pruning/Late Move Reduction in TOGA II. This suggests that each algorithm is able to make effective pruning decisions that the other algorithm is unable to make.

6.4.5 Implementation Details

In our implementations, data representations for $\vec{f_i}$ and $\Pi'(\vec{f_i})$ are straightforward. While we are not limited to just the following methods, we will briefly describe how we represented them in our implementations of RANKCUT CRAFTY and RANKCUT TOGA II.

As $\vec{f_i}$ is a vector of features of past moves, it can be implemented using an array or a set of variables. `Compute`($\vec{f_i}$) can then be performed incrementally for each move. For example, if the current move number is a chosen feature, then `MoveNumber++` in C pseudocode suffices to incrementally update the feature for the next move.

$\Pi'(\vec{f_i})$ can also be represented using arrays or a hash table. For example, if the current move number is the only feature used, $\Pi'(\vec{f_i})$ can be represented using an array

with `double P[MAX_MOVES]` in C pseudocode, or using a hash table that maps $\vec{f_i}$ to $\Pi'(\vec{f_i})$.

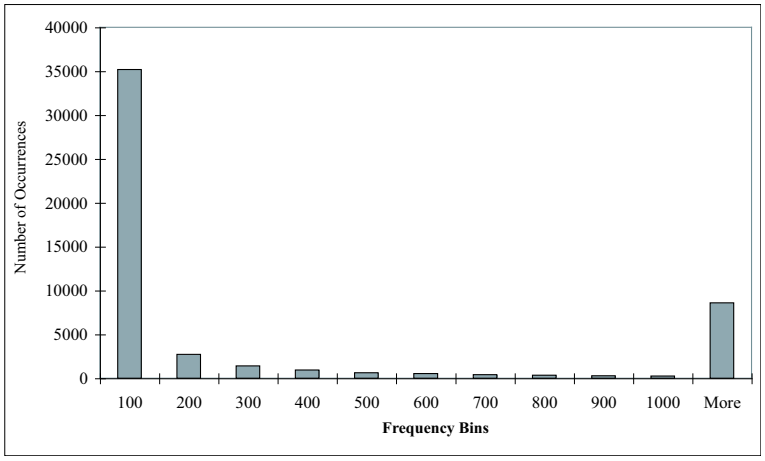


Figure 6.1: Histogram of the number of features collected for RANKCUT CRAFTY with $t < 0.75\%$. The x-axis consists of frequency bins of features. Each bin contains the count of features that are seen the number of times between the previous bin and the current bin, and the y-axis is the number of features within the frequency bin.

As RankCut prunes only if $\Pi'(\vec{f_i}) < t$, we only need to store all $\vec{f_i}$ such that $\Pi'(\vec{f_i}) < t$ if we do not update the frequencies of moves on-line. By deciding on a threshold t beforehand, our implementations of RankCut were able to extract the $\vec{f_i}$ with the appropriate probabilities. As we can see in the histograms shown in Figures 6.1 and 6.2, the number of features that are seen more than 1,000 times with $t < 0.75\%$ is relatively small - approximately 10,000 for RANKCUT CRAFTY and 15,000 for RANKCUT TOGA II. This resulted in our implementations of RankCut requiring less than 1 MB of memory to store $\vec{f_i}$.

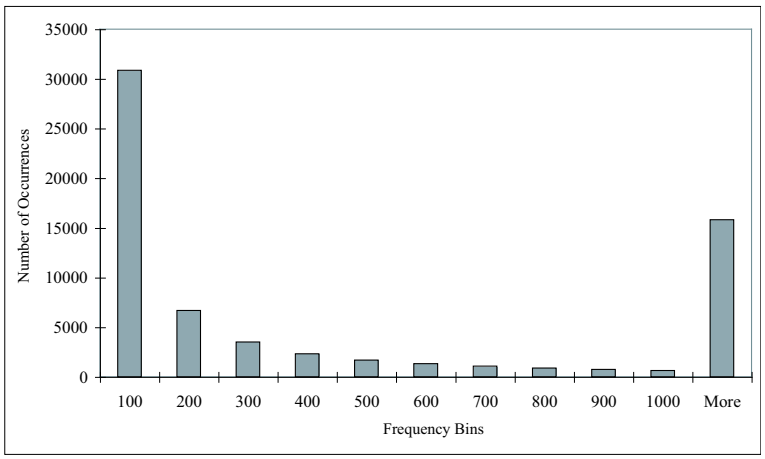


Figure 6.2: Histogram of the number of features collected for RANKCUT TOGA II with $t < 0.75\%$. The x-axis consists of the frequency bins of features. Each bin contains the count of features that are seen the number of times between the previous bin and the current bin, and the y-axis is the number of features within the frequency bin.

6.5 Chapter Conclusions

In this chapter, we introduce RankCut, which is a domain-independent forward pruning technique that exploits the move ordering that current game-playing programs typically perform for efficiency in Alpha-Beta search. RankCut can be implemented within an existing Alpha-Beta search, and we successfully implemented RankCut in CRAFTY, an open-source Chess-playing program. Compared to the unmodified version of CRAFTY, RankCut reduces the game-tree size by 10%-40% for search depths 8-12 while retaining tactical reliability. After playing a 124-game match against the original CRAFTY, RANKCUT CRAFTY had a winning percentage of 62%. This is despite the fact that CRAFTY also features Null-Move Pruning and Futility Pruning.

We also implemented RankCut in TOGA II and show that RankCut improves the performance of TOGA II, even though TOGA II features Null-Move Pruning, Futility Pruning and History Pruning. RANKCUT TOGA II had a winning percentage of 61.06% in a 104-game match against ORIGINAL TOGA II.

The simplicity of RankCut makes implementation in various games easy, and it can even be implemented on top of existing forward pruning techniques.

Chapter 7

Player to Move Effect

This chapter is based partially on our article "Properties of Forward Pruning in Game-Tree Search" presented at the Twenty-First National Conference on Artificial Intelligence (AAAI-06).

The Alpha-Beta algorithm [Knuth and Moore, 1975] is the standard approach used in game-tree search to explore all combinations of moves to some fixed depth. However, even with Alpha-Beta pruning, search complexity still grows exponentially with increasing search depth. To further reduce the number of nodes searched, practical game-playing programs perform forward pruning [Marsland, 1986, Buro, 1995a, Heinz, 1999], where a node is discarded without searching beyond that node if it is believed that the node is unlikely to affect the final Minimax value of the node. As all forward pruning techniques inevitably have a non-zero probability of making pruning errors, employing any forward pruning technique requires a compromise between accepting some risk of error and pruning more in order to search deeper.

However, the effects of different pruning errors vary. We can observe this by considering a Chess game where White is to move and winning. During Minimax search, most of White's moves will be good enough to maintain the advantage of the game, so

it is acceptable to prune away most of White's moves. However, it would be foolish for White to lose the advantage by making a move that allows a tactical counter-attack by Black, and White should therefore be concerned that no such counter-tactics by Black exists for all the moves he considers. This example highlights the asymmetric effects of pruning errors–Pruning errors in White-to-move positions, assuming that most of the better moves have been considered, are tolerated more readily than pruning errors in Black-to-move positions during search.

Forward pruning techniques should therefore consider how pruning errors propagate in game-tree search. In this chapter we show that the severity of forward pruning errors is asymmetric with respect to the player to move; the error is likely to be more severe when pruning on the opponent's moves rather than the player's own moves [Lim and Lee, 2006a]. This effect arises because pruning errors, when pruning exclusively on the children of Max nodes, cannot cause a poor move to be deemed better than a good move whose subtree does not contain errors; however, this is not the case when pruning exclusively on the children of Min nodes. This suggests that to do well, pruning should be done more aggressively on the player's own move and less aggressively on the opponent's move.

7.1 Theoretical Analysis

In this section, we show that forward pruning errors are propagated differently depending on the player to move. We first state a lemma showing how the different pruning errors affect Minimax results:

Lemma 1. *Assume that we are performing forward pruning only on the children of Max nodes throughout the tree. Then, for any unpruned node* u, $score_{Max}(u) \leq score_{*}(u)$,

where $score_{Max}(u)$ is the score of the algorithm that only forward prunes children of Max nodes. Conversely, if we are performing forward pruning only on the children of Min nodes, then for any unpruned node u, $score_{Min}(u) \geq score_{*}(u)$, where $score_{Min}(u)$ is the score of the algorithm that only forward prunes the children of Min nodes.

Proof. By induction on the depth d of node u in the tree, where d is defined as the number of nodes in a path from u to a leaf node.

Base Case:

$$score_{Max}(u) = utility(u) = score_{*}(u) \text{ if } d = 1$$

Assume that the inductive statement is true for $d = n$, and consider the case when $d = n + 1$.

If it is Max's turn to move at node u,

$$\begin{aligned} score_{Max}(u) &= \max\{score_{Max}(subset(child(u)))\} \\ &\leq \max\{score_{Max}(child(u))\} \\ &\leq \max\{score_{*}(child(u))\}, \text{ by induction} \end{aligned}$$

If its Min's turn to move at node u,

$$\begin{aligned} score_{Max}(u) &= \min\{score_{Max}(child(u))\} \\ &\leq \min\{score_{*}(child(u))\}, \text{ by induction} \end{aligned}$$

The proof of the converse statement is similarly done using induction. □

Note that the the inequalities are strict if and only if forward pruning discards a move erroneously, e.g. the child with the highest Minimax score is discarded in a Max node.

The standard approach for game-playing is to use the result of the game-tree search with the root node representing the current board position. We assume without loss of generality that the root node is a Max node.

Theorem 1. *Assume there are b moves at the root node and that there is a strict ordering based on the utility of the moves such that $score_*(u_i) > score_*(u_j)$ if $1 \leq i < j \leq b$ and u_i is the i^{th} child.*

If forward pruning is applied only to the children of Max nodes, then $\forall i$ such that no pruning error occurs in the subtree of u_i, i.e. $score_{Max}(u_i) = score_*(u_i)$, the new rank i' based on $score_{Max}$ has the property $i' \leq i$.

The converse holds if forward pruning is applied only to the children of Min's nodes, i.e., $\forall i$ such that no pruning error occurs in the subtree of u_i, i.e. $score_{Min}(u_i) = score_*(u_i)$, the new rank i' based on $score_{Min}$ has the property $i' \geq i$.

Proof. We will only prove the first statement, as the proof of the converse statement is similar. Assume, on the contrary, that $i' > i$. This implies that $\exists j > i$ such that $score_*(u_j) < score_*(u_i)$, but the new ordering j' based on $score_{Max}$ is $j' < i'$ and $score_{Max}(u_j) > score_{Max}(u_i)$. But $score_{Max}(u_j) \leq score_*(u_j)$ by Lemma 1, and $score_*(u_j) < score_*(u_i) = score_{Max}(u_i)$ which imply that $score_{Max}(u_j) < score_{Max}(u_i)$. Contradiction. □

7.2 Observing the Effect using Simulations

In order to show the significance of Theorem 1, we perform *Monte Carlo simulations* of random game-trees searched using algorithms that perform forward pruning.

In our simulations, we used game-trees of uniform branching factor 5. The values of the leaf nodes were chosen from the uniform distribution $[0, 1)$ and the root is a Max

node. We fix the number of nodes to be forward pruned – in each instance, three randomly chosen children at either Max or Min nodes were forward pruned. Unfortunately, error filtering effects that reduce error propagation to the root makes comparison more difficult; the type of pruning errors that is filtered more depends on the height of the tree. To ensure that the observed effects are not because of the filtering effect, we experimented with trees of heights ranging from four to six so that there would instances of both cases: having fewer Max nodes and having fewer Min nodes.

We recorded the rank of the move at the root (as ranked by a search without forward pruning) that was eventually chosen by the search. For each experimental setup, we ran a simulation of 10^6 randomly generated game-trees.

Figure 7.1 shows the number of times the search with forward pruning chooses a move of a particular rank. We see that when children of Max nodes are pruned erroneously, the probability of choosing a move ranked 4^{th} or 5^{th} decreases sharply towards zero; when children of Min nodes are pruned wrongly, the probability of choosing a move ranked 4^{th} or 5^{th} only tapers gradually. In other words, if we have to choose between forward pruning only the children of Max or Min nodes, and the eventual rank of the move chosen is important, we should choose to forward prune the children of Max nodes.

We also simulated more realistic game-trees, using the approach of Newborn [Newborn, 1977]. In his approach, every node in the tree receives a random number, and the value of a leaf node is the average of all random numbers in the nodes on the path from the root of the tree to the leaf node. This ensures some level of correlation between the Minimax values of sibling nodes, which is known to be non-pathological [Nau, 1982]. The random number in each node is chosen from the uniform distribution $[0, 1)$. The results with such branch-dependent leaf valued game-trees as shown in Figure 7.2 are

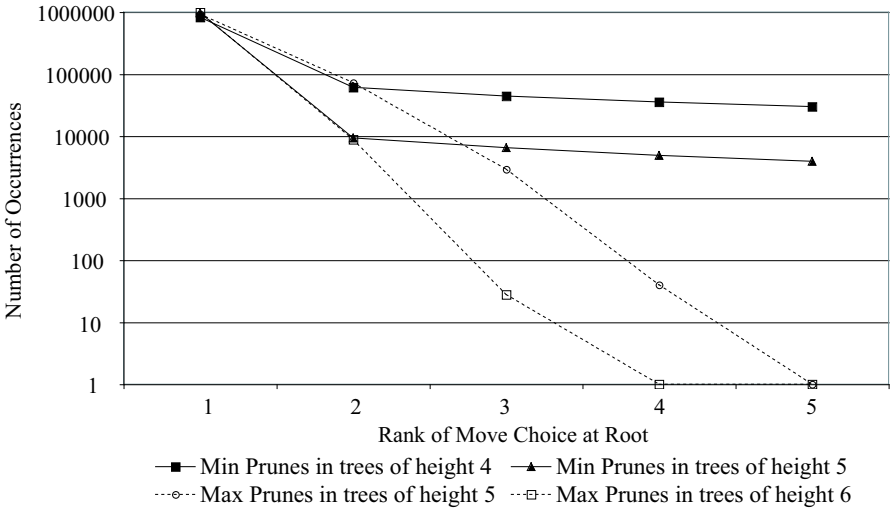


Figure 7.1: Log plot of the number of times ranked moves are chosen where either Max or Min nodes are forward pruned

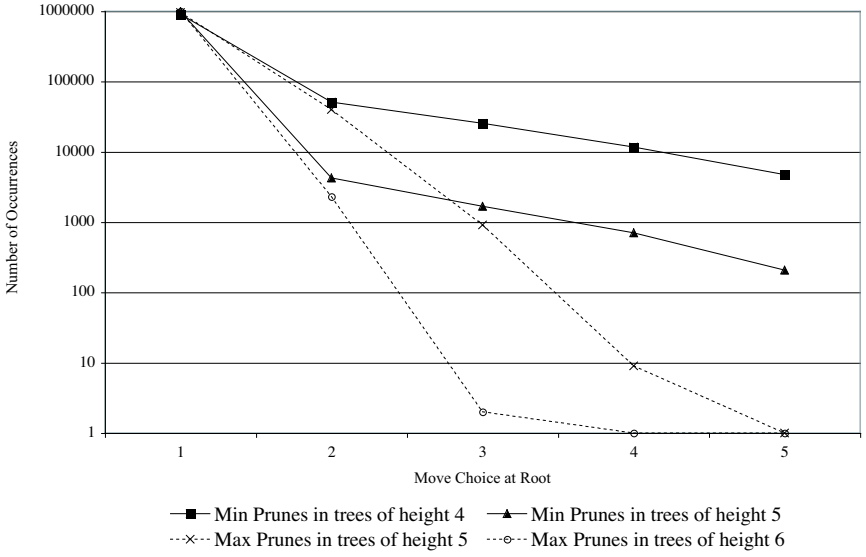


Figure 7.2: Log plot of the number of times ranked moves are chosen where either Max or Min nodes are forward pruned in game-trees with branch-dependent leaf values

similar to Figure 7.1 but slightly less pronounced.

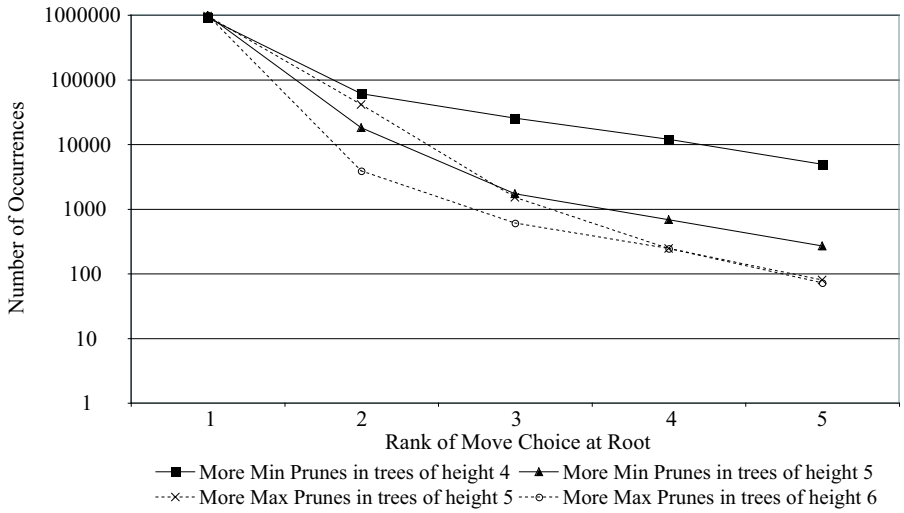


Figure 7.3: Log plot of the number of times ranked moves are chosen with unequal forward pruning on both Max and Min nodes in game-trees with branch-dependent leaf values

We may want to prune of both types of nodes in practice, if pruning on only the children of Max nodes does not provide enough computational savings. To simulate this, we ran experiments with branch-dependent leaf valued trees, where three randomly chosen children at either Max or Min nodes and one randomly chosen child of the other node type were forward pruned. Figure 7.3 shows the results of these experiments for various search depths. We see that the asymmetric effects of the severity of errors are still present. Most of the errors that resulted in a move ranked 4^{th} or 5^{th} being chosen are likely to have come from pruning children of Min nodes.

7.3 Observing the Effect using Real Chess Game-trees

The simulation results in Section 7.2 are consistent with Theorem 1. However, "real" game-trees typically have variable branching factors and heuristic leaf values whereas

the simulated game-trees have uniform branching factor and randomly generated leaf values. It therefore remains to be seen that the same effects can be observed using "real" game-trees.

To show that the player to move effect applies in "real" game-trees, we use the implementation of RankCut in the strong open-source Chess program TOGA II (Section 6.4.3). We will differentiate between the two versions of TOGA II when needed in our discussions by calling them ORIGINAL TOGA II and RANKCUT TOGA II to refer to the version of Toga II which does not implement RankCut and the one which implements RankCut respectively.

We first generated 56,167 Chess positions using ORIGINAL TOGA II, with a fixed time limit of 10 seconds per move, by starting from positions obtained from computer Chess test suites, and continuing play until the game ended.

Next, for each generated Chess positions, we use RANKCUT TOGA II to find its best move within a time limit of 10 seconds. We record the move made by RANKCUT TOGA II, but use ORIGINAL TOGA II to find the rank of the move made by RANKCUT TOGA II among all legal moves of that position. To ensure that the rank of the move is not affected by the pruning done by RANKCUT TOGA II, ORIGINAL TOGA II is made to search to the same depth as RANKCUT TOGA II did within the time limit. Two pruning schemes were tested - $t = 0.1$ on one type of nodes (Max or Min), and $t = 0.0$ on the other type of nodes. Note that even when $t = 0.0$, RANKCUT TOGA II forward prunes moves with features in which RankCut did not encounter any better moves during training.

Figure 7.4 shows the number of times RANKCUT TOGA II chooses a move of a particular rank. We see that even in real Chess game-trees, when the children of Max nodes are pruned erroneously, the probability of choosing a low ranked move decreases

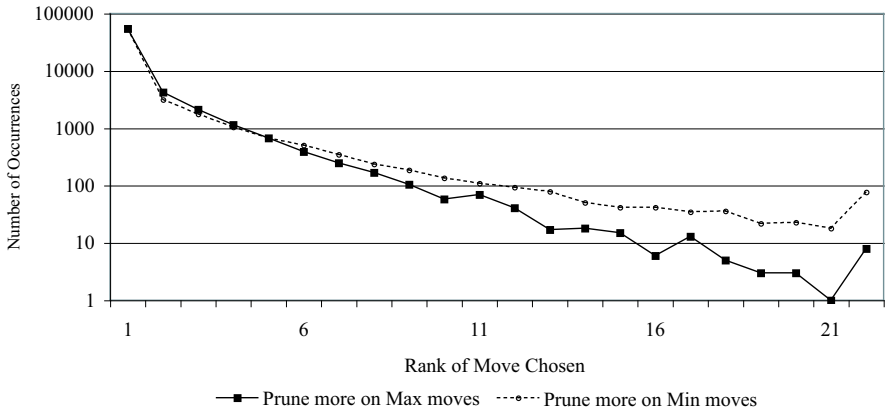


Figure 7.4: Log plot of the number of times ranked moves are chosen with unequal forward pruning on both Max and Min nodes in real Chess game-trees

more than when the children of Min nodes are pruned wrongly. In other words, if the eventual rank of the move chosen by a search is important, but we are able to tolerate a certain level of forward pruning errors, we should choose to forward prune more on the children of Max nodes than on the children of Min nodes.

7.4 Effect on Actual Game Performance

The experiments on both simulated and Chess game-trees have shown that the player to move of a node affects the propagation of forward pruning errors in terms of the rank of the move chosen by the root. We therefore expect that actual game performance will also be affected by the player to move. In other words, since the rank of the move chosen by the root is lower when forward pruning more aggressively on the children of Min nodes, the game performance of a search that forward prunes more aggressively on the children of Min nodes should also be poorer.

We tested the effect of player to move on actual game performance by running a set of matches between RANKCUT TOGA II and ORIGINAL TOGA II on 52 openings,

consisting of 32 openings used in [Jiang and Buro, 2003], and 20 Nunn-II positions under the blitz time control of 40 moves/10 minutes. RANKCUT TOGA II was modified to accept two thresholds t_{MAX} and t_{MIN}, corresponding to the threshold used to determine whether or not to forward prune when in a MAX or MIN node, respectively. We tested the asymetric Max-Min Thresholds pairs of $\{0.5\% - 0.0\%, 0.0\% - 0.5\%\}$, $\{0.75\% - 0.25\%, 0.25\% - 0.75\%\}$ and $\{1.0\% - 0.5\%, 0.5\% - 1.0\%\}$. Note that the differences in Max and Min thresholds are all 0.5%, a value high enough that we chose so that the effect of the player to move would be evident.

In our preliminary experiments with RANKCUT TOGA II, the scores achieved by the asymmetric Max-Min thresholds were statistically similar, and therefore we were unable to observe the effect of the player to move on game performance. We hypothesize that the amount of forward pruning that RankCut introduced was insufficient for the effect to manifest. Subsequently, RANKCUT TOGA II was tweaked to incorporate more forward pruning by (1) starting the forward pruning from depth 3 (instead of depth 7), and (2) all features $\vec{v}$ which were seen at least once before are used in decide whether or not to forward prune (instead of those seen at least 1,000 times).

While this made RANKCUT TOGA II forward prune more often, it also weaken RANKCUT TOGA II, which now loses to ORIGINAL TOGA II as seen in Table 7.1 - RANKCUT TOGA II with thresholds 0.00%, 0.25%, 0.5%, 0.75% and 1.00% won only 50.48%, 46.63%, 42.79%, 47.60% and 38.94% of games, respectively. Note that at $t = 0.00\%$ RANKCUT TOGA II still prunes moves that have features which never had a better move appear after it during training.

We are now able to observe the effect of the player to move of a node during forward pruning on game performance. The differences between the asymmetric pruning threshold of Max and Min players are evident - (1) $\{0.25\%$-$0.75\%\}$ won 45.19% of games

Parameters (Max-Min Thresholds)	Wins	Losses	Draws	Winning %
0.00%-0.00%	22	21	61	50.48%
0.00%-0.50%	25	26	53	49.52%
0.50%-0.00%	23	19	62	51.92%
0.50%-0.50%	16	31	57	42.79%
0.25%-0.25%	26	33	45	46.63%
0.25%-0.75%	21	31	52	45.19%
0.75%-0.25%	34	21	49	56.25%
0.75%-0.75%	11	16	77	47.60%
0.50%-0.50%	16	31	57	42.79%
0.50%-1.00%	21	21	62	50.00%
1.00%-0.50%	36	16	52	59.62%
1.00%-1.00%	11	34	59	38.94%

Table 7.1: Scores achieved by various Max-Min thresholds combinations against ORIGINAL TOGA II

whereas {0.75%-0.25%} won 56.25% of games, (2) {0.50%-1.00%} won 50.00% of games whereas {1.00%-0.50%} won 59.62%, and (3) {0.00%-0.50%} won 49.52% of games whereas {0.50%-0.00%} won 51.92% of games. For all three pairs of Max-Min thresholds combinations, the combination with a higher Max threshold won more games than the combination with a higher Min threshold.

Using the result of each Max-Min threshold pair individually does not result in a statistically significant result as outlined in Section 6.4.2. However, we are 95% confident that combinations with a higher Max threshold is stronger than ORIGINAL TOGA II, but are unable to draw that same conclusion when the Min threshold is higher.

7.5 Chapter Conclusion

In two-player perfect information games, the player can typically make only one move, so if the best move has been pruned, the game-tree search should then preferably return the second best move. Theorem 1 and our experimental results in simulated and Chess

game-trees suggest that different pruning errors relative to the player at the root node have different effects on the move quality chosen by the game-tree search. Pruning errors in children of Max nodes will not decrease the rank of moves that are correctly evaluated. This means that if the second best move is correctly evaluated but the best move is incorrectly evaluated due to pruning errors, then the game-tree search will return the second best move as the move to play. On the other hand, pruning errors in children of Min nodes can incorrectly increase the rank of moves and the game-tree search could possibly return the worst move as the move to play, even if the best move is correctly evaluated.

Experiments involving matches played between RANKCUT TOGA II and ORIGINAL TOGA II suggest that this effect might extend to actual game-playing performance. However, we were able to observe this effect when we increased the amount of forward pruning that RANKCUT TOGA II does. In addition, since the effects on the rank of the move chosen in simulations are clearly evident only in log scale, this suggests that the effect is relatively small and would explain why this effect has not been observed in applications and empirical experiments until now.

Nevertheless, the player to move effect on pruning error propagation appears to be present in game-tree search, and this suggests that all forward pruning techniques in practical settings should consider and experiment with the risk management strategy of forward pruning more aggressively on the children of Max nodes and more conservatively on the children of Min nodes.

Chapter 8

Depth of Node Effect

This chapter is modified from parts of our article "Properties of Forward Pruning in Game-Tree Search" presented at the Twenty-First National Conference on Artificial Intelligence (AAAI-06).

The Minimax algorithm propagates scores from leaf nodes via a process of alternating between maximizing and minimizing. This process confers some measure of filtering for pruning errors. However, such pruning errors propagate different in game-tree search based on several factors. For example, we have shown in chapter 7 that the player to move affects the propagation of forward pruning errors during game-tree search. This effect is a property of forward pruning in game-tree search and this suggests that forward pruning techniques should modify their behaviour depending on the player to move of a node.

We extend this work in this chapter to show that the depth of a node also affects how pruning errors are filtered [Lim and Lee, 2006a]. We build on Pearl's error propagation model [Pearl, 1984] for imperfect evaluation functions in game-tree search. This theoretical framework suggests a risk management strategy of pruning more near the root and less near the leaf nodes to maximize game performance. We present experimental

results on simulated and Chess game-trees that support this risk management strategy.

8.1 Intuition

For ease of explanation we use the *height* of a node u, defined as one less the number of nodes in the longest path from u to a leaf node. Depth and height of a node are closely related; a node of depth d is at height $h - d - 1$ for a game-tree of height h. To obtain some insights, we first consider the case of a single pruning error.

Proposition 1. *Assume a complete b-ary tree of height at least 3. Then the probability of a change in value of a random node at height k, selected with uniform probability from all nodes at that height, affecting the Minimax evaluation of its grandparent node at height $k + 2$ is no more than $\frac{1}{b}$.*

Proof. Consider the case where height k consists of Max nodes and a node at height k is chosen with uniform probability from all nodes at that height to have a change in value. We assume that the grandparent node g at height $k+2$ has the true value of v. We consider the case where the error decreases the node value first. If more than one child of g has value v, a single value reduction at depth k will not change g's value, so we consider the case where only one child, say m, has value v. Note that the value of g can only change if the value reduction occurs in m's children and not anywhere else. The probability of this occurring is no more than $1/b$, since m has b children. Now, consider the case where the error increases the node value. Consider any one of g's children, say m. Let m have value v. The value of the node m can change only in the case where only one of its children has value v and that child is corrupted by error. Hence the number of locations at height k that can change g's value is no more than b out of the b^2 nodes at that height.

The cases for Min nodes are the same when we interchange the error type. □

Proposition 1 can be recursively applied to show that the probability of an error propagating through l depths is no more than $1/b^{\lceil l/2 \rceil}$. However, the number of leaves in a b-ary tree of height l is b^{l-1}. If the probability of a leaf being mistakenly pruned is constant, the faster growth of the leaves will swamp the filtering effect, as we show in the next section.

8.2 Theoretical model for the propagation of error

Theoretical models have found that the Minimax algorithm can amplify the evaluation errors that occur at the leaf nodes. This is known as the *Minimax pathology* and was independently discovered by Nau [Nau, 1979] and Beal [Beal, 1980]. Pearl [Pearl, 1984] used a probabilistic game model, which we reproduce here, to examine this distortion and quantify the amplification of errors. Let p_k be the probability of a WIN for a node at height k and consider a uniform binary game-tree where the leaf nodes are either WIN or LOSS with probability p_0 and $1-p_0$, respectively. The leaf nodes also have an imperfect evaluation function that estimates the values with a bi-valued variable e, where $e = 1$ or $e = 0$ represent a winning and losing position, respectively. We denote the Minimax evaluation at position i as e_i and the WIN-LOSS status at position i as S_i. Positions where $i = 0$ represent leaf nodes. e_i and S_i are represented in negamax notation and refer to the player to move at position i. If node 3 has two children, 1 and 2, as in Figure 8.1, we write

$$S_3 = \begin{cases} L, & \text{if } S_1 = W \text{ and } S_2 = W, \\ W, & \text{otherwise} \end{cases} \tag{8.1}$$

$$e_3 = \begin{cases} 0, & \text{if } e_1 = 1 \text{ and } e_2 = 1, \\ 1, & \text{otherwise} \end{cases} \tag{8.2}$$

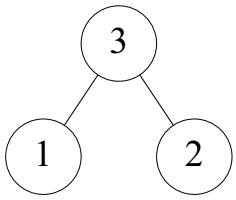


Figure 8.1: Representation of nodes in Pearl's model

Denote the probability of erroneously evaluating a LOSS position as winning and the probability of erroneously evaluating a WIN position as losing at position i as α_i and β_i respectively. In other words,

$$\alpha_0 = P(e = 1|S = L) \tag{8.3}$$

$$\beta_0 = P(e = 0|S = W) \tag{8.4}$$

The recurrence equations are:

$$\alpha_{k+1} = 1 - (1 - \beta_k)^2 \tag{8.5}$$

$$\beta_{k+1} = \frac{\alpha_k}{1 + p_k} \left[(1 - p_k)\alpha_k + 2p_k(1 - \beta_k)\right] \tag{8.6}$$

$$p_{k+1} = 1 - p_k^2 \tag{8.7}$$

Pearl also considered uniform *b-ary* game-trees where each non-leaf node has b successors, and similarly we obtain:

$$\alpha_{k+1} = 1 - (1 - \beta_k)^b \tag{8.8}$$

$$\beta_{k+1} = \frac{\left\{[p_k(1-\beta_k) + (1-p_k)\alpha_k]^b - [p_k(1-\beta_k)]^b\right\}}{1 - p_k^b} \tag{8.9}$$

$$p_{k+1} = 1 - p_k^b \tag{8.10}$$

Furthermore, p_k has three limit points [Pearl, 1984]:

$$\lim_{k\to\infty} p_{2k} = \begin{cases} 1 & \text{if } p_0 > \xi, \\ \xi & \text{if } p_0 = \xi, \\ 0 & \text{if } p_0 < \xi, \end{cases} \tag{8.11}$$

where ξ is the solution to $x^b + x - 1 = 0$. The limit points show that when the height of the tree is large enough, under this model, the outcome at the root is uncertain only for $p_0 = \xi$. Hence, we are mostly interested in the behaviour at these three limit points.

While Pearl's model assumed an imperfect evaluation function at the leaf nodes, we assume that the evaluation of leaf nodes are reliable, since we are considering only pruning errors. We adapt Pearl's model to understand the effects of forward pruning by considering α_0 and β_0 as the probabilities of pruning errors made at the frontier nodes (nodes that have leaf nodes as children). We can now demonstrate the effects of the depth of the node in forward pruning:

Theorem 2. *Assume that errors exists only at the leaves of uniform b-ary game trees. Then $\beta_{k+2}/\beta_k \leq b$ with $\beta_{k+2}/\beta_k = b$ for some cases. Similarly, $\alpha_{k+2}/\alpha_k \leq b$ with*

$\alpha_{k+2}/\alpha_k = b$ *for some cases.*

Proof. We recurse equations (8.8) and (8.9) once to get

$$\begin{aligned}\alpha_{k+2} = 1 - (1 - \frac{1}{1-p_k^b} \times \\ \{[p_k(1-\beta_k) + (1-p_k)\alpha_k]^b - [p_k(1-\beta_k)]^b\})^b\end{aligned} \tag{8.12}$$

$$\begin{aligned}\beta_{k+2} &= \frac{1}{1-p_{k+1}^b}\{\left[(1-p_k^b)(1-\beta_{k+1}) + p_k^b\alpha_{k+1}\right]^b \\ &\quad - \left[(1-p_k^b)(1-\beta_{k+1})\right]^b\}\end{aligned} \tag{8.13}$$

The value of α_{k+2} in equation (8.12) reaches its maximum value when $\beta_k = 0$. We also see that $\beta_{k+1} = 0$ when $\alpha_k = 0$ and therefore β_{k+2} in equation (8.13) reaches its maximum value when $\alpha_k = 0$.

We denote α_k when $\beta_0 = 0$ by α'_k. When $\beta_0 = 0$, we have $\beta_k = 0$ when k is even. α'_{k+2}/α'_k gives us the rate of increase in error propagation based on the value of α'_k:

$$\frac{\alpha'_{k+2}}{\alpha'_k} = \begin{cases} \frac{1-\left[1-(\alpha'_k)^b\right]^b}{\alpha'_k} & \text{if } p_k \to 0, \\ \frac{1-(1-\alpha'_k)^b}{\alpha'_k} & \text{if } p_k \to 1, \end{cases} \tag{8.14}$$

$$\lim_{\alpha_k \to 0} \frac{\alpha'_{k+2}}{\alpha'_k} = \begin{cases} 0 & \text{if } p_k \to 0, \\ b & \text{if } p_k \to 1, \end{cases} \tag{8.15}$$

Similarly, we denote β_k when $\alpha_0 = 0$ as β'_k:

$$\frac{\beta'_{k+2}}{\beta'_k} = \begin{cases} \frac{\left[1-(1-\beta'_k)^b\right]}{\beta'_k} & \text{if } p_k \to 0, \\ \frac{\left[1-(1-\beta'_k)^b\right]^b}{\beta'_k} & \text{if } p_k \to 1, \end{cases} \tag{8.16}$$

$$\lim_{\beta_k \to 0} \frac{\beta'_{k+2}}{\beta'_k} = \begin{cases} b & \text{if } p_k \to 0, \\ 0 & \text{if } p_k \to 1, \end{cases} \tag{8.17}$$

Lastly, the proof in Proposition 1 can be modified to show that $\beta_{k+2}/\beta_k \leq b$ and $\alpha_{k+2}/\alpha_k \leq b$ by considering one type of error instead of a single error. □

To help us gain more insight into the rate of error propagation, we simplify the analysis by considering uniform binary game trees. Setting $\beta_0 = 0$ and reapplying the recurrence equations for $b = 2$, we get

$$\alpha'_{k+2} = \frac{\alpha'_k}{1+p_k}\left[(1-p_k)\alpha'_k + 2p_k\right] \times \left\{2 - \frac{\alpha'_k}{1+p_k}\left[(1-p_k)\alpha'_k + 2p_k\right]\right\}. \tag{8.18}$$

Similarly, we can set $\alpha_0 = 0$ to get:

$$\beta'_{k+2} = \frac{\beta'_k(2-\beta'_k)}{2-p_k^2}\left\{p_k^2\left[1-(1-\beta'_k)^2\right] + 2(1-p_k^2)\right\} \tag{8.19}$$

Several interesting observations can be made from Figure 8.2, which shows the plot of α'_{k+2}/α'_k and β'_{k+2}/β'_k for the limit points of p_k. If $p_k \to 1$, errors are being filtered out when $\beta'_k < 1-\xi$, giving $\lim_{k\to\infty} \beta'_{2k} = 0$. However, when $\beta'_k > 1-\xi$, error rate will increase with the height giving $\lim_{k\to\infty} \beta'_{2k} = 1$. If $\beta'_k = 1-\xi$, the rate of error propagation is constant, $\lim_{k\to\infty} \beta'_{2k} = 1-\xi$. Similarly, when $p_k \to 0$, $\lim_{k\to\infty} \alpha'_{2k}$ is 0 when $\alpha'_k < \xi$, is 1 when $\alpha'_k > \xi$, and is ξ when $\alpha'_k = \xi$. For $p_k = \xi$, both types of errors

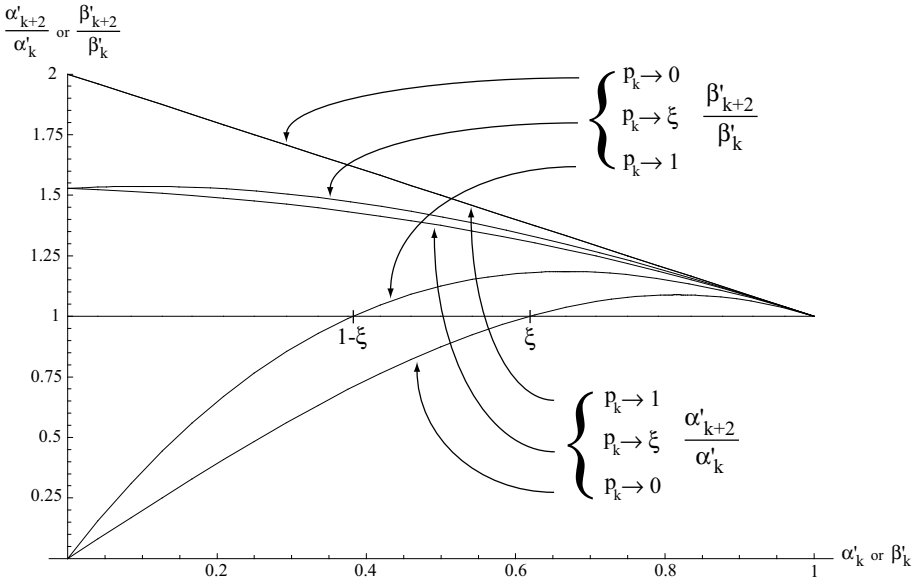


Figure 8.2: Rates of change in error propagation for $b = 2$

grow with the height. These results are also given in [Pearl, 1984] for errors caused by imperfect evaluation functions.

The upper bound on the rates of change in error propagation to the root of b by Theorem 2 is clearly too conservative as shown in Figure 8.3. For example, when $b = 2$, the maximum of β'_{k+2}/β'_k for $p_0 = \xi$ occurs when $\beta'_k \approx 0.099$ where $\beta'_{k+2}/\beta'_k \approx 1.537$, which is less than the bound of 2 that the theorem suggests.

8.3 Theoretical Optimal Forward Pruning Scheme

The theoretical model presented provides insights to the rate of forward pruning error propagation with respect to the depth of a node in game-tree search. Since $\frac{\alpha'_{k+2}}{\alpha'_k}$ and $\frac{\beta'_{k+2}}{\beta'_k}$ are greater than one for $p_0 = \xi$, this means that the rate of pruning error propagation will increase as the distance from the leaf node increases. The intuitive scheme is therefore to perform the most amount of forward pruning near the root and the least amount of

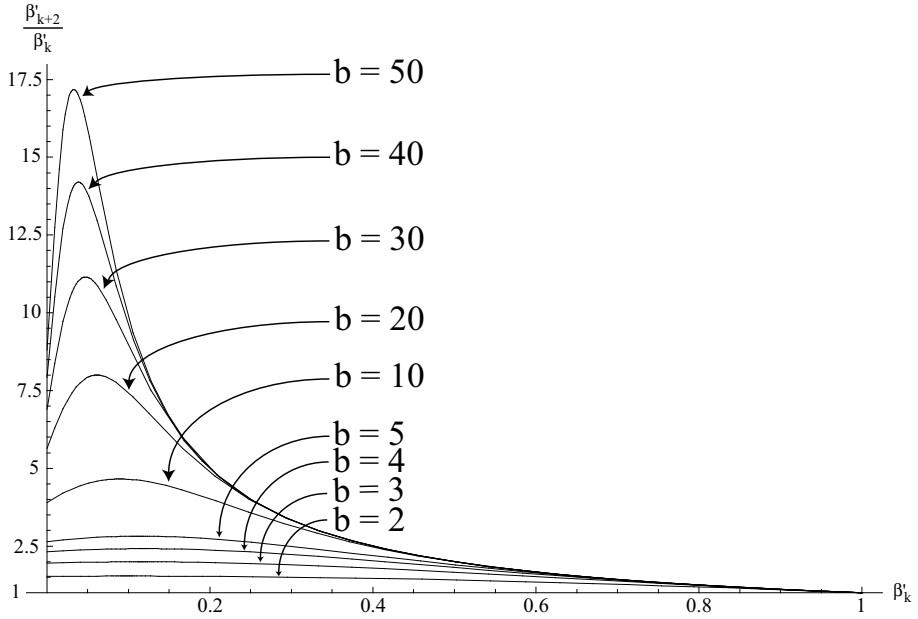


Figure 8.3: Plot of $\frac{\beta'_{k+2}}{\beta'_k}$ when $p_0 = \xi$ for various b

forward pruning near the leaf nodes. The theory, unfortunately, does not provide us with the *actual* risk management strategy to maximize game-playing performance.

In this section, we compute the theoretical optimal forward pruning scheme for a given error threshold with respect to the depth of the node. We first simplify the problem by assuming that the game-tree is of uniform breadth and depth. Furthermore, the probability of a forward pruning error is the same as the probability of that error propagating to the root. Formally, the simplifying assumptions are:

1. The game-tree is of uniform breadth b and height h

2. The forward pruning scheme is allowed to make at most $e \in [0, 1]$ errors

3. $w(n)$ is defined as the number of nodes pruned if node n is forward pruned and is equal to the size of a full-width subtree of height i where i is the height of the node n

4. $p(n)$ is the probability of node n being forward pruned wrongly and is equal to $P_i \in [0, 1]$, where i is the height of the node n

5. $f(n)$ is the probability of the pruning error propagating to the root and is equal to $F_i \in [0, 1]$, where i is the height of the node n

6. $P_i = F_i, \forall i$

We therefore want to maximize the number of nodes pruned

$$\sum_{n \in \text{Pruned Nodes}} w(n) \tag{8.20}$$

subject to $\sum_{n \in \text{Pruned Nodes}} p(n)f(n) = e$, which we can rewrite as

$$\sum_{i=0}^{h-1} P_i b^{h-i} b^i = \sum_{i=0}^{h-1} P_i b^h \tag{8.21}$$

subject to the constraint $\sum_{i=0}^{h-1} F_i b^{h-i}/b^{\lceil (h-i)/2 \rceil} = \sum_{i=0}^{h} P_i b^{h-i}/b^{\lceil (h-i)/2 \rceil} = e$. Note that the summation is for $i = 0, 1, \ldots, h-1$ as we do not forward prune the root node.

We now use lagrange multipliers to find the optimal solution by setting for all i:

$$\frac{\partial}{\partial P_i} \sum_{i=0}^{h-1} P_i b^h + \lambda (\sum_{i=0}^{h-1} P_i b^{h-i}/b^{\lceil (h-i)/2 \rceil} - e) = b^h + \lambda b^{\lfloor (h-i)/2 \rfloor} = 0 \tag{8.22}$$

which implies

$$\lambda = -b^{h - \lfloor (h-i)/2 \rfloor} \tag{8.23}$$

Since λ is equal for all P_i, this forces $i \geq h - 1$, and therefore the constraints show

that the optimal forward pruning scheme is to prune when $i = h - 1$, or only at the root.

8.4 Observing the Effect using Simulations

To illustrate the implications of our results, we once again perform a number of Monte Carlo simulations. In our experiments, we use game-trees with uniform branching factor 5 and branch-dependent leaf values to simulate actual game-trees. Each node is assigned a pruning probability q_i: during search, a Bernoulli trial with probability q_i of pruning each child is performed, where i is the depth of the node. We test two different pruning reduction schemes – Multiplicative and Linear pruning reduction. The multiplicative pruning reduction schemes multiply the pruning probability by a constant factor for every additional 2 depths, or $q_{i+2} = q_i \times c$, and $q_1 = q_0 \times c$, where c is the multiplicative factor. A multiplicative factor of 1.0 is equivalent to a Constant Pruning scheme. Linear pruning reduction schemes reduce q_i for each depth by subtracting a constant c from the previous depth, or $q_{i+1} = q_i - c$. Figure 8.4 shows the proportion of correct Minimax evaluations for various pruning reduction schemes with starting pruning probability $q_0 = 0.1$. We see that the linear pruning reduction schemes are clearly inadequate to prevent amplification of pruning errors propagating to the root, even though the linear pruning scheme of $c = 0.02$ reduces q_i to zero when at search depth 6.

While the experiments have shown that multiplicative pruning reduction schemes can prevent the amplification of pruning errors propagating to the root, it might be possible that multiplicative pruning reduction schemes are not pruning enough to justify forward pruning at all. It is more interesting to consider the question "Given a fixed time limit, what is the best pruning reduction scheme that allows the deepest search while making less than a pre-defined threshold of errors?".

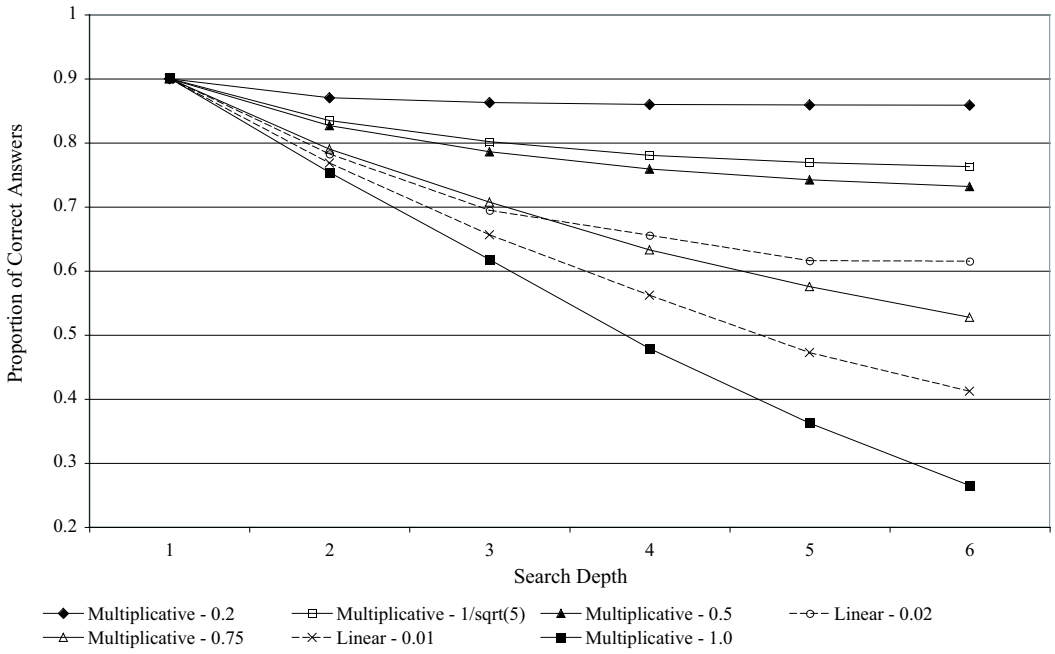


Figure 8.4: Comparison of various pruning reduction schemes

We set the error threshold at 0.25, which means that the search should return a back-up evaluation equal to the true Minimax value of the tree at least 75% of the time[1]. To simulate a fixed time limit, we used $\alpha\beta$ search and iterative deepening to search until the maximum number of nodes, which we set at 1000, were searched. We tested two additional pruning reduction schemes – Root Pruning and Leaf Pruning. In the Root pruning scheme, only the root node forward prunes, or $q_0 = c > 0$ and $q_i = 0$, for $i > 0$. The Leaf pruning scheme only forward prunes leaf nodes, or $q_i = 0$, for $i < d$ and $q_d = c > 0$, where d is the search depth. We first used 8 iterations of binary searches with 10^5 simulated game-trees each time to find pruning probabilities p_0 for the various pruning schemes that return correct answers at least 75% of the time. Next, we ran a simulation of 10^6 game-trees and, for each generated game-tree, we performed every

[1]In a real game, we would be able to use domain dependent information to decide when to prune. This is likely to result in a smaller pruning error rate for the same aggressiveness in pruning

pruning scheme with the pruning probabilities found using binary search.

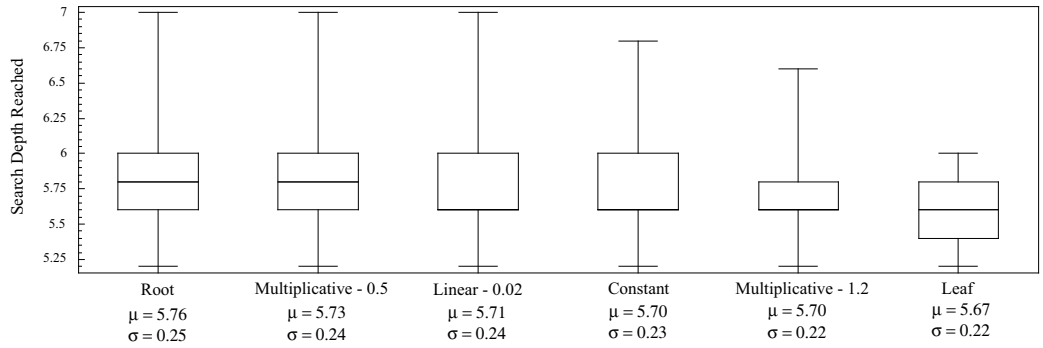


Figure 8.5: Box plot showing the search depths reached with correct answers by each pruning scheme

The Root pruning scheme is the best pruning scheme as it achieves, on average, the deepest search depths among all pruning schemes as shown in Figure 8.5. A box plot gives a five-number summary of: minimum data point, first quartile, median, third quartile, and maximum data point. The mean and standard deviation of the search depths reached for each pruning scheme are also given. These results suggest that pruning rate decrease with the depth of the nodes.

8.5 Observing the Effect using Chess Game-Trees

8.5.1 RANKCUT TOGA II

To see the effects of the depth of a node on pruning error propagation in Chess game-trees, experiments were done on 1,364 middle-game positions obtained from test suites using RANKCUT TOGA II.

We tested three pruning schemes, (1) pruning only at the root (RootOnly), (2) pruning constantly throughout the game-tree (Constant), and (3) pruning only at the leaf

nodes (LeafOnly). To observe the effect more clearly, we modified RANKCUT TOGA II to forward prune more by start forward pruning from depth 3 (instead of depth 7).

As RANKCUT TOGA II does not forward prune at the root node, and only starts forward pruning at depth 3 onwards, the RootOnly scheme prunes only in nodes at the level below the root, and LeafOnly prunes only in nodes at depth 3. The Constant scheme therefore prunes at nodes between the level below the root and at depth 3, inclusive. This does not affect the validity of the experiments as the various pruning schemes prune differently depending on the depth of the node, even though the root and leaf nodes do not perform forward pruning.

TOGA II and RANKCUT TOGA II use iterative deepening (Section 2.2.3) for real-time decisions. This means that the search depth reached within a time limit is variable. The 'true' Minimax value of the tree is assumed to be the value returned by TOGA II searched to the same depth achieved by RANKCUT TOGA II (with forward pruning). In addition, TOGA II and RANKCUT TOGA II use principal variation search (PVS) (Section 2.2.2) and aspiration search (Section 2.2.2) to accelerate Alpha-Beta search. As previous search results are used as estimates for future search, this results in complex interaction between pruning errors in previous iterations and search performance in future iterations. We therefore removed PVS and aspiration search by setting Alpha and Beta to ∞ and $-\infty$, respectively, for each function call to the search.

We set the error threshold at 0.25, which means that the search should return a back-up evaluation equal to the true Minimax value of the tree at least 75% of the time. We used 10 iterations of binary search between [0.0, 1.0] to find the optimal value of the pruning threshold for each of the three schemes on a set of 1,000 different middle-games, also obtained from test suites. We then performed every pruning scheme with the pruning probabilities found using binary search on 1,364 middle-game positions. We

tested two fixed time limits of 10 seconds and 5 seconds per position.

Experimental Results

For each pruning scheme, we record the number of correct solutions (as deemed by a search using TOGA II to the same search depth), and the depth achieved by RANKCUT TOGA II. To observe the gain in search depth achieved by forward pruning, we also record the depth achieved by TOGA II in the same time limit, and store the differences between the search depths reached by RANKCUT TOGA II and TOGA II.

Pruning Scheme	Pruning Threshold	# Correct	% Correct	$\mu_{\text{Search Depth Gain}}$	$\sigma_{\text{Search Depth Gain}}$
RootOnly	13.18%	1038	76.10%	0.08285	0.1050
LeafOnly	11.13%	1029	75.44%	-0.04762	0.1193
Constant	1.86%	1023	75.00%	-0.07136	0.1466

Table 8.1: Statistics for pruning schemes with time limit of 5 seconds per position

Source of Variation	SS	df	MS	F	P-value	F crit
Between Groups	14.25	2	7.12	57.67	$2.5693e^{-25}$	2.999
Within Groups	381.33	3087	0.1235			
Total	395.58	3089				

Table 8.2: One-way ANOVA to test for differences in search depth gain among the three pruning schemes with time limit of 5 seconds per position

Table 8.1 shows the statistics for pruning schemes with the time limit of 5 seconds per position. The RootOnly scheme is the most successful of the three, and is the only scheme that achieved higher search depths than TOGA II within the same time limit while achieving 75% accuracy of Minimax results. One-way Analysis of variance (ANOVA) is used in Table 8.2 to show that this result is statistically significant. Tukey's Honestly Significantly Different (HSD) comparison test revealed showed that the RootOnly scheme was significantly different from the other two schemes.

Similarly, Table 8.3 shows the statistics for pruning schemes with the time limit of

Pruning Scheme	Pruning Threshold	# Correct	% Correct	$\mu_{\text{Search Depth Gain}}$	$\sigma_{\text{Search Depth Gain}}$
RootOnly	12.99%	1019	74.71%	0.08145	0.1378
LeafOnly	10.45%	1023	75.00%	-0.07625	0.1468
Constant	1.66%	1014	74.34%	-0.09862	0.1818

Table 8.3: Statistics for pruning schemes with time limit of 10 seconds per position

Source of Variation	SS	df	MS	F	P-value	F crit
Between Groups	19.62	2	9.808	63.12	$1.381e^{-27}$	2.999
Within Groups	474.43	3053	0.1235			
Total	494.05	3053				

Table 8.4: One-way ANOVA to test for differences in search depth gain among the three pruning schemes with time limit of 10 seconds per position

10 seconds per position. The RootOnly scheme is again the most successful of the three, and is the only scheme that achieved higher search depths than TOGA II within the same time limit while achieving 75% accuracy of Minimax results. Table 8.4 shows that this result is statistically significant. In addition, Tukey's HSD test showed that the RootOnly scheme was significantly different from the other two schemes.

8.5.2 Learning to Forward Prune Experiments

We report results from Kocsis' PhD thesis [Kocsis, 2003] which introduces learning algorithms for forward pruning. As some of the experiments involved learning depth parameters that maximize performance of forward pruning, this presents additional evidence for the effect of the depth of the node on pruning error propagation during game-tree search. We recall the necessary background needed to formulate the relevant learning algorithm and describe experimental results from [Kocsis, 2003]. We will then explain these results using the effect of the depth of the node in forward pruning error propagation.

Forward-Pruning Vectors Consider a forward pruning algorithm that searches a subset of legal moves, with the size of the subset of moves fixed for each search depth. We can represent these maximum number of moves to consider for each search depth by a vector $\vec{v} = (v_0, v_1, \ldots, v_D)$. A vector which determines forward pruning decisions is termed a *forward-pruning vector* (FPV).

Optimization Problem We can formulate maximizing game-playing performance with forward pruning as an optimization problem. In other words, we want to maximize the game-playing performance given the parameters of a forward pruning algorithm, where the parameters are the amount of forward pruning to perform at each depth.

As shown in the previous sections, the depth of a node affects the propagation of forward pruning errors in game-tree search. Since game-playing programs use variants of Minimax search, it is therefore reasonable to expect that the depth of a node affects game-playing performance of forward pruning in such programs. Given our theoretical analysis and experiments on the effect of the depth of a node on error propagation in game-tree search, we expect to see optimal game-playing performance when forward pruning is done less with increasing search depths.

Formally, the problem is formulated [Kocsis, 2003] as: maximize *performance($\vec{v}$)* subject to

$$ncount(\vec{v}) \leq N$$

$$\vec{v} = (v_0, v_1, \ldots, v_D)$$

$$v_d \in W_d$$

$$d = \{0, 1, \ldots, D\}$$

where $ncount()$ is the size of the search tree, *performance($\vec{v}$)* is the quality of the move (which can be defined in several ways and is left undefined), D is the maximum search depth, W_d is the set of possible forward pruning parameters for depth d, and N is the maximum number of nodes allowed to be explored.

TS-FPV Algorithm

It is difficult to solve the optimization problem given in the last section by testing each FPV because it is a multivariate optimization problem. Kocsis introduces an algorithm called TS-FPV that is based on tabu search [Glover, 1989]. Like tabu search, TS-FPV is a local search and it selects FPVs from the neighborhood of the current FPV being considered. TS-FPV uses a list called a tabu list (TL) to record some of the recent FPVs previously investigated. These recent solutions are avoided (as they are "tabu").

There are two phases in the algorithm: *intensification* and *diversification*. Intensification tries to find neighboring solutions that improve the current FPV, where diversification explores untested neighborhoods of solutions. In TS-FPV, the frequency of each possible width at each depth is stored. During the diversification phase, candidate solutions are explored in descending order of the frequencies of each value of the current FPV. This is seen as a form of long-term memory.

Relevant FPV variants

We described a simple variant of FPV that uses one value to represent the amount of forward pruning to perform at each search depth in Section 8.5.2. [Kocsis, 2003] extends FPVs by mapping the values along an additional dimension - the iteration number. The iteration number is the current search iteration of an iterative deepening search (Section 2.2.3), which is commonly performed by real-time game-playing programs. The two

relevant FPV variants are: (1) FPV-d, which forward prunes the same amount at each search depth, and (2) FPV-l, which forward prunes the same amount at a certain distance to the leaf nodes.

To illustrate this, we interpret the forward pruning that occurs under the variants with the FPV $\{v_1, v_2, v_3, v_4, v_5\}$ while searching to depth 5.

1. At iteration 1, FPV-d forward prunes v_1 of nodes at the root. FPV-l forward prunes v_5 of nodes at the root.

2. At iteration 2, FPV-d forward prunes v_1 of nodes at the root, and v_2 at depth 2. FPV-l forward prunes v_4 of nodes at the root, and v_5 of at depth 2.

3. At iteration 3, FPV-d forward prunes v_1 of nodes at the root, v_2 at depth 2, and v_3 at depth 3. FPV-l forward prunes v_3 of nodes at the root, v_4 at depth 2, and v_5 at depth 3.

4. And so on.

Experimental Setup

TS-FPV is used to maximize performance of FPV-d and FPV-l. The experiments for TS-FPV are done using CRAFTY, which has 5 phases of move-generation, (1) moves from the transposition table, (2) capture moves, (3) killer moves, and (4) 3 of the remaining moves sorted by the History Heuristic, and (5) the remaining moves (Section 6.4.2 for more details). In the experiments, moves are considered for forward pruning only from phase 4 onwards, and are also re-ordered using the Neural MoveMap heuristic [Kocsis, 2003, Chapter 3].

3,000 randomly selected middle-game positions are used to measure the performance of the FPV variants. The performance is the average difference in scores between the

search with forward pruning and the score returned by a 12-ply search. In other words, the scores returned by the 12-ply search are considered the "true" Minimax value of the position. The result is statistically significant with 3,000 test positions. The limit on the number of nodes to expand, $ncount()$, is set to the number of nodes expanded by the search without forward pruning searched to the reference depth.

The possible FPV values in the experiments are {1, 2, 3, 4, 5, 10, 20, 30, 40, 50, all}, and the reference search depths are 4, 5, 6, 7 and 8. For each reference search depth and FPV variant, TS-FPV is used to learn the best FPV, where the search with forward pruning searches to one more the reference search depth.

Experimental Results

The top 3 FPVs of the various search depths for FPV-l and FPV-d are presented in Tables 8.5 and 8.6, respectively. In our discussions, we focus on the values of the FPVs with respect to the distance to the root or leaves. Interested users should refer to [Kocsis, 2003] for detailed analysis on the experimental results.

Due to the use of iterative deepening and Alpha-Beta search window enhancements of CRAFTY, it is not possible to exactly predict the exact forward pruning scheme that TS-FPV will find using our theoretical analysis of forward pruning error propagation in uniform game-trees with Minimax search. There are several reasons: (1) the Minimax values returned at lower iterations are used as estimates to Alpha-Beta search windows for later iterations and (2) positions evaluated incorrectly due to forward pruning errors might be stored and later reused in the transposition tables. There is therefore a complex interaction between the (correct or incorrect) scores returned by the search with forward pruning at lower iterations and the effect of these earlier scores in subsequent iterations.

There is, however, a fairly straightforward conjecture that theory suggests - the best

Reference Search Depth	FPV-l values	performance
4	all 2 5 20 10	1.04
4	5 4 20 10 10	0.82
4	5 2 5 all 10	0.79
5	30 2 5 30 20 10	0.74
5	10 10 30 5 5 20	0.46
5	5 2 10 all 20 10	0.45
6	5 10 all 10 10 all 20	1.08
6	10 10 all 5 10 all 20	1.02
6	40 5 20 5 10 all 20	0.76
7	all 3 all 30 10 10 all 20	0.25
7	10 10 10 all 20 10 all 20	0.09
7	10 4 10 10 all all all 20	0.09
8	5 2 10 20 30 20 all all 20	-0.16
8	all 3 20 40 10 10 all all 10	-0.19
8	5 2 10 all 20 30 all 20 20	-0.23

Table 8.5: Top 3 FPV-l values for various search depths [Kocsis, 2003]

Reference Search Depth	FPV-d values	performance
4	3 2 all 20 20	0.32
4	5 5 all 10 10	0.17
4	5 5 20 10 20	0.11
5	4 10 40 10 10 40	0.19
5	5 30 all 3 5 10	-0.17
5	5 5 all 10 20 all	-0.33
6	10 30 20 4 all 10 20	-0.09
6	5 5 20 30 all 20 20	-0.49
6	5 10 30 5 all 20 all	-0.77
7	all 5 20 all all 4 10 20	-0.96
7	all 20 20 40 10 10 20 5	-1.06
7	4 5 all 10 30 all all all	-1.12
8	10 10 all 10 10 all 30 all 20	-1.00
8	all all all 10 20 5 10 10 all	-1.07
8	30 10 30 10 10 all 20 all 20	-1.09

Table 8.6: Top 3 FPV-d values for various search depths [Kocsis, 2003]

pruning scheme is to prune more near the root and less near the leaves. The experimental results reported by Tables 8.5 and 8.6 provide further evidence to this conjecture. We consider the vector {5 10 all 10 10 all 20} where FPV-l achieved its best performance of 1.08 with reference search depth 6. The vector is of length 7 as FPV-l searches to depth 7 (one more than the reference depth). Note that the root at iteration 7 considers only at most 5 moves before forward pruning the remaining moves, and this is the most aggressive action of all iterations and search depths for this vector. Similarly, we consider the vector {3 2 all 20 20} where FPV-d achieved its best performance of 0.32 with reference search depth 4. Using this vector, FPV-d forward prunes most aggressively, considering only 2 moves, when at depth 2, and slightly less aggressively when at the root. The remaining top performing vectors exhibit this similar trend - forward pruning is done more aggressively near the root and less aggressively near the leaves.

8.6 Discussion

There is anecdotal evidence that supports our analysis of the effect of the depth of nodes in forward pruning. Adaptive Null-Move Pruning is a variant of Null-Move Pruning that essentially prunes less when near the leaf nodes. In developing Adaptive Null-Move Pruning [Heinz, 1999], it was initially expected that the best performance in an adaptive form of null-move pruning would come from pruning more when near the leaf nodes and less when closer to the root. This belief was consistent with the expectations of other researchers [Greenblatt et al., 1988, Goetsch and Campbell, 1990, Donninger, 1993] that the amount of forward pruning should increase with increasing distance of the node from the root of the tree. However, Heinz found that the opposite is true: Adaptive Null-Move Pruning works better by pruning less near the leaf nodes, which agrees with our results.

Heinz notes that this scheme is contrary to static forward pruning methods which prune more near the leaf nodes. We conjecture that static forward pruning methods can prune more when near the leaf nodes simply because the static evaluations become more accurate near the leaf nodes, and should prune less when far from the leaf nodes as they become inaccurate.

Other forward pruning techniques appear to avoid having to deal with the effect of the depth of a node on pruning error propagation by not forward pruning in the parts of the search tree near to the leaf nodes. For example, (1) RankCut, discussed in chapter 6, starts forward pruning only when the search depth is 7 or more, (2) Multi-Cut $\alpha\beta$-Pruning does not prune close to the horizon so as to reduce "the time overhead involved" [Björnsson and Marsland, 2001], (3) ProbCut [Buro, 1995b, Jiang and Buro, 2003] works only nodes at higher depths as it uses depth-reduced search to estimate search results for deep searches, and (4) History Pruning/Late Move Reduction as implemented in TOGA II does not forward prune when the depth is 4 or less.

8.7 Chapter Conclusion

Existing literature had painted the pessimistic picture that the Minimax algorithm corrupts the back-up values in the presence of errors. While there are alternative explanations for Minimax pathology, including but not limited to, [Smith and Nau, 1994, Sadikov et al., 2005, Lustrek et al., 2005], our analysis show that the Minimax algorithm is filtering pruning errors, but at a slower rate than the rate of growth of leaf nodes.

Since the rate at which the Minimax algorithm can filter out errors is smaller than the rate at which leaf nodes are introduced for each additional search depth, this suggests that forward pruning techniques should prune less as search depth increases. We

simulated game-trees with correlated leaf values, which have been shown to be non-pathological, and demonstrated that it is better to prune more aggressively near the root and less aggressively near the leaf nodes. Experimental data with Chess game-trees in RANKCUT TOGA II and CRAFTY also support this risk-management strategy.

The theoretical analysis therefore offers an explanation for the experimental results and the anecdotal optimal forward pruning scheme observed by Chess programmers. The risk management strategy of pruning more near the root and less near the leaf nodes should help to maximize performance of forward pruning techniques in game-tree search.

Chapter 9

Conclusion and Future Research

In this chapter, we begin by summarizing the work contained in this thesis, indicating the primary results. Finally, we adopt a broader perspective and discuss possible directions of future research.

9.1 Conclusion

Game-tree search has provided the foundation to solve computationally-hard problems such as playing board games at high levels within standard time controls and determining game-theoretic values of games. The successes of game-tree search can be at least partially attributed to a refinement to the Minimax paradigm called the Alpha-Beta algorithm. The Alpha-Beta algorithm eliminates portions of the game-tree that will never be selected by a Minimax player, and is able to significantly reduce the game-tree complexity. In addition, innovative search enhancements such as transposition tables, move ordering and search extensions, have led to higher game-playing performance. Nevertheless, the exponential growth of the game-tree complexity for games with high branching factors still overwhelms computational limits of modern computers.

The goal of our work has been to improve the state-of-the-art for forward pruning techniques in game-tree search. This thesis has therefore focused on novel applications of forward pruning techniques and better understanding of the theoretical properties of forward pruning in game-tree search. In this thesis, we have (1) solved the game of Tigers and Goats, a high game-tree complexity problem, by using forward pruning techniques in forward searches, (2) introduced an effective forward pruning technique called RankCut that can be applied alongside existing forward pruning techniques, and (3) shown that two factors, namely the player to move and the depth of a node, affect forward pruning error propagation in game-tree search and suggested risk management strategies for forward pruning techniques to maximize game-playing performance.

9.1.1 Tigers and Goats

In the absence of human expert knowledge, we used co-evolved neural networks as a move ordering and forward pruning heuristic for Goat in the game of Tigers and Goats. This heuristic enabled us to show that the game is a draw for Goats. We have therefore shown that co-evolutionary computing is capable of creating heuristics in the absence of human expert knowledge for use in a search to find the game-theoretic result of a two-player zero sum game with perfect information.

By pre-computing as much subproblems as can be handled efficiently both in memory and disk, we then showed that the game is at least a draw for Tigers, thus proving that the game of Tigers and Goats is a draw under optimal play. In light of this result, we were able to retrospectively incorporate aggressive forward pruning techniques to re-confirm the result using three days of computational time.

9.1.2 RankCut

RankCut is designed to be a domain-independent forward pruning technique that exploits the move ordering that current game-playing programs typically perform for efficiency in Alpha-Beta search. RankCut can be implemented within an existing Alpha-Beta search, and we successfully implemented RankCut in CRAFTY and TOGA II, both open-source Chess-playing programs. We have shown using test suites and matches to show that RankCut is able to improve game-playing performance, even when implemented alongside existing forward pruning techniques.

9.1.3 Properties of Forward Pruning in Game-Tree Search

The theoretical analysis of forward pruning in game-tree search in chapters 7 and 8 provide frameworks to understand how pruning errors propagate in game-tree search. The frameworks show that, in uniform game-trees, the player to move and the depth of a node in game-tree search affects the pruning error propagation. In particular, we showed that pruning errors in Max nodes are less likely to result in poor move decisions compared to pruning errors in Min nodes. Furthermore, if the probability of pruning errors at each node is fixed, the rate of pruning errors propagating to the root increases with increasing depth of the node.

Due to the inherent difficulty of theoretically analyzing "real" game-trees, we conducted experiments in simulated and Chess game-trees, and the results agree with the intuitive extensions of the theoretical frameworks to "real" game-trees - in other words, results suggest that (1) pruning more in Max nodes than in Min nodes performs better than pruning more in Min nodes than in Max nodes, and (2) pruning more at the root and less near the leaves produces the best game-playing performance.

9.2 Future Research

In this thesis, we presented work that successfully applied novel forward pruning techniques to game-tree search in computationally hard search problems. We predict that forward pruning, or selective search, will play bigger roles in search performance as the search problems becoming even more computationally demanding. The long-term goal of selective search research is to match and even surpass the intuitive ability of humans to selectively search the state-space and yet be able to make accurate decisions; for the short to medium term, we can suggest the following areas of future research.

9.2.1 Tigers and Goats

While we have weakly solved the game of Tigers and Goats, it is always desirable to be able to strongly solve a game and to play it perfectly. Strongly solving Tigers and Goats is possible with current state of the art machines by emulating [Romein and Bal, 2003] and completely enumerating the state-space of Tigers and Goats.

There are other variants of Tigers and Goats that have different winning criterions. It would be interesting to investigate and find the game-theoretical values of these variants.

9.2.2 RankCut

One potential problem is that RankCut assumes the statistics of $\Pi(\vec{f_i})$ collected without forward pruning remain the same when forward pruning. While our experiments indicate that the assumption is reasonable for practical purposes, one possible solution is to recollect the statistics with forward pruning until the probabilities stabilize. However, this approach needs experiments to verify its effectiveness.

$\Pi(\vec{f_i})$ was estimated using the relative frequency of a better move appearing. The

choice of using relative frequency is intentional to ensure ease of understanding and programming. Nevertheless, it is possible to model $\Pi(\vec{f_i})$ using other statistical distributions, such as a binomial distribution, or even a bayesian graphical model. Alternatively, if the threshold t is pre-defined, the problem can be viewed as a classification problem, and any classification methods such as neural networks, support vector machines, or decision trees can be used for decision-making. Future research can be done to test if there is better way to represent $\Pi(\vec{f_i})$.

We have shown in our experiments that RankCut is effective in practical Chess-playing programs. In these experiments, we used an intuitive set of features that have worked well. However, the experiments have not determined which features are most important. This is useful as fewer features will likely require less training data before RankCut is able to work well.

Lastly, RankCut is able to identify when moves generated beyond a certain point are not likely to affect the final score, and therefore we believe that RankCut will benefit game-playing programs most in games with large branching factor and where good or bad moves are easily identifiable. One candidate is the game of Go where the number of legal moves ranges from 100–360 for the opening phase and most of the middle-game. Another example is the game of Abalone, a two-player game invented in 1990 by Laurent Levi and Michel Lalet. Abalone has an average branching factor of 80 [Aichholzer et al., 2002] which places the game-tree complexity of Abalone between those of Chess and Go.

We have done some initial experiments in the game of Abalone by training a neural network as an leaf evaluation function and implemented RankCut together with Alpha-Beta search. Due to the large branching factor of Abalone, Alpha-Beta search without forward pruning could only search up to an average of 4 plies in 1 minute, but is able to

search up to an average of 8 plies in 1 minute with RankCut. Under fixed time limits of 1 minute per move, Alpha-Beta search with RankCut was able to beat the commercial version of ABA-PRO [Aichholzer et al., 2002], arguably the strongest Abalone-playing entity, at a fixed playing level of 9 by +12 -5 =3 in a 20-game series, whereas Alpha-Beta search without RankCut lost handily by +2 -15 =5 to ABA-PRO.

More recently, joint work with Cheng Wei Chang and Wee Sun Lee have resulted in a Abalone program called ABA-CUT [Chang, 2007] that plays the game Abalone using hard-coded heuristics and RankCut. A finely-tuned version of ABA-CUT was able to defeat ABA-PRO, fixed playing level of 8, +5 -0 =1. In addition, experiments using ABA-CUT with and without RankCut show performance difference in self-play games.

However, as these games were not under tournament conditions, these results at best suggest that RankCut is effective in other games. More research needs to be done to show the effectiveness of RankCut in games with high branching factor.

9.2.3 Properties of Forward Pruning in Game-Tree Search

We have shown that two factors, the depth and the player to move of a node, affect the rate of forward pruning error propagation in game-tree search. For each factor, we also determined the risk management strategy to use that minimizes the amount of pruning error propagation. Some possible areas of research are to (1) theoretically derive and empirically find the best pruning schemes for each factor, (2) analyze the interaction of both factors in actual game-tree search, and (3) discover other factors that affect pruning error propagation.

The game of Go can be considered as the grand challenge of game AI at this point in time. One interesting development in computer Go has been the introduction of Monte

Carlo methods that combine game-tree search and randomly generated moves for evaluation [Coulom, 2006, Kocsis and Szepesvári, 2006]. The random nature of Monte Carlo methods corresponds well with the theoretical analysis of the properties of forward pruning presented in this thesis, and should extend to Monte Carlo tree search. More research on how to incorporate risk management strategies in forward pruning can be done to further improve the state of the art for Monte Carlo tree search.

Appendix A

Additional Tigers and Goats Endgame Database Statistics

		Tiger to Move			
Goat to Move	Outcome	Tiger Wins	Draw	Goat Wins	Total
	Tiger Wins	23,217,329	11,249	652	23,229,230
	Draw	6,584,426	1,459,852	1,509	8,045,787
	Goat Wins	667,879	98,308	146,966	913,153
	Total	30,469,634	159,409	149,127	32,188,170

Table A.1: Statistics of database S5 (4 goats captured)

		Tiger to Move			
Goat to Move	Outcome	Tiger Wins	Draw	Goat Wins	Total
	Tiger Wins	1,923,410	4,954	132	1,928,496
	Draw	3,879,335	2,342,280	4,743	6,226,358
	Goat Wins	457,474	570,870	286,767	1,315,111
	Total	6,260,219	2,918,104	291,642	9,469,965

Table A.2: Statistics of database S4 (3 goats captured)

	Tiger to Move				
Goat to Move	Outcome	Tiger Wins	Draw	Goat Wins	Total
	Tiger Wins	23,355	575	11	23,941
	Draw	354,808	838,197	6,226	1,199,231
	Goat Wins	87,558	515,197	279,768	882,523
	Total	465,721	1,353,969	286,005	2,105,695

Table A.3: Statistics of database S3 (2 goats captured)

	Tiger to Move				
Goat to Move	Outcome	Tiger Wins	Draw	Goat Wins	Total
	Tiger Wins	63	25	0	88
	Draw	3,541	73,551	3,614	80,706
	Goat Wins	2,848	123,961	125,572	252,381
	Total	6,452	197,537	129,186	333,175

Table A.4: Statistics of database S2 (1 goat captured)

	Tiger to Move				
Goat to Move	Outcome	Tiger Wins	Draw	Goat Wins	Total
	Tiger Wins	35	21	4	60
	Draw	59	1,505	1,248	2,812
	Goat Wins	52	7,942	22,615	30,609
	Total	146	9,468	23,867	33,481

Table A.5: Statistics of database S1 (0 goats captured)

Appendix B

The mathematics of counting applied to Tigers and Goats

Consider the 5×5 Tigers and Goats board, with grid points numbered 1 to 25, and the 8 symmetry transformations that permute the set $D = \{1, \ldots, 25\}$ as shown in Figure B.1. We identify a board position with a function $f : D \rightarrow \{Tiger, Goat, Empty\} = \{T, G, E\}$ that assigns to each grid point its status, subject to the constraint that there are exactly 4 Tigers and a number of Goats that varies from 16 to 20. We show how to compute the number of distinct board positions, modulo symmetry, for the case of 20 Goats, which we abbreviate as $4T1E$, i.e. 4 Tigers and 1 empty spot. Any symmetry permutation of the square board can be obtained by some sequence of flips around the axes as shown in Figure B.1. The shaded areas at right are used in the analysis of rotational symmetries.

Let $S = \{f : D \rightarrow \{T, G, E\} | \text{f assumes 4 values T, 20 values G, 1 value E}\}$ Observation: the group G of permutations of the domain D induces permutations on the set S of functions. Def: two functions f and g are equivalent iff there is a permutation P in G

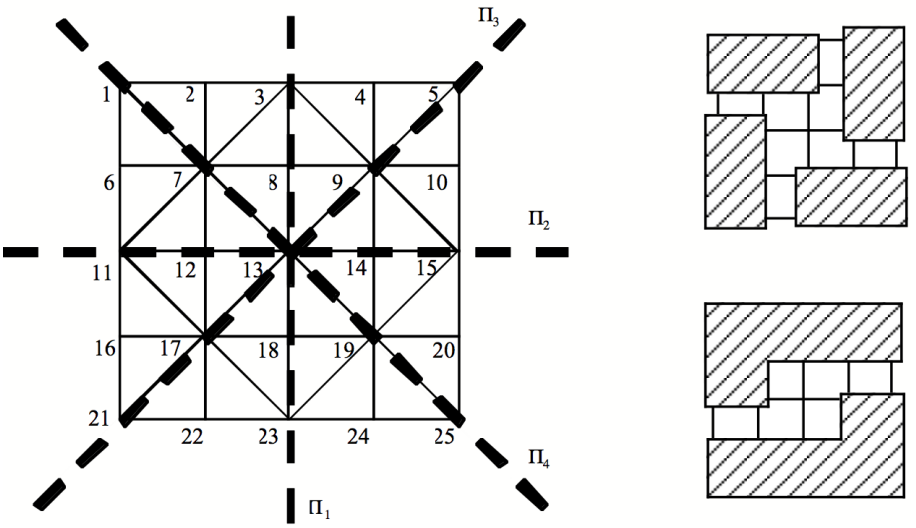


Figure B.1: Symmetry permutations of the Tigers and Goats board

such that $f = P(g)$. Def: a function f is invariant under permutation P iff $f = P(f)$. Def: let $I(P)$ be the number of functions f in S that are invariant under P.

Burnside's Lemma. *The number of equivalence classes of S under G is $\frac{1}{|G|} \cdot \sum I(P)$, where the sum is taken over all permutations P in G·.*

In order to evaluate Burnside's formula we investigate, for each of the 8 permutations of G, how many board positions remain invariant.

- Identity permutation, '1'. Any board position remains invariant. There are 25 possibilities to place the empty spot, multiplied by $\binom{24}{4}$ possibilities to place the 4 tigers, resulting in 265,650 board positions invariant under the identity permutation.

- Rotation by $90°$ or by $270°$. The empty spot must be on the center point 13, and each of the 4 Tigers must be placed in "his own" 2×3 area (shaded in Figure B.1, top right), such that the 4 chosen spots are symmetric under rotation (e.g. on spots

2, 10, 24, 16). Since the location of 1 Tiger determines the place of the other 3 as well, there are 6 board positions invariant under rotations by $\pm 90°$.

- Rotation by $180°$. The empty spot must be on the center point 13. 2 Tigers can be placed anywhere in the upper angular area consisting of 12 spots, shaded in Figure B.1, bottom right. The location of the other 2 Tigers is then determined by symmetry, resulting in $\binom{12}{2} = 66$ invariant board positions.

- Flipping around the vertical or horizontal axis through the center. The empty spot can be anywhere along the axis. i.e. in 5 different places. After the empty spot has been placed, three cases of Tiger placements must be distinguished:

  a 4 Tigers located on the flipping axis: there is only 1 way to place them

  b 2 Tigers located on the flipping axis ($\binom{4}{2}$ ways to place them), 1 Tiger in each half of the remaining board (10 ways to place the first Tiger, the other is determined by symmetry)

  c 2 Tigers in each half of the remaining board ($\binom{10}{2}$ ways to place the first 2, the other 2 are determined by symmetry). In total we have $5 \times (1 + 10 \binom{4}{2} + \binom{10}{2}) = 530$ invariant board positions.

- Flipping around either diagonal axis through the center. This turns out to be the same as flipping around the vertical or horizontal axis, resulting in 530 invariant board positions.

Summing up all these invariants and dividing by $|G|$ yields:

1 Identity	$25 \times \binom{24}{4}$ = 265,650
2 Rotations $\pm 90°$	6
1 Rotations $180°$	$\binom{12}{2}$ = 66
4 Flips	530
8 Permutations	267,848

Number of equivalence classes

$=$ number of inequivalent board positions for 4 Tigers and 20 Goats

$$= \tfrac{267{,}848}{8} = 33,481.$$

The calculations for 16 to 19 Goats are similar but more complicated, resulting in the following numbers of inequivalent board positions:

4T 20G 1E	33,481
4T 19G 2E	333,175
4T 18G 3E	2,105,695
4T 17G 4E	9,469,965
4T 16G 5E	32,188,170

In each case, the number of inequivalent board positions is roughly $\frac{1}{8}$ of the number of distinct positions if symmetry is ignored.

Appendix C

Implementing Retrograde Analysis for Tigers and Goats

The first step in retrograde analysis is to initialize all easily recognizable terminal positions. It is simple to identify terminal nodes as they can only be of three types:

- If there are 16 goats with at least 1 goat in immediate danger of being captured, then Tiger to Move, Tiger wins.
- If Tiger has no legal moves, then Tiger to Move, Tiger Loses.
- If Goats has no legal moves, then Goat to Move, Goat Loses.

After the initialization has set these terminal positions, it sets the value of all remaining positions to a draw. An iterative process can then correctly determine the correct value of all remaining positions. Note that if a position is lost for the player to move, all of the successors of the position can be marked as wins for the opponent. Similarly, if a position is won for the player to move, all preceding positions are "potential" losses for the opponent. However, they are only true losses if all of their successors are also won

for the player to move.

The retrograde analysis algorithm [Gasser, 1996, Wu and Beal, 2002] can be summarized now as follows:

- Initialize the database by determining the number of successors for each position. If the node is terminal, we compute its value and set its value appropriately, and its status to known. Otherwise, the number of successors is stored, and its status is set to unknown.

- Perform multiple passes through the database. The database is traversed and for each node with known status, all preceding positions are notified of its value. Each predecessor updates its score if it is improved by the child. Repeat iterating through the database until no score changes to any position occurs during a complete round.

C.1 Indexing Scheme

Note that with k similar pieces which are unlabeled, the $k!$ arrangements of k similar, labeled pieces on the board are equivalent. The index range can therefore be reduced by a factor of $k!$. There are $\binom{q}{k} = \frac{q!}{k!(q-k)!}$ placements of k similar pieces on q squares. Let the position of k similar pieces be $\{p_1, p_2, \ldots, p_i\}$ where $p_1 < p_2 < \ldots < p_i$ and $p_i \in [0, q-1]$. A space-efficient indexing scheme is then given by the following algorithm [Nalimov et al., 2000] as shown in Pseudocode 10.

The tigers were selected as the pieces to be constrained to select a "canonical" board. We define a canonical board by choosing the board with the lowest index of tigers after performing the 8 symmetric operations (identity, rotation by 90, 180 and 270 degrees, reflection on the x-axis, y-axis and the two diagonals) on the board. There are

Pseudocode 10 index($\{p_1, p_2, \ldots, p_i\}$)

```
1:  index ← 0
2:  while k > 0 do
3:      while p_1 ≠ 0 do
4:          index ← index + C(q−1, k−1)
5:          q ← q − 1
6:          for i ← 1, 2, ..., k do
7:              p_i ← p_i − 1
8:      k ← k − 1, q ← q − 1
9:      for i ← 1, 2, ..., k do
10:         p_i ← p_{i+1} − 1
```

1,666 unique canonical forms for all possible tiger configurations. Note that there are $\binom{25}{4} = 12,650$ possible tiger configurations. To reduce running time, the indices of the canonical boards for each tiger configuration can be pre-computed and stored in an array of size 12,650. Table C.1 shows the different total index size for different number of goats on the board. There are still some board positions which are symmetric to each other within the indexing scheme. This occurs when the tigers are symmetric under a certain operation (e.g. reflection), but since all combinations of goats are generated, there will be combinations of goats which are also symmetric under the same operation but are assigned different indices.

	Number of Goats	Index Size	Space Complexity
S_5	16	33,901,434	32,188,170
S_4	17	9,971,010	9,469,965
S_3	18	2,215,780	2,105,695
S_2	19	349,860	333,175
S_1	20	34,986	33,481

Table C.1: Comparison of index size with actual space complexity

C.1.1 Inverse Operation

During the construction of the endgame databases, the board needs to be accessed and evaluated. However, if we store the board separately in the database, the storage of each separate board will take approximately 50 bits, since the naive method is to use 2 bits for each board position. This separate board representation will greatly increase the space requirements of the endgame databases. This storage requirement can be eliminated by introducing an inverse operator for the indexing scheme, where we can get the position of pieces by providing the index, the number of pieces and the total number of squares:

Pseudocode 11 reconstruct($index$, $numPieces$, $totalSquares$)

```
1:  k ← numPieces
2:  q ← totalSquares − 1
3:  for i ← 1, 2, ..., numPieces do
4:      p_i ← 0
5:  while k > 0 do
6:      if C(q−1, k−1) ≤ index then
7:          while C(q−1, k−1) ≤ index do
8:              index ← index − C(q−1, k−1)
9:              q ← q − 1
10:             for i ← numPieces − k, numPieces − k + 1, ..., numPieces do
11:                 p_i ← p_i + 1
12:     k ← k − 1, q ← q − 1
13:     for i ← 1, 2, ..., k do
14:         p_i ← p_{i+1} + 1
```

Appendix D

RankCut Experimental Setup

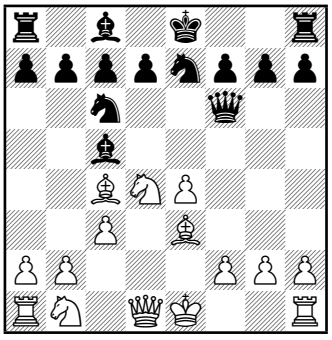
- Test suites "Encyclopedia of Chess Middlegames" (ECM, 879 positions), "Win at Chess" (WAC, 300 positions), and "1001 Winning Chess Sacrifices" (WCS, 1001 positions).
- LCT II Test. `http://perso.wanadoo.fr/lefouduroi/test_lct_native.htm`
- Nunn Positions. `http://www.Chessbaseusa.com/fritz5/nunnmtch.htm`
- Nunn-II Positions. `http://www.computerschach.de/index.php?option=com_remository&Itemid=52&func=fileinfo&id=3`
- The hardware used was a Apple PowerMac Dual 1.8 GHz PowerPC G5 with 2.25 GB Ram.
- All versions of CRAFTY and TOGA II used the default settings. No endgame database was used.

- Pondering was turned off. CPU time was used during test suites and elapsed time was used during matches.

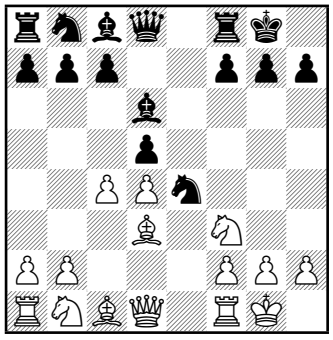
Appendix E

Chess Openings

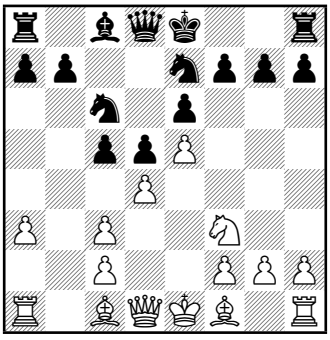
E.1 32 Openings from [Jiang and Buro, 2003]



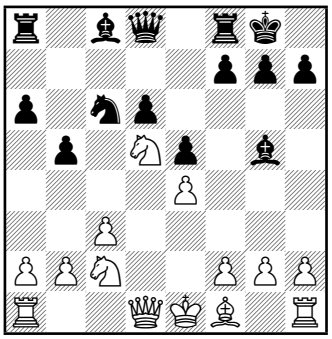
1 e4 e5 2 ♘f3 ♘c6 3 d4 e×d4 4 ♘×d4 ♗c5 5 ♗e3 ♕f6 6 c3 ♘ge7 7 ♗c4



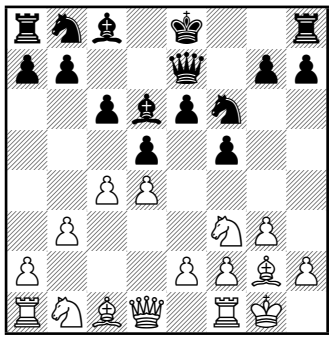
1 e4 e5 2 ♘f3 ♘f6 3 ♘×e5 d6 4 ♘f3 ♘×e4 5 d4 d5 6 ♗d3 ♗d6 7 O-O O-O 8 c4



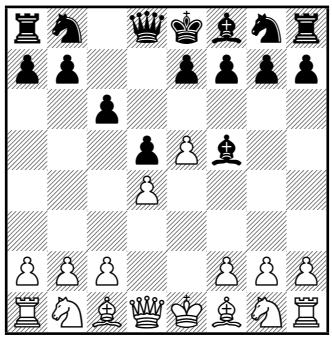
1 e4 e6 2 d4 d5 3 ♘c3 ♗b4 4 e5 c5 5 a3 ♗×c3+ 6 b×c3 ♘e7 7 ♘f3 ♘bc6



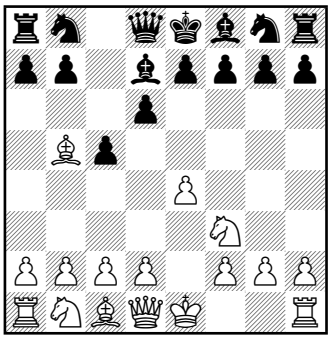
1 e4 c5 2 ♘f3 ♘c6 3 d4 c×d4 4 ♘×d4 ♘f6 5 ♘c3 e5 6 ♘db5 d6 7 ♗g5 a6 8 ♘a3 b5 9 ♘d5 ♗e7 10 ♗×f6 ♗×f6 11 c3 ♗g5 12 ♘c2 O-O



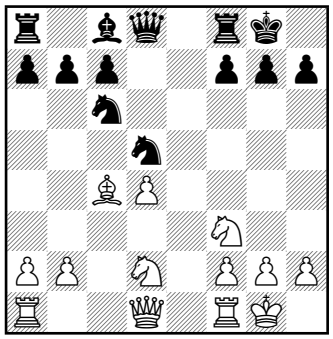
1 d4 f5 2 c4 ♘f6 3 ♘f3 e6 4 g3 d5 5 ♗g2 c6 6 O-O ♗d6 7 b3 ♕e7



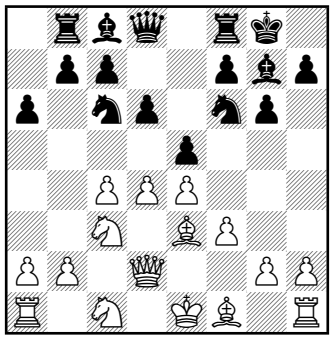
1 e4 c6 2 d4 d5 3 e5 ♗f5



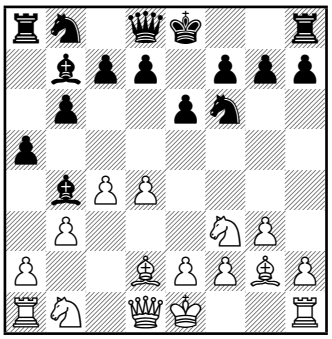
1 e4 c5 2 ♘f3 d6 3 ♗b5+ ♗d7



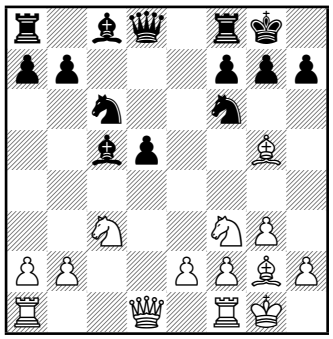
1 e4 e5 2 ♘f3 ♘c6 3 ♗c4 ♗c5 4 c3 ♘f6 5 d4 e×d4 6 c×d4 ♗b4+ 7 ♗d2 ♗×d2+ 8 ♘b×d2 O-O 9 O-O d5 10 e×d5 ♘×d5



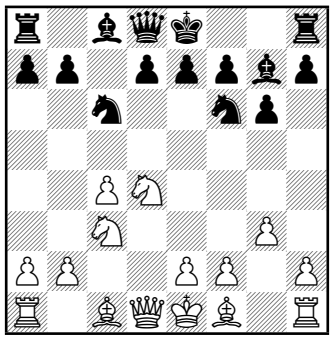
1 d4 ♘f6 2 c4 g6 3 ♘c3 ♗g7 4 e4 d6 5 f3 O-O 6 ♗e3 ♘c6 7 ♘ge2 a6 8 ♕d2 ♖b8 9 ♘c1 e5



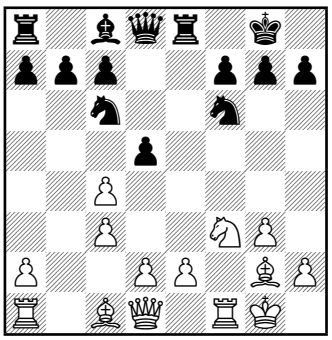
1 d4 ♘f6 2 c4 b6 3 ♘f3 e6 4 g3 ♗a6 5 b3 ♗b7 6 ♗g2 ♗b4+ 7 ♗d2 a5



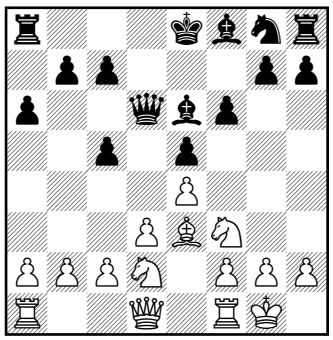
1 d4 d5 2 c4 e6 3 ♘c3 c5 4 c×d5 e×d5 5 ♘f3 ♘f6 6 g3 ♘c6 7 ♗g2 ♗e7 8 O-O O-O 9 d×c5 ♗×c5 10 ♗g5



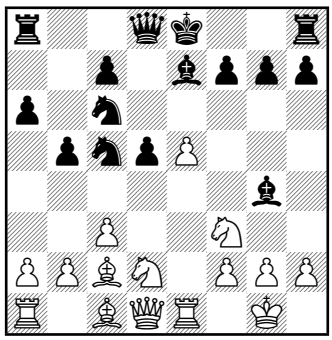
1 c4 c5 2 ♘c3 ♘c6 3 ♘f3 ♘f6 4 d4 c×d4 5 ♘×d4 g6 6 g3 ♗g7



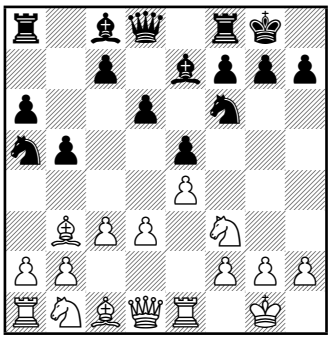
1 c4 e5 2 ♘c3 ♘f6 3 g3 ♘c6 4 ♗g2 ♗b4 5 ♘f3 O-O 6 O-O e4 7 ♘g5 ♗×c3 8 b×c3 ♖e8 9 f3 e×f3 10 ♘×f3 d5



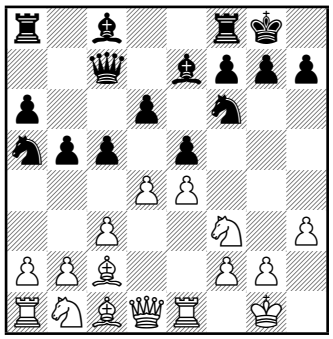
1 e4 e5 2 ♘f3 ♘c6 3 ♗b5 a6 4 ♗×c6 d×c6 5 O-O ♕d6 6 d3 f6 7 ♗e3 c5 8 ♘bd2 ♗e6



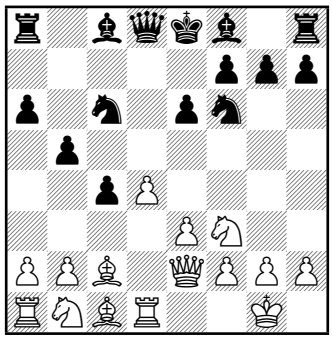
1 e4 e5 2 ♘f3 ♘c6 3 ♗b5 a6 4 ♗a4 ♘f6 5 O-O ♘×e4 6 d4 b5 7 ♗b3 d5 8 d×e5 ♗e6 9 c3 ♘c5 10 ♗c2 ♗g4 11 ♖e1 ♗e7 12 ♘bd2



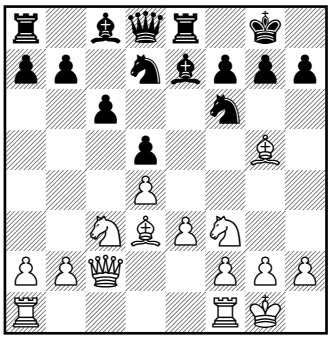
1 e4 e5 2 ♘f3 ♘c6 3 ♗b5 a6 4 ♗a4 ♘f6 5 O-O ♗e7 6 ♖e1 b5 7 ♗b3 O-O 8 d3 d6 9 c3 ♘a5



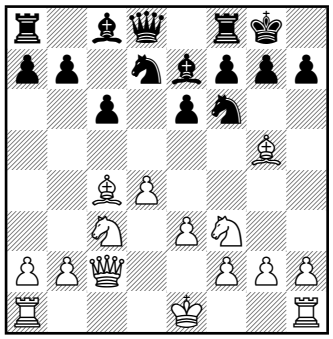
1 e4 e5 2 ♘f3 ♘c6 3 ♗b5 a6 4 ♗a4 ♘f6 5 O-O ♗e7 6 ♖e1 b5 7 ♗b3 d6 8 c3 O-O 9 h3 ♘a5 10 ♗c2 c5 11 d4 ♕c7



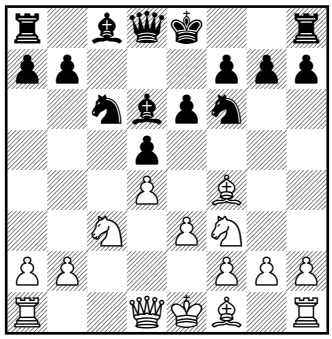
1 d4 d5 2 c4 d×c4 3 ♘f3 ♘f6 4 e3 e6 5 ♗×c4 c5 6 O-O ♘c6 7 ♕e2 a6 8 ♖d1 b5 9 ♗b3 c4 10 ♗c2



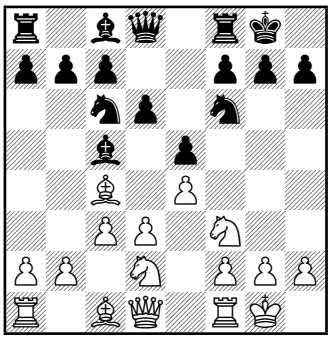
1 d4 d5 2 c4 e6 3 ♘c3 ♘f6 4 c×d5 e×d5 5 ♗g5 ♗e7 6 ♕c2 c6 7 e3 O-O 8 ♗d3 ♘bd7 9 ♘f3 ♖e8 10 O-O



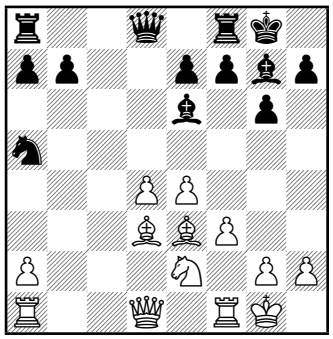
1 d4 d5 2 c4 e6 3 ♘c3 ♘f6 4 ♗g5 ♗e7 5 ♘f3 ♘bd7 6 e3 O-O 7 ♕c2 c6 8 ♗d3 d×c4 9 ♗×c4



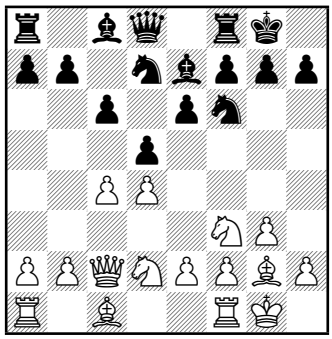
1 d4 d5 2 c4 c6 3 ♘c3 ♘f6 4 c×d5 c×d5 5 ♘f3 ♘c6 6 ♗f4 e6 7 e3 ♗d6



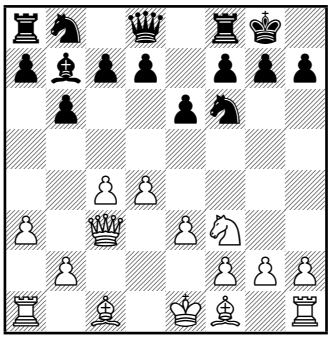
1 e4 e5 2 ♘f3 ♘c6 3 ♗c4 ♗c5 4 d3 d6 5 O-O ♘f6 6 c3 O-O 7 ♘bd2



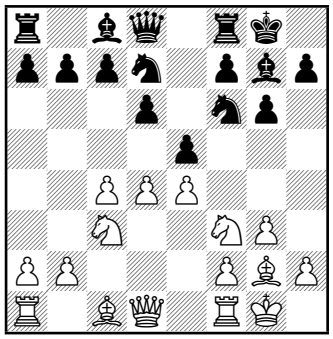
1 d4 ♘f6 2 c4 g6 3 ♘c3 d5 4 c×d5 ♘×d5 5 e4 ♘×c3 6 b×c3 ♗g7 7 ♗c4 c5 8 ♗e3 ♘c6 9 ♘e2 O-O 10 O-O ♗g4 11 f3 ♘a5 12 ♗d3 c×d4 13 c×d4 ♗e6



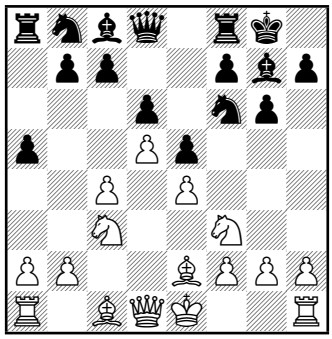
1 d4 ♘f6 2 c4 e6 3 g3 d5 4 ♗g2 ♗e7 5 ♘f3 O-O 6 O-O c6 7 ♕c2 ♘bd7 8 ♘bd2



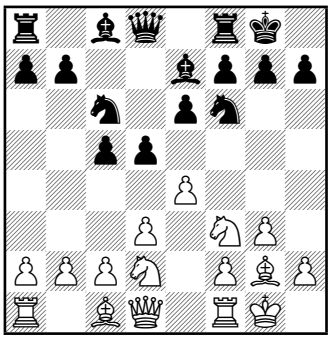
1 d4 ♘f6 2 c4 e6 3 ♘c3 ♗b4 4 ♕c2 O-O 5 a3 ♗×c3+ 6 ♕×c3 b6 7 ♘f3 ♗b7 8 e3



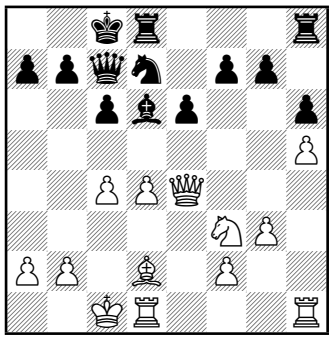
1 d4 ♘f6 2 c4 g6 3 ♘c3 ♗g7 4 ♘f3 O-O 5 g3 d6 6 ♗g2 ♘bd7 7 O-O e5 8 e4



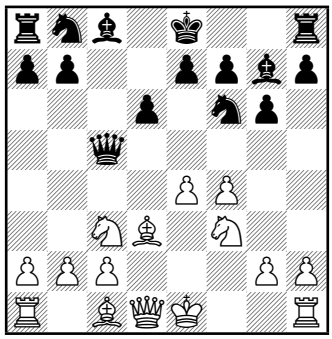
1 d4 ♘f6 2 c4 g6 3 ♘c3 d6 4 e4 ♗g7 5 ♘f3 O-O 6 ♗e2 e5 7 d5 a5



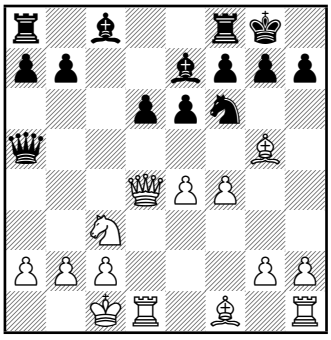
1 ♘f3 d5 2 g3 c5 3 ♗g2 ♘c6 4 d3 e6 5 O-O ♘f6 6 ♘bd2 ♗e7 7 e4 O-O



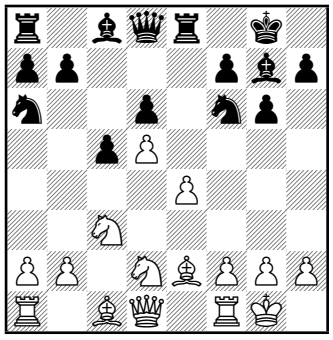
1 e4 c6 2 d4 d5 3 ♘c3 d×e4 4 ♘×e4 ♗f5 5 ♘g3 ♗g6 6 h4 h6 7 ♘f3 ♘d7 8 h5 ♗h7 9 ♗d3 ♗×d3 10 ♕×d3 ♕c7 11 ♗d2 ♘gf6 12 O-O-O e6 13 ♘e4 O-O-O 14 g3 ♘×e4 15 ♕×e4 ♗d6 16 c4



1 e4 d6 2 d4 g6 3 ♘c3 ♗g7 4 f4 ♘f6 5 ♘f3 c5 6 d×c5 ♕a5 7 ♗d3 ♕×c5



1 e4 c5 2 ♘f3 ♘c6 3 d4 c×d4 4 ♘×d4 ♘f6 5 ♘c3 d6 6 ♗g5 e6 7 ♕d2 ♗e7 8 O-O-O O-O 9 f4 ♘×d4 10 ♕×d4 ♕a5



1 d4 ♘f6 2 c4 c5 3 d5 e6 4 ♘c3 e×d5 5 c×d5 d6 6 e4 g6 7 ♘f3 ♗g7 8 ♗e2 O-O 9 O-O ♖e8 10 ♘d2 ♘a6

Bibliography

[Abramson, 1989] Abramson, B. (1989). Control Strategies for Two-Player Games. *ACM Computing Survey*, 21(2):137–161.

[Aichholzer et al., 2002] Aichholzer, O., Aurenhammer, F., and Werner, T. (2002). Algorithmic fun - Abalone. *Special Issue on Foundations of Information Processing of TELEMATIK*, 1:4–6.

[Akl and Newborn, 1977] Akl, S. G. and Newborn, M. M. (1977). The Principle Continuation and the Killer Heuristic. In *ACM Annual Conference*, pages 466–473.

[Allen, 1989] Allen, J. D. (1989). A Note on the Computer Solution of Connect-Four. In Levy, D. N. and Beal, D. F., editors, *Heuristic Programming in Artificial Intelligence 1: the First Computer Olympaid*, pages 134–135. Ellis Horwood, Chichester, England.

[Allis, 1994] Allis, L. V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, Masstricht, The Netherlands.

[Allis et al., 1994] Allis, L. V., van der Meulen, M., and van den Herik, H. J. (1994). Proof-Number Search. *Artificial Intelligence*, 66:91–124.

[Anantharaman, 1990] Anantharaman, T. (1990). *A Statistical Study of Selective Min-Max Search in Computer Chess*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA.

[Anantharaman et al., 1990] Anantharaman, T., Campbell, M. S., and Hsu, F. (1990). Singular extensions: Adding selectivity to brute-force searching. *Artificial Intelligence*, 43(1):99–109.

[Baudet, 1978] Baudet, G. M. (1978). *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA.

[Baxter et al., 1998] Baxter, J., Tridgell, A., and Weaver, L. (1998). KnightCap: A Chess Program That Learns by Combining TD(lambda) with Game-Tree Search. *Fifteenth International Conference on Machine Learning*, pages 28–36.

[Beal, 1980] Beal, D. F. (1980). An Analysis of Minimax. In *Advances in Computer Chess 2*, pages 103–109. Edinburgh University Press.

[Beal, 1989] Beal, D. F. (1989). Experiments with the null move. In *Advances in Computer Chess 5*, pages 65–79. Elsevier Science.

[Berlekamp et al., 2001] Berlekamp, E., Conway, J., and Guy, R. (2001). *Winning Ways For Your Mathematical Plays*, volume 4 volumes. A.K. Peters, 2nd edition edition.

[Berliner, 1974] Berliner, H. J. (1974). *Chess as Problem Solving: The Development of a Tactics Analyzer*. PhD thesis, Carnegie-Mellon University.

[Birmingham and Kent, 1977] Birmingham, J. A. and Kent, P. (1977). Tree searching and tree pruning techniques. *Advances in Computer Chess 1*, pages 89–107.

[Björnsson and Marsland, 2000] Björnsson, Y. and Marsland, T. A. (2000). Learning search control in adversary games. In van den Herik and Monien, editors, *Advances in Computer Games 9*, pages 147–164. University of Maastricht.

[Björnsson and Marsland, 2000a] Björnsson, Y. and Marsland, T. A. (2000a). Risk Management in Game-Tree Pruning. *Information Science*, 122(1):23–41.

[Björnsson and Marsland, 2000b] Björnsson, Y. and Marsland, T. A. (2000b). Selective depth-first search methods. In van den Herik, H. J. and Iida, H., editors, *Games in AI Research*, pages 31–46.

[Björnsson and Marsland, 2001] Björnsson, Y. and Marsland, T. A. (2001). Multi-cut alpha-beta-pruning in game-tree search. *Theoretical Computer Science*, 252(1–2):177–196.

[Björnsson and Newborn, 1997] Björnsson, Y. and Newborn, M. (1997). Kasparov versus Deep Blue: Computer Chess Comes of Age. *International Computer Chess Association Journal*, 20(2):92–92.

[Björnsson and van den Herik, 2005] Björnsson, Y. and van den Herik, H. J. (2005). The 13^{th} World Computer-Chess Championship. *International Computer Games Association Journal*, 28(3).

[Bourzutschky et al., 2005] Bourzutschky, M., Tamplin, J., and Haworth, G. (2005). Chess endgames: 6-man data and strategy. *Theoretical Computer Science*, 349(2):140–157.

[Bouzy, 2003] Bouzy, B. (2003). Monte Carlo Go developments. In *Advances in Computer Games. Many Games, Many Challenges*, pages 159–174. Kluwer.

[Bouzy, 2005] Bouzy, B. (2005). Associating domain-dependent knowledge and Monte Carlo approaches within a Go program. *Information Sciences*, 175(4):247–257.

[Brockington, 1997] Brockington, M. (1997). *Asynchronous Parallel Game-Tree Search*. PhD thesis, Department of Computing Science, University of Alberta.

[Buro, 1995a] Buro, M. (1995a). ProbCut: An effective selective extension of the $\alpha - \beta$ algorithm. *International Computer Chess Association Journal*, 18(2):71–76.

[Buro, 1995b] Buro, M. (1995b). ProbCut: An effective selective extension of the alpha-beta algorithm. *International Computer Chess Association Journal*, 18(2):71–76.

[Buro, 1997a] Buro, M. (1997a). The Othello Match of the Year: Takeshi Murakami vs. Logistello. *International Computer Chess Association Journal*, 20(3):189–193.

[Buro, 1997b] Buro, M. (1997b). The Othello Match of the Year: Takeshi Murakami vs. Logistello. *International Computer Chess Association Journal*, 20(3):189–193.

[Buro, 1997c] Buro, M. (1997c). Toward opening book learning. Technical Report 2, NECI.

[Buro, 1998] Buro, M. (1998). From simple features to sophisticated evaluation functions. Technical Report 60, NECI.

[Buro, 1999] Buro, M. (1999). Experiments with Multi-ProbCut and a new high-quality evaluation function for Othello. In van den Herik, H. J. and Iida, H., editors, *Games in AI Research*. Institute for Knowledge and Agent Technology IKAT, Universiteit Maastricht, Maastricht, The Netherlands.

[Cazenave, 2001] Cazenave, T. (2001). Abstract proof search. In Marsland, T. A. and Frank, I., editors, *Computers and Games 2000, Lecture Notes in Computer Science*, pages 40–55. Springer, New York.

[Chang, 2007] Chang, C. W. (2007). Searching game trees with high branching factor: Abalone. Undergraduate Research Opportunity Program (UROP) Project Report (NUS).

[Chellapilla and Fogel, 1999] Chellapilla, K. and Fogel, D. (1999). Evolving Neural Networks to Play Checkers without Expert Knowledge. *IEEE Transactions on Neural Networks*, 10(6):1382–1391.

[Chellapilla and Fogel, 2001a] Chellapilla, K. and Fogel, D. (2001a). Evolving an Expert Checkers Playing Program without using Human Expertise. *IEEE Transactions on Evolutionary Computation*, 5(5):422–428.

[Chellapilla and Fogel, 2001b] Chellapilla, K. and Fogel, D. (2001b). Evolving an Expert Checkers Playing Program without Using Human Expertise. *IEEE Transactions on Evolutionary Computation*, 5(4):422–428.

[Chong et al., 2003] Chong, S. Y., Ku, D. C., Lim, H. S., Tan, M. K., and White, J. D. (2003). Evolved Neural Networks Learning Othello Strategies. In *Congress on Evolutionary Computation (CEC'03)*, pages 2222–2229.

[Condon and Thompson, 1983] Condon, J. H. and Thompson, K. (1983). Belle. In Frey, P. W., editor, *Chess Skill in Man and Machine*, pages 201–210. Springer, 2nd edition.

[Coulom, 2006] Coulom, R. (2006). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In Ciancarini, P. and van den Herik, H. J., editors, *Proceedings of the 5th International Conference on Computers and Games*, Turin, Italy.

[Dayan et al., 2001] Dayan, P., Schraudolph, N. N., and Sejnowski, T. J. (2001). *Learning to evaluate Go positions via temporal difference methods*, pages 74–96. Springer Verlag.

[Donninger, 1993] Donninger, C. (1993). Null move and deep search: Selective search heuristics for obtuse Chess programs. *International Computer Chess Association Journal*, 16(3):137–143.

[Fogel, 2002] Fogel, D. (2002). *Blondie24: Playing at the Edge of AI*. Academic Press, London, UK.

[Fotland, 2004] Fotland, D. (2004). Go Intellect Wins 19 x 19 Go Tournament. *ICGA Journal*, 27(3):131–141.

[Gasser, 1996] Gasser, R. (1996). Solving Nine-Man's-Morris. *Computational Intelligence*, 12(1):24–41.

[Gelly and Silver, 2007] Gelly, S. and Silver, D. (2007). Combining online and offline knowledge in uct. In *Proceedings of ICML 2007*.

[Gelly et al., 2006] Gelly, S., Wang, Y., Munos, R., and Teytaud, O. (2006). Modication of uct with patterns in monte-carlo go. Technical Report RR-6062, INRIA.

[Glover, 1989] Glover, F. (1989). Tabu Search - Part I. *ORSA Journal of Computing*, 1(3):190–206.

[Goetsch and Campbell, 1990] Goetsch, G. and Campbell, M. S. (1990). Experiments with the Null-Move Pruning. In Marsland, T. and Schaeffer, J., editors, *Computers, Chess, and Cognition*, pages 159–168. Springer-Verlag.

[Greenblatt et al., 1988] Greenblatt, R. D., Eastlake, D. E., and Crocker, S. D. (1988). The Greenblatt Chess Program. In *Computer Chess Compendium*, pages 56–66. Springer-Verlag.

[Heinz, 1998a] Heinz, E. A. (1998a). DarkThought Goes Deep. *International Computer Chess Association Journal*, 21(4):228–244.

[Heinz, 1998b] Heinz, E. A. (1998b). Extended Futility Pruning. *International Computer Chess Association Journal*, 21(2):75–83.

[Heinz, 1999] Heinz, E. A. (1999). Adaptive Null-Move Pruning. *International Computer Chess Association Journal*, 22(3):123–132.

[Heinz, 2000] Heinz, E. A. (2000). *Scalable Search in Computer Chess*. Vieweg Verlag.

[Heinz, 2001a] Heinz, E. A. (2001a). New self-play results in computer Chess. In Frank, I. and Marsland, T. A., editors, *Proceedings of the 2nd International Conference CG2000, Lecture Notes in Computers Science 2063*, pages 262–272. Springer.

[Heinz, 2001b] Heinz, E. A. (2001b). Self-Play Experiments in computer Chess revisited. In van den Herik, J. and Monien, B., editors, *Advances in Computer Chess 9*, pages 73–91.

[Hopp and Sanders, 1995] Hopp, H. and Sanders, P. (1995). Parallel game tree search on SIMD machines. In *Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 349–361.

[Iida et al., 2002] Iida, H., Sakuta, M., and Rollason, J. (2002). Computer Shogi. *Artificial Intelligence*, 134:121–144.

[Irving et al., 2000] Irving, G., Donkers, H. H. L. M., and Uiterwijk, J. W. H. M. (2000). Solving Kalah. *International Computer Games Association Journal*, 23(3):139–148.

[Jiang and Buro, 2003] Jiang, A. and Buro, M. (2003). First Experimental Results of ProbCut Applied to Chess. In van den Herik, H. J., Iida, H., and Heinz, E. A., editors, *Advances in Computer Games Conference 10*, pages 19–31.

[Junghanns et al., 1997] Junghanns, A., Schaeffer, J., Brockington, M., Bjornsson, Y., and Marsland, T. (1997). Diminishing returns for additional search in Chess. In van den Herik, J. and Uiterwijk, J., editors, *Advances in Computer Chess 8*, pages 53–67. Univ. of Rulimburg.

[Kendall and Whitewell, 2001] Kendall, G. and Whitewell, G. (2001). An Evolutionary Approach for the Tuning of A Chess Evaluation Function using Population Dynamics. In *2001 IEEE Congress on Evolutionary Computation (CEC 2001)*, pages 995–1002.

[Kishimoto and Müller, 2004] Kishimoto, A. and Müller, M. (2004). A general solution to the graph history interaction problem. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 644–649.

[Knuth, 1975] Knuth, D. E. (1975). Estimating the efficiency of backtrack programs. *Math of Comp 29*, pages 121–136.

[Knuth and Moore, 1975] Knuth, D. E. and Moore, R. W. (1975). An analysis of Alpha-Beta pruning. *Artificial Intelligence*, 6:293–326.

[Kocsis, 2003] Kocsis, L. (2003). *Learning Search Decisions*. PhD thesis, Universiteit Maastricht, IKAT/Computer Science Department.

[Kocsis and Szepesvári, 2006] Kocsis, L. and Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. In *ECML 2006*, pages 282–293.

[Levy and Newborn, 1991] Levy, D. and Newborn, M. (1991). *How Computers Play Chess*. Computer Science Press.

[Levy et al., 1989] Levy, D. N. L., Broughton, D. C., and Taylor, M. (1989). The SEX algorithm in computer Chess. *International Computer Chess Association Journal*, 12(1):10–21.

[Lim and Nievergelt, 2004] Lim, Y. and Nievergelt, J. (2004). Computing Tigers and Goats. *International Computer Games Association Journal*, 27(3):131–141.

[Lim and Lee, 2006a] Lim, Y. J. and Lee, W. S. (2006a). Properties of Forward Pruning in Game-Tree Search. In *Proceedings of Twenty-First National Conference on Artificial Intelligence (AAAI-06)*.

[Lim and Lee, 2006b] Lim, Y. J. and Lee, W. S. (2006b). RankCut – A Domain Independent Forward Pruning Method for Games. In *Proceedings of Twenty-First National Conference on Artificial Intelligence (AAAI-06)*.

[Lincke, 2002] Lincke, T. R. (2002). *Exploring the Computational Limits of Large Exhaustive Search Problems*. PhD thesis, ETH Zurich, Swiss.

[Lustrek et al., 2005] Lustrek, M., Gams, M., and Bratko, I. (2005). Why Minimax Works: An Alternative Explanation. In *IJCAI*, pages 212–217.

[Luuberts and Miikkulainen, 2001] Luuberts, A. and Miikkulainen, R. (2001). Co-Evolving a Go-Playing Neural Network. In Belew, R. and Juille, H., editors, *2001 Genetic and Evolutionary Computation Conference Workshop Program (GECCO-2001, San Francisco, CA)*, pages 14–19. San Francisco, CA: Kaufmann.

[Marsland, 1983] Marsland, T. A. (1983). Relative Efficiency of Alpha-Beta Implementations. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-83)*, pages 763–766.

[Marsland, 1986] Marsland, T. A. (1986). A Review of Game-Tree Pruning. *International Computer Chess Association Journal*, 9(1):3–19.

[Marsland and Popowich, 1985] Marsland, T. A. and Popowich, F. (1985). Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(4):442–452.

[McAllester, 1988] McAllester, D. A. (1988). Conspiracy Numbers for Min-Max Searching. *Artificial Intelligence*, 35:287–310.

[Moriarty and Miikkulainen, 1994] Moriarty, D. and Miikkulainen, R. (1994). Evolving Neural Networks to Focus Minimax Search. In *Proceedings of 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 1371–1377.

[Moriarty and Miikulainen, 1995] Moriarty, D. and Miikulainen, R. (1995). Discovering Complex Otherllo Strategies Through Evolutionary Neural Networks. *Connection Science*, 7(3–4):195–209.

[Müller, 2002] Müller, M. (2002). Computer Go. *Artificial Intelligence*, 134(1–2):145–179.

[Nalimov et al., 2000] Nalimov, E. V., Haworth, G. M., and Heinz, E. A. (2000). Space-efficient indexing of Chess endgame tables. *International Computer Games Association Journal*, 23(3):148–162.

[Nau, 1979] Nau, D. S. (1979). *Quality of decision versus depth of search on game-trees*. PhD thesis, Duke University.

[Nau, 1982] Nau, D. S. (1982). An investigation of the causes of pathology in games. *Artificial Intelligence*, 19:257–278.

[Newborn, 1977] Newborn, M. M. (1977). The efficiency of the alpha-beta search on trees with branch-dependent terminal node scores. *Artificial Intelligence*, 8:137–153.

[Nievergelt, 2000] Nievergelt, J. (2000). Exhaustive search, combinatorial optimization and enumeration: Exploring the potential of raw computing power. *Sofsem 2000 - Theory and Practice of Informatics*, Springer LNCS Vol. 1963:18–35.

[Pearl, 1984] Pearl, J. (1984). *Heuristics – Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Publishing Co., Reading, MA.

[Plaat, 1996] Plaat, A. (1996). *Research Re: search & Re-search*. PhD thesis, Erasmus University Rotterdam, Rotterdam, Netherlands.

[Plaat et al., 1996] Plaat, A., Schaeffer, J., Pijls, W., and de Bruin, A. (1996). Exploiting Graph Properties of Game Trees. In *13th National Conference on Artificial Intelligence*, volume 1, pages 234–239.

[Polya, 1937] Polya, G. (1937). Kombinatorische Anzahlbestimmungen fuer Gruppen, Graphen und Chemische Verbindungen. *Acta Mathematica*, 68:145–253.

[Reinefeld, 1983] Reinefeld, A. (1983). An Improvement of the Scout Tree Search Algorithm. *International Computer Chess Association Journal*, 6(4):4–14.

[Reinefeld, 1989] Reinefeld, A. (1989). *Spielbaum Suchverfahren*, volume Informatik-Fachberichte 200. Springer-Verlag, New York, NY. In german.

[Reinefeld and Marsland, 1987] Reinefeld, A. and Marsland, T. A. (1987). A Quantitative Analysis of Minimax Window Search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 951–954.

[Rijswijck, 2000] Rijswijck, J. V. (2000). Are Bees better than Fruitflies? (Experiments with a Hex playing program). *AI'00: Advances in Artificial Intelligence, 13th biennial Canadian Society for Computational Studies of Intelligence (CSCI) Conference*, pages 13–25.

[Rollason, 2000] Rollason, J. (2000). SUPER SOMA - Solving Tactical Exchanges in Shogi without Tree Searching. In *Computers and Games: Proceedings CG2000*.

[Romein and Bal, 2003] Romein, J. and Bal, H. (2003). Solving the Game of Awari using Parallel Retrograde Analysis. *IEEE Computer*, 36(10):26–33.

[Rosin and Belew, 1997] Rosin, C. D. and Belew, R. K. (1997). New methods for competitive coevolution. *Evolutionary Computation*, 5(1):1–29.

[Sadikov et al., 2005] Sadikov, Bratko, and Kononenko (2005). Bias and Pathology in Minimax Search. *TCS: Theoretical Computer Science*, 349.

[Schaeffer, 1986] Schaeffer, J. (1986). *Experiments in Search and Knowledge*. PhD thesis, University of Waterloo.

[Schaeffer, 1989] Schaeffer, J. (1989). The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-11(11):1203–1212.

[Schaeffer, 1990] Schaeffer, J. (1990). Conspiracy Numbers. *Artificial Intelligence*, 43:67–84.

[Schaeffer, 1997] Schaeffer, J. (1997). *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer, New York.

[Schaeffer et al., 2005] Schaeffer, J., Björnsson, Y., Burch, N., Kishimoto, A., Müller, M., Lake, R., Lu, P., and Sutphen, S. (2005). Solving Checkers. *IJCAI-05*, pages 292–297.

[Schaeffer et al., 2007] Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Muller, M., Lake, R., Lu, P., and Sutphen, S. (2007). Checkers is solved. *Science*. to appear.

[Schaeffer et al., 1992] Schaeffer, J., Culberson, J., Treloar, N., Knight, B., Lu, P., and Szafron, D. (1992). A World Championship Caliber Checkers Program. *Artificial Intelligence*, 53(2–3):273–290.

[Seo et al., 2001] Seo, M., Iida, H., and Uiterwijk, J. W. H. M. (2001). The PN*-search algorithm: Application to Tsume-Shogi. *Artificial Intelligence*, 129(1–2):253–277.

[Slagle and Dixon, 1969] Slagle, J. H. and Dixon, J. K. (1969). Experiments with some programs that search game trees. *Journal of the ACM*, 16(2):189–207.

[Slate and Atkin, 1977] Slate, D. J. and Atkin, L. R. (1977). Chess 4.5 - the Northwestern University Chess Program. In Frey, P., editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag.

[Smith and Nau, 1994] Smith, S. J. J. and Nau, D. S. (1994). An analysis of forward pruning. In *Proceedings of 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 1386–1391.

[Thompson, 1982] Thompson, K. (1982). Computer Chess Strength. In Clarke, M., editor, *Advances in Computer Chess 3*, pages 55–56.

[Thompson, 1986] Thompson, K. (1986). Retrograde analysis of certain endgames. *International Computer Chess Association Journal*, 9(3):131–139.

[Thomsen, 2000] Thomsen, T. (2000). Lambda-search in game trees – With application to Go. *International Compuer Games Association Journal*, 23(4):203–217.

[Uiterwijk and van den Herik, 2000] Uiterwijk, J. W. H. M. and van den Herik, H. J. (2000). The advantage of the initiative. *Information Sciences*, 122(1):43–58.

[van den Herik et al., 2002] van den Herik, H. J., Uiterwijk, J. W. H. M., and van Rijswijck, J. (2002). Games Solved: Now and in the Future. *Artificial Intelligence*, 134(1–2):277–311.

[Wágner and Virág, 2001] Wágner, J. and Virág, I. (2001). Solving Renju. *International Computer Games Association Journal*, 24(1):30–34.

[Wu and Huang, 2005] Wu, I.-C. and Huang, D.-Y. (2005). A New Family of k-in-a-row Games. In *The 11th Advances in Computer Games Conference (ACG'11)*, Taipei, Taiwan.

[Wu and Beal, 2002] Wu, R. and Beal, D. (2002). A Memory Efficient Retrograde Algorithm and Its Application To Chinese Chess Endgames. In *More Games of No Chance*, volume 42, pages 213–227. MSRI Publications.

[Zobrist, 1969] Zobrist, A. L. (1969). A Hashing Method with Applications for Game Playing. Technical Report Tech. Rep. 88, Computer Sciences Department, University of Wisconsin.